FROST Implementation Security Assessment

Coinbase, Inc.

November 22, 2021 – Version 1.0

Prepared for

Jeff Barksdale

Prepared by

Marie-Sarah Lacharité

Giacomo (Jack) Pope

Executive Summary

Synopsis

During the autumn of 2021, Coinbase, Inc. engaged NCC Group's Cryptography Services Practice to conduct a security assessment of their FROST implementation in Go. The FROST (Flexible Round-Optimized Schnorr Threshold) signature scheme[1] includes multi-party protocols for Distributed Key Generation (DKG) and signing a message. DKG allows n parties to jointly generate a key, which they split into shares such that any set of $t \leq n$ parties can jointly generate a signature on a message. Coinbase's FROST implementation is part of a larger library, `kryptology`, which provides an API with functions for each round of DKG and signing.

Source code for the `kryptology` library was provided via a SendSafely workspace set up by Coinbase. Only the FROST implementation and its dependencies in other modules of the library were in scope.

The review comprised 15 person-days over 3 calendar weeks and was delivered by one NCC Group consultant who was assisted by a shadow resource for all three weeks.

Scope

NCC Group's evaluation included:

- pkg/core/curves: implementations of several elliptic curve groups that may be used for DKG or FROST signing.
- pkg/sharing: implementations of Shamir secret sharing and Feldman Verifiable Secret Sharing (VSS) used in DKG and FROST signing.
- pkg/dkg/frost: implementation of FROST's two-round DKG protocol. The DKG API provides the functions `NewDkgParticipant()`, `Round1()`, and `Round2()`.
- pkg/ted25519: implementation of a three-round variant of FROST's signing protocol without a Signature Aggregator (SA). The FROST signing API provides the functions `NewSigner()`, `SignRound1()`, and `SignRound2()`.

Limitations

NCC Group's evaluation of FROST covered only the code in the `kryptology` library, not its use. The protocol could be rendered insecure due to a number of factors outside of the scope of the library, such as:

- authentication of parties in DKG and signing;
- encryption, authentication, and reliable delivery of messages sent during DKG and signing;
- serialization and deserialization of messages sent during DKG and signing;
- ejection of a party from the set of potential signers after detection of their misbehavior;
- secure low-level elliptic curve group primitives provided by dependencies outside of `kryptology`.

Key Findings

The assessment uncovered a set of common application flaws. The most notable findings were:

- Error codes returned by system-level randomness APIs are not checked, which may lead to **weak secret key shares and nonces** (finding NCC-E002578-002 on page 5).
- Values received from other parties are not sufficiently validated, which may cause the Go process to **panic and terminate** during DKG (finding NCC-E002578-008 on page 10) or may allow a misbehaving participant to evade detection during signing and cause all parties to **output an invalid signature** (finding NCC-E002578-010 on page 13).

Strategic Recommendations

- **Augment library documentation** to give users guidance on how to securely use the API.
  - In particular, clearly outline what are the responsibilities of the caller (e.g. checking that the message is indeed one they want to sign, authenticating participants) and what are the responsibilities of the library (e.g. verifying proofs).
- **Expand the suite of test functions** to ensure that all code paths and edge cases receive coverage, especially validation of scalars, curve points, and lists received from other parties.
- **Keep up to date** with the progression of the FROST IRTF Internet Draft,[2] which will be updated at least once every 6 months while it is being considered for publication as an RFC (Request for Comments). The next update should be in February 2022.
  - Consider also following the author's working copy of the Internet Draft on their GitHub page,[3] which may receive more frequent updates.
  - Consider following discussions about FROST on the mailing list[4] of the CFRG (Crypto Forum Research Group).

[1] https://eprint.iacr.org/2020/852
[2] https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/
[3] https://github.com/chelseakomlo/frost-spec
[4] https://mailarchive.ietf.org/arch/browse/cfrg/?q=frost

Dashboard

Finding Breakdown

Critical issues	0	
High issues	0	
Medium issues	4	🟧🟧🟧🟧
Low issues	5	🟨🟨🟨🟨🟨
Informational issues	2	⬜⬜
Total issues	**11**	

Category Breakdown

Configuration	2	🟨🟨
Cryptography	7	🟧🟧🟨🟨🟨⬜⬜
Data Validation	1	🟧
Denial of Service	1	🟧

Component Breakdown

DKG	1	🟧
DKG, Signing	2	🟧🟨
FROST	1	⬜
General	2	🟨🟨
Signing	1	🟨
curves	3	🟧🟧⬜
curves, DKG, Signing	1	🟨

Key

🟥 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix B on page 42.

Title	Status	ID	Risk
Calls to `Read()` in `Scalar.Random()` Functions May Silently Fail	Reported	002	Medium
Curve P256 Validates Points on the Wrong Curve	Reported	004	Medium
Insufficient Validation May Lead to Panics	Updated	008	Medium
Insufficient Validation May Lead to Undetected Misbehaving Parties and Invalid Signatures	New	010	Medium
Outdated Module Versions in go.sum, Outdated Minimum Go Version in go.mod	Reported	001	Low
Generation of Random Scalars and Schnorr Challenges Uses Unnecessary Hash-to-Field Operation	Updated	003	Low
Code is Not Constant-Time	Reported	005	Low
Implementation Does Not Identify Misbehaving Participants	Reported	006	Low
Minor Deviations from FROST Specification	Updated	009	Low
FROST Implementation Does Not Follow Most Recent Specification	New	007	Informational
`ScalarP256`'s `SetBytesWide()` Method May Return Incorrect or Non-Canonical Scalars	New	011	Informational

Finding	**Calls to `Read()` in `Scalar.Random()` Functions May Silently Fail**
Risk	**Medium** Impact: High, Exploitability: Undetermined
Identifier	NCC-E002578-002
Status	Reported
Category	Data Validation
Component	curves
Location	• kryptology/pkg/core/curves/bls12381_curve.go (line 43) • kryptology/pkg/core/curves/ed25519_curve.go (line 33) • kryptology/pkg/core/curves/k256_curve.go (line 28) • kryptology/pkg/core/curves/p256_curve.go (line 28) • kryptology/pkg/core/curves/pallas_curve.go (line 213)
Impact	If access to `crypto/rand` ever fails for any reason, the library will use weak secret key shares, nonces, and other security-relevant values.

Description Each of the various curves in kryptology/pkg/core/curves implements a `Scalar` interface (specified in kryptology/pkg/core/curves/curve.go) including a `Random()` function that takes as input a `reader` (of type `io.Reader`) from which it reads some random bytes. This `Random()` function is used in various places in the code in scope for this review:

- when a blinding factor is sampled in `Pedersen.Split()` (pkg/sharing/pedersen.go, line 93),
- when a polynomial's coefficients are sampled in `Polynomial.Init()` (pkg/sharing/polynomial.go, line 22),
- when an initial share of the long-term secret key is sampled in round 1 of DKG (pkg/dkg/frost/dkg_round1.go, line 44),
- when a share of the random nonce is sampled in round 1 of DKG (pkg/dkg/frost/dkg_round1.go, line 66), and
- when the components `di` and `ei` of the random nonce are sampled in round 1 of signing (pkg/ted25519/frost/round1.go, lines 32 and 34).

The following excerpt from k256_curve.go is a representative example of an implementation of one of these `Random()` functions. (The functions for the other curves listed in the "Location" field above are very similar.) The function first attempts to read 64 bytes from `reader` and then hashes them to a scalar using a curve-specific hash function.

```
28 func (s *ScalarK256) Random(reader io.Reader) Scalar {
29   if reader == nil {
30     return nil
31   }
32   var seed [64]byte
33   _, _ = reader.Read(seed[:])
34   return s.Hash(seed[:])
35 }
```

The `io.Reader` interface stipulates that the `Read()` function returns two values: the number of bytes read (`n`) and any error that was encountered (`err`). However, the return values of `reader.Read()` (highlighted on line 33) are ignored; the function does not check whether `Read()` was able to read all 64 bytes, nor whether there were any other errors. Thus, if an error occurs, `Random()` will silently proceed to hash a slice of all zero bytes, which will not return a uniformly random scalar.

Currently, the only `io.Reader` that appears to be used in `kryptology` is `crypto/rand`, a cryptographically secure random number generator that uses the randomness APIs of the system it is running on. Its documentation[5] states that "[o]n return, n == len(b) if and only if err == nil." Therefore, it would be sufficient to check only one of the two return values.

Recommendation

- Whenever the `Read()` function of an object implementing the `io.Reader` interface is called, check its return values and propagate the error as necessary in the calling functions.
- Specifically, in the `Random()` implementation of all of the curves listed in this finding, check that the call to `Read()` returned the expected number of bytes and return an error if not.

Coinbase Category Security / Implementation issues

[5] https://pkg.go.dev/crypto/rand

Finding	**Curve P256 Validates Points on the Wrong Curve**
Risk	**Medium** Impact: High, Exploitability: Undetermined
Identifier	NCC-E002578-004
Status	Reported
Category	Cryptography
Component	curves
Location	kryptology/pkg/core/curves/p256_curve.go (line 346)
Impact	Curve P256 validates points are on K256 (secp256k1) rather than P256 (secp256r1). In general, this could allow invalid point attacks leading to the recovery of secrets. In FROST, this could allow misbehaving parties to go undetected during the key generation and signing protocols, leading to incorrect signatures.

Description An *invalid point attack* is where one party is able to supply a point $P(x, y)$ that does not satisfy the chosen curve equation $E : y^2 = x^3 + Ax + B \pmod{p}$, and another party uses this point as part of a computation with a secret value. The result of this computation may leak information about the second user's secret value.

Proper implementation requires that before performing any operations on a user-supplied point, the point is checked to be on the curve (and abort otherwise). When this check is omitted or improperly implemented, an attacker may be able to force another party to, in effect, perform elliptic curve operations over a new curve $E' : y^2 = x^3 + Ax + B' \pmod{p}$ that may offer significantly reduced security.

Due to the Pohlig-Hellman algorithm,[6] the hardness of the discrete logarithm problem for an elliptic curve group is bounded by (the square-root of) the largest prime divisor of the group's order. For curves such as NIST-P256 (secp256r1) and the Bitcoin curve K256 (secp256k1), the groups have a prime order and offer 128-bit security. When an invalid point attack can be performed, the modification of the curve equation allows the attacker to select a new curve, where the group order may have many small prime factors. Therefore, an invalid point attack can heavily reduce the security of the discrete logarithm problem on the curve and hence the security of any protocol that relies on this hardness (such as FROST). An example of this attack was recently carried out against the Bluetooth pairing protocol in 2018.[7]

The vulnerability appears within `kryptology` in the file pkg/core/curves/p256_curve.go; it is introduced by the point validation function for P256:

```
346 func (p *PointP256) IsOnCurve() bool {
347         return btcec.S256().IsOnCurve(p.x, p.y)
348 }
```

Rather than ensuring the point is on secp256r1 as intended, the point is checked to be on the secp256k1 curve instead. (This function may have been copied and pasted from k256_curve.go.) With high probability, this bug will result in the function returning false and legitimate, valid curve points will be incorrectly identified as invalid.

The `IsOnCurve()` function does not appear to be directly used within the FROST protocol's implementation. If it were, it may lead to misbehaving parties evading detection during key

[6] https://en.wikipedia.org/wiki/Pohlig%E2%80%93Hellman_algorithm
[7] https://www.cs.technion.ac.il/~biham/BT/bt-fixed-coordinate-invalid-curve-attack.pdf

generation (which involves verifying other parties' signatures for proof of knowledge of their shares a_{i0} of the long-term secret key) or during signing (which involves jointly computing the group commitment R as a function of the participants' nonce commitment parts D_ℓ and E_ℓ).

However, the `IsOnCurve()` function is used in other (out of scope) areas of `kryptology`, for example, in pkg/tecdsa/gg20/participant/dkg_round3.go:

```go
// This is a sanity check to make sure nothing went wrong when
// computing the public key
if !dp.Curve.IsOnCurve(y.X, y.Y) || y.IsIdentity() {
	return nil, fmt.Errorf("invalid public key")
}
```

For an attacker, this could allow an invalid point attack. By submitting a point which is valid on the Bitcoin curve, the NIST-P256 curve equation and corresponding arithmetic are modified to a new curve which is neither secp256r1 nor secp256k1. Using a small sample of points from secp256k1 and performing an invalid point attack, it was found that the resulting curve equation offered as little as 30-bit security due to the smallness of the prime factors of the point's group order.

The P256 function containing the bug is not covered by any tests. Additionally, when a point is set on the curve, the `Set()` function calls directly `elliptic.P256().IsOnCurve(x, y)` rather than the wrapper function defined on line 346 of p256_curve.go.

```go
func (p *PointP256) Set(x, y *big.Int) (Point, error) {
	// check is identity or on curve
	xx := subtle.ConstantTimeCompare(x.Bytes(), []byte{})
	yy := subtle.ConstantTimeCompare(y.Bytes(), []byte{})
	// Checks are constant time
	onCurve := elliptic.P256().IsOnCurve(x, y)
	if !onCurve && (xx&yy) != 1 {
		return nil, fmt.Errorf("invalid coordinates")
	}
	x = new(big.Int).Set(x)
	y = new(big.Int).Set(y)
	return &PointP256{x, y}, nil
}
```

As a result, the vulnerability is introduced when explicitly calling the `isOnCurve(Point)` function, rather than the `Set(x, y)` function as used in pkg/core/curves/p256_curve_test.go.

More generally, any user-submitted points must be checked to be valid before processing. For compressed points, this is implicitly handled since only valid x-coordinates are lifted, and invalid x-coordinates are mapped to the identity point. For uncompressed points, the code for the secp256r1 and secp256k1 curves implement a bytes-to-point function:

```go
func (p *PointP256) FromAffineUncompressed(bytes []byte) (Point, error) {
	if len(bytes) != 65 {
		return nil, fmt.Errorf("invalid byte sequence")
	}
	if bytes[0] != 4 {
		return nil, fmt.Errorf("invalid sign byte")
	}
	x := new(big.Int).SetBytes(bytes[1:33])
	y := new(big.Int).SetBytes(bytes[33:])
	return &PointP256{x, y}, nil
}
```

Above is the example from pkg/core/curves/p256_curve.go, but the function is essentially identical for pkg/core/curves/k256_curve.go. Here, the point is returned without any validation, which could allow an invalid curve attack if at any point this function is executed on user input. For the BLS12-381 curves, Pallas/Vesta curves and Ed25519 curve, the point is created using packages outside of the scope of our audit and so were unable to be checked.

An attack against secp256r1 and secp256k1 can be mitigated by asserting `IsOnCurve()` with the supplied coordinates before returning the point and returning an error if the point is not on the curve.

The fix for the vulnerability appearing on line 347 requires only modifying the above function using the (intended) `elliptic.P256().IsOnCurve(P.x, P.y)`, which will ensure all supplied points are on the correct curve, maintaining the 128-bit security promised by secp256r1. Note that this function is not constant-time and that Go's `crypto/elliptic` package offers this check in constant time only for P224 and P521. (See `elliptic` documentation.[8])

Recommendation

- Repair the `PointP256` function `IsOnCurve()` (pkg/core/curves/p256_curve.go) such that it checks points are on the correct curve.

```go
func (p *PointP256) IsOnCurve() bool {
        return elliptic.P256().IsOnCurve(p.x, p.y)
}
```

- Include `IsOnCurve()` in the tests and allow for both positive and negative test cases for the function.
- Ensure that the wrapper functions are consistent. The `Set()` wrapper for a curve should use the defined `IsOnCurve()` function rather than calling to the curve's defaults, which would have allowed this bug to be identified in testing.
- Ensure that all point arithmetic for an elliptic curve is performed on coordinates (x, y) which lie on the curve.

Coinbase Category Cryptographic / Mathematical

[8] https://cs.opensource.google/go/go/+/refs/tags/go1.17.3:src/crypto/elliptic/elliptic.go;l=85

Finding	**Insufficient Validation May Lead to Panics**
Risk	**Medium** Impact: High, Exploitability: Undetermined
Identifier	NCC-E002578-008
Status	Updated
Category	Denial of Service
Component	DKG
Location	• kryptology/pkg/dkg/frost/dkg_round1.go • kryptology/pkg/dkg/frost/dkg_round2.go
Impact	• Misbehaving parties can trigger other parties to panic during DKG, resulting in process termination. • An incorrect local configuration can trigger a panic.

Description

The FROST Distributed Key Generation (DKG) protocol is implemented in two main methods: `DkgParticipant.Round1()` (pkg/dkg/frost/dkg_round1.go) and `DkgParticipant.Round2()` (pkg/dkg/frost/dkg_round2.go). This finding is about how insufficient validation of the local `DkgParticipant` struct (in round 1) and received messages (in round 2) may lead to panics or other errors.

Local `DkgParticipant` struct

`DkgParticipant` is a struct (defined in pkg/dkg/frost/participant.go) with the following members:

```go
type DkgParticipant struct {
        round                    int
        Curve                    *curves.Curve
        otherParticipantShares   map[uint32]*dkgParticipantData
        Id                       uint32
        SkShare                  curves.Scalar
        VerificationKey          curves.Point
        VkShare                  curves.Point
        feldman                  *sharing.Feldman
        verifiers                *sharing.FeldmanVerifier
        secretShares             []*sharing.ShamirShare
        ctx                      byte
}
```

The library's `frost` package provides a convenience function `NewDkgParticipant()` (pkg/dkg/frost/participant.go) that creates a `DkgParticipant`, however, not all users of this package may use this function, and users could modify the values of any of the (exported) members of the `DkgParticipant` struct. Therefore, all `DkgParticipant` methods should verify that the struct is valid and consistent with itself.

First, there is some redundancy in the information in this struct. For example, the particular elliptic curve group is captured in both the `Curve` member and in `feldman`, which is a struct that also has a `curve` member. In `DkgParticipant.Round1()`, if the type of `DkgParticipant.Curve.Scalar` is different than the type of `DkgParticipant.feldman.curve.Scalar`, then a panic will be triggered by calling `dp.feldman.Split()` (line 56). This would be due to the scalar type of the `s` argument (which comes from `dp.Curve.Scalar`) being different than the type of `dp.feldman.curve.Scalar`. The `Split()` method calls `Shamir.getPolyAndSha` res(), which calls poly.Evaluate(), where the panic occurs (pkg/sharing/polynomial.go, line 31, copied below for reference). In Evaluate(), the scalar type of out is ultimately determined by the s argument, while the type of its argument x is ultimately determined by dp.feldman.curve.Scalar. When the types of out and x do not match, Mul() returns nil, which Evaluate() proceeds to try to dereference when it calls Add() on it, resulting in the panic.

```
func (p Polynomial) Evaluate(x curves.Scalar) curves.Scalar {
  degree := len(p.Coefficients) - 1
  out := p.Coefficients[degree].Clone()
  for i := degree - 1; i >= 0; i-- {
    out = out.Mul(x).Add(p.Coefficients[i])
  }
  return out
}
```

Another example is the way the DkgParticipant struct captures the total number of participants: dp.feldman has a limit member, and dp.otherParticipantShares is a map that should contain dp.feldman.limit-1 elements. One of the outputs of round 1 is p2pSend , a slice containing all of the secret shares (to be sent individually to each other party). Its length is determined by the length of dp.otherParticipantShares, and its contents are taken from shares, which is a slice whose length depends on dp.feldman.limit. If the dp.otherParticipantShares map does not contain the correct number of keys, or those keys are not in the correct range (1 to dp.feldman.limit), then either not enough shares will be output in p2pSend, or there will be a panic as a result of an index out of bounds error (line 99).

Lastly, note that the DkgParticipant struct's member Id has type uint32, but when forming message hashes for the Schnorr PoK, this value is converted to a byte (line 74), which silently truncates the top 24 bits. This may lead to unexpected results.

Received Messages

In round 2, each party processes the Round1Bcast message it received from each other party:

```
type Round1Bcast struct {
        verifiers *sharing.FeldmanVerifier
        wi, ci    curves.Scalar
}
```

The types of these values should be validated for consistency with each other and with the receiving party's DkgParticipant struct, dp, before operating on them.

The wi value should be checked to be a scalar of the same type as dp.Curve.Scalar, otherwise prod1 will be nil (line 54) and a panic will arise as a result of the attempt to dereference it (line 57).

The verifiers member is a FeldmanVerifier struct which contains a Commitments array of points. During validation of another party's message, a scalar multiplication operation is applied to the first point of Commitments and the negative of the scalar ci (line 56). If they are not on the same curve, prod2 will be nil, and thus so will prod (line 57), and a panic will arise as a result of the attempt to dereference prod (line 66).

Note that line 70 takes care of checking that dp.Curve.Scalar has the same type as ci: the Cmp() function will return -2 if not.

Recommendation

- Validate externally supplied data at every layer for defense in depth. This includes validating the consistency of a `DkgParticipant`'s member values before they are used in any method and validating the types of values received from other parties before they are used in computations.
- Use data structures that do not contain duplicate or redundant information whenever possible to simplify checking of their consistency.
- Consider using the `recover()` function[9] to gracefully handle panics.

Coinbase Category Security / Implementation issues

[9] https://golang.org/ref/spec#Handling_panics

Finding	**Insufficient Validation May Lead to Undetected Misbehaving Parties and Invalid Signatures**
Risk	**Medium** Impact: Medium, Exploitability: Undetermined
Identifier	NCC-E002578-010
Status	New
Category	Cryptography
Component	DKG, Signing
Location	• kryptology/pkg/dkg/frost/dkg_round2.go • kryptology/pkg/ted25519/frost/round2.go • kryptology/pkg/ted25519/frost/round3.go
Impact	Insufficient validation of received values (scalars, curve points, lists) may allow misbehaving parties to go undetected and may lead to incorrect protocol output.

Description

All scalars and points received from other parties should be validated, even if they have the correct type (see finding NCC-E002578-008 on page 10). In particular, scalars should be in the correct range relative to the group order (and non-zero when necessary), points should be on the correct curve (and in the correct group when applicable), and lists should have the correct lengths. Many of these checks do not occur in the code during DKG and signing.

- **DKG, round 2:** Each party must validate the `Round1Bcast` struct and the `ShamirShare` it receives from each other party. `Round1Bcast` contains a `FeldmanVerifier` struct that contains an array, `Commitments`, of points.
  - Each point in this array should be checked to be on the curve the recipient expects, i.e., the receiving `DkgParticipant`'s `Curve` (`dp.Curve`), and in the correct subgroup if the curve's order is not prime.
  - The length of the array should be checked to be what the recipient expects, i.e., `dp.feldman.threshold`.

  `Round1Bcast` also contains two scalars, `wi` and `ci`.
  - `wi` should be checked to be within the range `0` to `q`-1 (inclusive) where `q` is the order of the group of `dp.Curve`.
  - `ci`, which should be a hash value, should be checked to be within the range 1 to `q`-1 (inclusive) where `q` is the order of the group of `dp.Curve`. Note that `ci` should not be `0` according to the specification (draft-komlo-frost-00):

    ```
    >   Let G be a group with prime order q and generator g, and let H be a
    >   cryptographic hash function mapping to Z_(q)^(*).  A Schnorr
    >   signature is generated over a message m by the following steps:
    ```

  If these checks on the received `Round1Bcast` struct are done, then the checks performed by the `FeldmanVerifier` method `Verify()` (from pkg/sharing/feldman.go) should be sufficient to validate the `ShamirShare`: this method already checks that the `ShamirShare`'s ID is not `0` and that the value is in the range 1 to `q`-1 (inclusive) where `q` is the order of the curve group associated with the first point in the `Commitments` array.
  - Finally, the number of received `Round1Bcast` structs and `ShamirShare`s should be verified to be `dp.feldman.limit`.
- **Signing, round 2:** Each party must validate the `Round1Bcast` struct it receives from each other party. `Round1Bcast` contains two curve points, `Di` and `Ei`, for the commitment shares.
  - The points `Di` and `Ei` should be checked to be on the curve the recipient expects, i.e., the receiving `Signer`'s `curve` (`signer.curve`), in the correct subgroup if the curve's order

is not prime, and not the group's identity element (the point at infinity). Note that a comment on line 53 of pkg/ted25519/frost/round2.go says `Check Dj, Ej on the curve`, but this does not appear to happen.

The number of received `Round1Bcast` structs is already checked to be what the recipient expects (`signer.threshold`).

- **Signing, round 3:** Each party must validate the `Round2Bcast` struct it receives from each other party. `Round2Bcast` contains a scalar, `zi`, and a curve point, `vki`.
  - The scalar `zi` should be checked to be within the range `0` to `q`-1 (inclusive) where `q` is the order of the group of `signer.Curve`.
  - The point `vki` should be checked to equal the other party's public key share (g^{s_i} or `Y_(i)`) as computed during DKG. Without this check, verification of `zi` as a valid signature share (step 7.b, line `90` of pkg/ted25519/frost/round3.go) is easily bypassed: a misbehaving party could simply send the point at infinity for `vki` and `di+ei*ri` for `zi`: these values would satisfy the equation `zi*G = Ri + c*Li*vki`, since the right-hand side would simplify to `Ri = D_i + r_i*E_i = di*G + r_i*e_i*G`. However, including this signature share in the final signature, which is the sum of the `zi`s, would render it invalid since there would be no contribution from the user's share of the private key.

    The point `vki` should be verified by computing it from the `verifiers` broadcast during round 1 of DKG. This is described in the specification (draft-komlo-frost-00) as follows:

    ```
    >   4.  Each P_(i) calculates their public verification share Y_(i) =
    >      g^{s_(i)}, and the group's public key Y = PROD(A_(j0), j=1...n).
    >      Any participant can compute the public verification share of any
    >      other participant by calculating Y_(i) = PROD( (A_(jk))^((i^k mod
    >      q)), j=1...n, k=0...t-1)
    ```

    Note that since `vki` must always be computed locally by each party in order to verify each other party's signature share, it could be removed from `Round2Bcast` during signing. Similarly, it does not need to be sent[10] after round 2 of DKG. Instead, each user's public key share could be computed after DKG and passed to the new `Signer` when it is created.

  The number of received `Round2Bcast` structs is already checked to equal what the recipient expects (`signer.threshold`) on line `60` of pkg/ted25519/frost/round3.go.

Recommendation

- Explicitly verify that all received scalars are in the expected range, that all curve points are in the expected group, and that all lists have the expected length.
- Compute each party's public key share (`vki`) from the `verifiers` broadcast by each party in round 1 of DKG; remove it from the broadcast values in round 2 of signing.

Coinbase Category Cryptographic / Mathematical

[10] The `Round2()` method of `DkgParticipant` outputs a `Round2Bcast` struct containing the (joint) verification key and the user's verification key share. It is unclear how these values are used, as no code appears to process them.

Finding	**Outdated Module Versions in go.sum, Outdated Minimum Go Version in go.mod**
Risk	**Low** Impact: Low, Exploitability: Low
Identifier	NCC-E002578-001
Status	Reported
Category	Configuration
Location	• kryptology/go.mod • kryptology/go.sum
Impact	Allowing outdated versions of Go or outdated dependencies may result in known vulnerabilities being exposed in `kryptology` packages.

Description A library can be no more secure than its individual components and the toolchain used to build it. Keeping `kryptology`'s dependency modules and Go itself up to date can help keep the library secure by ensuring that all known issues in those components are fixed.

Dependency Modules

The module's go.sum file lists dependency modules and their checksums. One of these dependencies is deprecated and several are out of date:

- The indirect dependency `golang/protobuf` (whose dependency chain is `btcsuite/btcd` > `onsi/gomega` > `golang/protobuf`) is deprecated.
- The direct dependencies `x/crypto`, `x/tools`, and `btcsuite/btcd` are out of date.
- Additionally, several indirect dependencies are out of date.

The output of `go list -u -m all` shows which modules can be updated by displaying the latest available version in square brackets when it differs from the currently used version. The dependency modules that can be updated are copied below:

- `git.sr.ht/~sircmpwn/getopt v0.0.0-20191230200459-23622cc906b3 [v1.0.0]`
- `github.com/btcsuite/btcd v0.21.0-beta.0.20201114000516-e9c7a5ac6401 [v0.22.0-beta]`
- `github.com/creack/pty v1.1.9 [v1.1.17]`
- `github.com/decred/dcrd/lru v1.0.0 [v1.1.1]`
- `github.com/fsnotify/fsnotify v1.4.7 [v1.5.1]`
- `github.com/gogo/protobuf v1.3.1 [v1.3.2]`
- `github.com/golang/protobuf v1.2.0 [v1.5.2] (deprecated)`
- `github.com/jessevdk/go-flags v1.4.0 [v1.5.0]`
- `github.com/kisielk/errcheck v1.2.0 [v1.6.0]`
- `github.com/kkdai/bstream v0.0.0-20161212061736-f391b8402d23 [v1.0.0]`
- `github.com/kr/pretty v0.2.1 [v0.3.0]`
- `github.com/kr/pty v1.1.1 [v1.1.8]`
- `github.com/mimoo/StrobeGo v0.0.0-20181016162300-f8f6d4d2b643 [v0.0.0-20210601165009-122bf33a46e0]`
- `github.com/onsi/ginkgo v1.7.0 [v1.16.5]`
- `github.com/onsi/gomega v1.4.3 [v1.16.0]`
- `github.com/stretchr/objx v0.1.0 [v0.3.0]`
- `github.com/yuin/goldmark v1.3.5 [v1.4.2]`
- `golang.org/x/crypto v0.0.0-20210711020723-a769d52b0f97 [v0.0.0-20210921155107-089bfa567519]`
- `golang.org/x/mod v0.4.2 [v0.5.1]`

- `golang.org/x/net v0.0.0-20210405180319-a5a99cb37ef4 [v0.0.0-2021110119342 0-4a448f8816b3]`
- `golang.org/x/sys v0.0.0-20210630005230-0f9fa26af87c [v0.0.0-2021110318473 4-ae416a5f93c7]`
- `golang.org/x/term v0.0.0-20201126162022-7de9c90e9dd1 [v0.0.0-202109272227 41-03fcf44c2211]`
- `golang.org/x/text v0.3.3 [v0.3.7]`
- `golang.org/x/tools v0.1.5 [v0.1.7]`
- `gopkg.in/yaml.v2 v2.2.1 [v2.4.0]`

These modules can be upgraded (and the go.mod and go.sum files updated) with `go get -d -u all`.

Go and Core Library

The `kryptology` module's go.mod file specifies a minimum Go version of 1.15, while the current latest major release[11] is 1.17 from August 2021. Since version 1.15, there have been several minor updates to Go's runtime, compiler, linker, and modules from the core library that are used in `kryptology`, such as `crypto/hmac`, `crypto/elliptic`, and `crypto/rand`.

Some specific recently fixed issues are the following:

- `crypto/elliptic`: incorrect operations on the P-224 curve,[12] fixed in 1.15.7.
- `crypto/hmac`: undefined behavior leading to invalid outputs,[13] fixed in 1.16.

Raising the minimum Go version in go.mod will ensure that the module is never built for an older major version of Go that includes known vulnerabilities such as these.

Recommendation

- Upgrade all direct and indirect dependencies with `go get -d -u all`.
- Regularly check for updated, deprecated, and retracted dependencies by examining the output of `go list -m -u all` and then upgrading them with `go get`. Also run `go mod tidy` to ensure that go.mod and go.sum are not tracking any unused dependencies.
- Update the minimum specified version of Go in go.mod to the latest version, 1.17, and continue keeping the `kryptology` module's minimum version up to date with the latest major releases of Go. These are typically released every 6 months, so the next release is expected around February 2022.
- Monitor the golang-announce mailing list[14] for security-related updates to minor versions of Go and update when necessary.

Coinbase Category Security / Implementation issues

[11] Go version history: https://golang.org/project#go1
[12] https://groups.google.com/g/golang-announce/c/mperVMGa98w
[13] https://golang.org/doc/go1.16#crypto/hmac
[14] golang-announce mailing list: https://groups.google.com/g/golang-announce

Finding **Generation of Random Scalars and Schnorr Challenges Uses Unnecessary Hash-to-Field Operation**

Risk **Low** Impact: High, Exploitability: Undetermined

Identifier NCC-E002578-003

Status Updated

Category Cryptography

Component curves, DKG, Signing

Location Random scalar generation:

- kryptology/pkg/core/curves/bls12381_curve.go (line 49)
- kryptology/pkg/core/curves/ed25519_curve.go (line 39)
- kryptology/pkg/core/curves/k256_curve.go (line 34)
- kryptology/pkg/core/curves/p256_curve.go (line 34)
- kryptology/pkg/core/curves/pallas_curve.go (line 219)

Schnorr challenges:

- kryptology/pkg/dkg/frost/dkg_round1.go (line 82)
- kryptology/pkg/dkg/frost/dkg_round1.go (line 68)
- kryptology/pkg/ted25519/round2.go (line 71)

Impact Unnecessary cryptographic operations reduce readability, make errors harder to identify, and decrease performance.

Description Each of the various curves in kryptology/pkg/core/curves implements a `Scalar` interface (specified in kryptology/pkg/core/curves/curve.go) including a `Random()` function that takes as input a `reader` (implementing the `io.Reader` interface) from which it reads some random bytes. This `Random()` function is used in various places in the code in scope for this review, listed in finding NCC-E002578-002 on page 5. The implementations of `Random()` follow the same basic steps: a number of bytes are read from `reader`, the `Hash()` function is called on them, and the return value is the output of `Hash()`.

The `Hash()` function is also called in FROST DKG when each party computes the Schnorr Proof of Knowledge (PoK) of its secret key share (round 1) and verifies each other party's proof (round 2). In signing, it is called when computing the binding values (which are called ρ_i in the FROST eprint paper, `r_(p)` in draft-komlo-frost-00, `rj` in `kryptology`) in round 2. (There is an additional call to hash function when computing the Schnorr challenge in round 3, but this is not implemented using the scalar `Hash()` function.)

The curves' implementations of `Hash()` also follow the same basic steps: the input bytes (and a curve-specific domain separation string) are hashed with `expandMsgXmd()`, a `big.Int` is created from the output, reduced modulo the group order, and used to instantiate a new `Scalar` object, which is the output of `Random()`.

Below is a representative example from k256_curve.go:

```go
func (s *ScalarK256) Random(reader io.Reader) Scalar {
  if reader == nil {
    return nil
  }
  var seed [64]byte
  _, _ = reader.Read(seed[:])
  return s.Hash(seed[:])
}

func (s *ScalarK256) Hash(bytes []byte) Scalar {
        xmd, err := expandMsgXmd(sha256.New(), bytes,
         ➜  []byte("secp256k1_XMD:SHA-256_SSWU_RO_"), 48)
        if err != nil {
                return nil
        }
        v := new(big.Int).SetBytes(xmd)
        return &ScalarK256{
                value: v.Mod(v, btcec.S256().N),
        }
}
```

In Hash(), the name expandMsgXmd appears to refer to the expand_message_xmd function from the IETF draft "hash-to-curve" specification,[15] in Section 5: Hashing to a finite field. The expand_message_xmd function takes an arbitrary-length byte string and hashes it to a uniformly random byte string whose length is big enough (relative to the field size) to effectively eliminate any bias from reducing it modulo the field size

Hashing to a finite field is an appropriate operation in contexts where there is a need for a *deterministic* mapping from arbitrary-length inputs to *uniformly random* scalars. In the context of the Random() function, however, it is unnecessary: there is no need for determinism since the input to Hash() is freshly generated every time, and the input itself is already uniformly random, since it was generated with the cryptographically secure pseudorandom number generator crypto/rand.

Additionally, the added complexity may make it harder to identify errors. For example, as mentioned in finding NCC-E002578-002 on page 5, the return values of reader.Read() are not checked, so if this function fails, the Random() function will proceed to hash a seed of 64 zero bytes. Ultimately, the output of Random() will *look* random, but it will be a fixed string every time the function is called. The tests in k256_curve_test.go, copied below, will not catch such an error: they check only whether the output of Random() is a slice of zero bytes.

```
func TestScalarK256Random(t *testing.T) {
        k256 := K256()
        sc := k256.Scalar.Random(testRng())
        s, ok := sc.(*ScalarK256)
        assert.True(t, ok)
        expected, _ := new(big.Int).SetString(
         ➜  "2f71aaec5e14d747c72e46cdcaffffe6f542f38b3f0925469ceb24ac1c65885d",
         ➜  16)
        assert.Equal(t, s.value, expected)
        // Try 10 random values
        for i := 0; i < 10; i++ {
                sc := k256.Scalar.Random(crand.Reader)
                _, ok := sc.(*ScalarK256)
                assert.True(t, ok)
                assert.True(t, !sc.IsZero())
        }
}
```

[15] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-12

The curves' implementation of `Random()` can be simplified to generate a random scalar as follows: continue using `crypto/rand`'s `Read()` function to read a number of random bytes (say `n`), then reduce it modulo the group size (say, `q`). Here, care must be taken to ensure two properties:

- First, no biases should be introduced in the modular reduction: each of the `q` possible values should be output uniformly randomly. Avoiding this bias requires generating a greater number `n` of random bytes. Specifically, to obtain a distribution modulo `q` that is indistinguishable from uniform, the number of random bytes `n` should be at least ($\log_2$(`q`) + 128)/8 bytes. For group sizes `q` that are roughly 256 bits, this results in `n` $\geq$ 48 bytes. The implementations of `Random()` for the curves mentioned in this finding already generate 64 random bytes, which is sufficient.
- Second, if constant-time code is desired, the time taken to do the modular reduction step itself should depend only on the *lengths* of the operands (the random byte string and the modulus), not their values. Constant-time functions on arbitrarily big integers are typically constructed using as building blocks the low-level arithmetic operations available on a specific platform's architecture, e.g. bitshifts, addition, and multiplication of 32- or 64-bit words. Note, however, that these building blocks themselves may not be consistently constant-time across different CPUs,[16] and that, in general, memory access patterns must also be taken into account when writing constant-time code. Unfortunately, Go's `math/big` module does not provide any constant-time functions for modular reduction. An additional issue is that simply setting the `big.Int` instance value from the random bytes will imply allocating and accessing an internal array of words whose length depends on the mathematical value itself; thus, if the highest-order random bytes are zero, then that array will be shorter, and the memory access pattern will be different.[17] Coinbase could consider evaluating available third-party libraries, such as `saferith`,[18] to see if they meet their requirements.

Note that if constant-time code is *not* required, it would also possible to use `crypto/rand`'s `Int()` function[19] with a parameter of `max` equal to the group order to generate a uniform number in {`0`,...,`q`-`1`}. This function uses rejection sampling to ensure that it generates a uniformly random value. However, it is not constant-time, not only because setting the value implies a memory access pattern that depends on the mathematical value of the integer (and not only the size of the source byte sequence), but also because the `Cmp()` function uses data-dependent branching and notably will return much faster when comparing values with different array lengths.[20]

In the context of FROST DKG and signing, the scalar `Hash()` function can also simply be replaced by a call to cryptographic hash function that outputs sufficiently many bits (e.g., at least 384 if hashing modulo a 256-bit value) and a modular reduction. (The inputs to the hash functions used in DKG and signing are all public, so there is no need for these operations to be constant-time.)

Finally, note that FROST requires different hash functions ($H()$, $H_1()$, $H_2()$) because the security proofs model them as independent random oracles. In practice, making them "distinct" can be accomplished by using a standard cryptographic hash function, e.g. SHA-512, and prepending context strings (e.g. "FROST-SHA512-H").

Recommendation

[16] https://www.bearssl.org/ctmul.html#per-cpu-information
[17] https://cs.opensource.google/go/go/+/refs/tags/go1.17.3:src/math/big/nat.go;l=50
[18] https://github.com/cronokirby/saferith, https://eprint.iacr.org/2021/1121
[19] https://pkg.go.dev/crypto/rand#Int
[20] https://cs.opensource.google/go/go/+/refs/tags/go1.17.3:src/math/big/nat.go;l=153

- Remove the calls to `Hash()` in the `Random()` functions listed in the "Location" field of this finding. Generate random scalars by reducing the output of `crypto/rand`'s `Reader` modulo the group size, taking care to ensure that the modular reduction is done in constant time with respect to the lengths of the random output from `Reader` and the modulus.
- Remove the calls to scalars' `Hash()` functions in the FROST functions listed in the "Location" field of this finding. Generate hashes by using a cryptographic hash function of sufficient output size, e.g. SHA-512, and reducing its output modulo the group size. Include context strings in the hash input to ensure domain separation, thus correctly modelling the independent random oracles in security proofs.
- Ensure that test cases are meaningful. For example, check that `Random()` does not output the same value every time it is called.

Coinbase Category	Cryptographic / Mathematical

Finding	**Code is Not Constant-Time**
Risk	**Low** Impact: High, Exploitability: Undetermined
Identifier	NCC-E002578-005
Status	Reported
Category	Cryptography
Location	throughout
Impact	Information about secret values (such as keys and nonces) may leak through timing side-channels, possibility enabling signature forgery or impersonation attacks.

Description A commonly desired property of cryptographic code that operates on secret values is that it is *constant-time*: for all inputs of a certain length, the sequence of operations performed on that input (including reading particular memory addresses) depends *only* on the input's length, not its value. Constant-time code makes attacks on timing-based side channels impossible. These side channels can occur at various levels, ranging from microarchitectural (e.g. when data does or does not need to be loaded into a CPU cache), to implementation (e.g. exponentiation using the square-and-multiply algorithm), to protocol (e.g. returning an error instead of continuing). For example, timing information could be measured by another process running on the same hardware, or it could be measured remotely over a network.

For code to be constant-time, there must be no memory accesses at secret-dependent locations, nor any conditional jumps where the condition depends on the secret value. Several properties of the code in `kryptology` make it non-constant time and are detailed in this finding: (i) use of `math/big`, (ii) incorrect use of `crypto/subtle`, (iii) secret-dependent branching, (iv) data-dependent function API conformity, (v) short-circuit optimizations, and (vi) potential architecture-dependent optimizations. Additionally, (vii) some of `kryptology`'s dependencies are non-constant time.

(i) Use of `math/big`

The `math/big` package from Go's standard library is used extensively throughout `kryptology`. This package is meant to provide a *general-purpose* multi-precision library; its design is not meant to provide security properties such as constant-time execution or data-independent memory accesses.

For instance, consider the variable-length big integer type `big.Int`, used in `kryptology` to model coordinates of elliptic curve points, elliptic curve parameters, hash values, constants (`0`, 1, 2), etc. As the following code excerpts show, the `big.Int` type is a `struct` with two members: a boolean value (`neg`) representing whether the integer is positive or negative, and a (variable-length) slice (`nat`) of 32- or 64-bit words representing the absolute value of the integer.

```
type Int struct {
  neg bool // sign
  abs nat  // absolute value of the integer
}
```

Listing 1: math/big/int.go

```go
// An unsigned integer x of the form
//
//   x = x[n-1]*_B^(n-1) + x[n-2]*_B^(n-2) + ... + x[1]*_B + x[0]
//
// with 0 <= x[i] < _B and 0 <= i < n is stored in a slice of length n,
// with the digits x[i] as the slice elements.
//
// A number is normalized if the slice contains no leading 0 digits.
// During arithmetic operations, denormalized values may occur but are
// always normalized before returning the final result. The normalized
// representation of 0 is the empty or nil slice (length = 0).
//
type nat []Word
```

Listing 2: math/big/nat.go

```go
// A Word represents a single digit of a multi-precision unsigned integer.
type Word uint

const (
  _S = _W / 8 // word size in bytes

  _W = bits.UintSize // word size in bits
  _B = 1 << _W       // digit base
  _M = _B - 1        // digit mask
)
```

Listing 3: math/bits/arith.go

```go
const uintSize = 32 << (^uint(0) >> 63) // 32 or 64

// UintSize is the size of a uint in bits.
const UintSize = uintSize
```

Listing 4: math/bits/bits.go

In particular, the `big.Int` type does not have any notion of what length a value should *appear* to have, and there is no interface or means to force the library to allocate a certain number of words to a `big.Int`. Even when setting a `big.Int`'s value with `SetBytes()`, the value will be *normalized* and any leading zero words will be removed. This normalization entirely precludes writing constant-time code using `big.Int`s since the *size* of memory allocated to a `big.Int` will depend on its *value*, violating the requirement of constant-time code that any memory accesses are independent of secret values.

For example, suppose the upper 4 bytes of a user's secret value (e.g., a secret key or nonce) are zero — for a 256-bit value generated uniformly at random, this happens with probability about 1 in 4 billion, which may be frequent enough in some settings. When creating a `big.Int` and setting it to have this value, Go will allocate *seven* 32-bit words, not eight, on a 32-bit system. The normalization means that even if the `big.Int` was initialized by calling `SetBytes()` on a 32-byte array with the first 4 bytes equal to `0`, the internal representation of the value will always exclude the high-order zero word. Since the size of a `big.Int` in memory depends on its value, any functions involving it cannot be constant-time.

Additionally, the `math/big` package provides various functions operating on `big.Int`s that are themselves not constant-time. For example, some of the `big.Int` functions used in `kryptology` (primarily concentrated in pkg/core/curves) are `Add()`, `Sub()`, `Mul()`, `Mod()`, `ModInverse()`, `Exp()`, `ModSqrt()`, `Neg()`, `Cmp()`, and `Rsh()`. These functions provide no guarantees about the sequence of operations or memory accesses they perform.

There have been discussions about modifying the `math/big` package to be constant-time since 2017,[21] however, the proposal is still classified as "Incoming" (not yet "Accept" or "Likely Accept") among all of the Go language's current proposals.[22] Coinbase could consider evaluating available third-party libraries, such as `saferith`,[23] which is meant to provide a similar API to `math/big`, to see if they meet their requirements.

(ii) Incorrect Use of `subtle`

The `crypto/subtle` package in Go's standard library provides a few utility functions for comparing bytes, byte slices, and integers. The package's documentation states that the functions it provides "require careful thought to use correctly." These functions were written to run in constant time: the sequence of steps they take and the memory accesses they make depend only on the *size* of the arguments (e.g. the lengths of the byte slices), not their *values* or *contents*. For instance, consider the function `subtle.ConstantTimeCompare()`, copied here for reference.

```go
// ConstantTimeCompare returns 1 if the two slices, x and y, have equal contents
// and 0 otherwise. The time taken is a function of the length of the slices and
// is independent of the contents.
func ConstantTimeCompare(x, y []byte) int {
  if len(x) != len(y) {
    return 0
  }

  var v byte

  for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
  }

  return ConstantTimeByteEq(v, 0)
}
```

Listing 5: crypto/subtle/constant_time.go

It takes as input two byte slices `x` and `y`, and outputs 1 iff they have equal contents, otherwise it outputs `0`. Given two pairs of inputs (`x1`,`y1`) and (`x2`,`y2`), the constant-time guarantee is the following: if the lengths of `x1` and `x2` are equal, and the lengths of `y1` and `y2` are equal, then the function will perform the exact same sequence of operations when running on (`x1`,`y1`) as it will perform when running on (`x2`,`y2`). As an example, to compare two 256-bit values in constant time, two byte slices of length 32 must be passed to the function.

There are several instances in `kryptology` where this padding to fixed-length byte slices does not occur, and thus using `ConstantTimeCompare()` does not provide the desired constant-time guarantees.

- First, whenever `Bytes()` is used to convert a `big.Int` to a byte slice, its length will depend

[21] https://github.com/golang/go/issues/20654
[22] https://github.com/golang/go/projects/1
[23] https://github.com/cronokirby/saferith, https://eprint.iacr.org/2021/1121

on whether the higher-order bytes are zero, as the following code excerpt shows.

```
// Bytes returns the absolute value of x as a big-endian byte slice.
//
// To use a fixed length slice, or a preallocated one, use FillBytes.
func (x *Int) Bytes() []byte {
  buf := make([]byte, len(x.abs)*_S)
  return buf[x.abs.bytes(buf):]
}
```

Listing 6: math/big/int.go

The `big.Int` `Bytes()` function is used to generate the arguments to `ConstantTimeCompare()` in the following locations:

- function `ConstantTimeEqByte()` (pkg/core/mod.go, line 44)
- functions `IsZero()` and `IsOne()` (pkg/core/curves/bls12381_curve.go, lines 79 and 83)
- function `Set()` (pkg/core/curves/ed25519_curve.go, lines 490 and 491)
- functions `IsZero()`, `IsOne()`, and `Set()` (pkg/core/curves/k256_curve.go, lines 61, 65, 419, and 420)
- functions `IsZero()`, `IsOne()`, and `Set()` (pkg/core/curves/p256_curve.go, lines 61, 65, 418, and 419)
- function `Set()` (pkg/core/curves/pallas_curve.go, lines 564 and 565)

In all of these instances, the `big.Int`s should be serialized into a pre-allocated array of the correct size (e.g. the group size or hash size) using `FillBytes()`[24] instead of `Bytes()`.

Additionally, there are other instances where functions from `crypto.subtle` are used in ways that will not give the desired behavior.

- `ConstantTimeCopy()`[25] takes as input a bit and two byte slices. If the slices do not have equal length, the function panics. Therefore, this function should not be used with `big.Int`s converted to variable-length byte slices using `Bytes()`. This pattern occurs in the following locations:
  - function `ToAffineCompressed()` (pkg/core/curves/pallas_curve.go, line 894)

  Again, in this instance, any `big.Int`s should be serialized into a pre-allocated array of the correct size using `FillBytes()` instead of `Bytes()`.

(iii) Secret-Dependent Branching

For code to be constant-time, there must be no jumps or branching (with `if-else`, `switch`, `break`, `goto` etc.) depending on any secret value. Different branches may take different amounts of time to execute, and this difference may be exacerbated by CPU optimizations such as branch prediction. In general, the pattern `if (v) { return x } else { return y }` can be handled in constant-time with the `crypto/subtle` function `ConstantTimeSelect()`.[26] The `ConstantTimeCopy()` function may also be useful in replacing some of these branches.

Numerous examples of such secret-dependent branching were identified throughout the code: branching based on the result of comparisons involving `big.Int` values, `int` values, results of other non-constant time computations, etc. Some examples are identified in the following list. It is not exhaustive, but instead meant to illustrate the patterns that should be replaced in the code.

- The function `ConstantTimeEqByte()` (pkg/core/mod.go, line 35), which checks whether two `big.Int`s represent the same integer, is not constant-time (for reasons in addition to

[24] https://pkg.go.dev/math/big#Int.FillBytes
[25] https://pkg.go.dev/crypto/subtle#ConstantTimeCopy
[26] https://pkg.go.dev/crypto/subtle#ConstantTimeSelect

its use of `big.Int`s): it uses secret-dependent branching (the `if`-`else` statements on lines 44 and 52). This may leak whether the two integers have the same absolute value or the same sign.

```go
35 func ConstantTimeEqByte(a, b *big.Int) byte {
36   if a == nil && a == b {
37     return 1
38   }
39   if a == nil || b == nil {
40     return 0
41   }
42   // Determine if the byte representations are the same
43   var sameBytes byte
44   if subtle.ConstantTimeCompare(a.Bytes(), b.Bytes()) == 1 {
45     sameBytes = 1
46   } else {
47     sameBytes = 0
48   }
49
50   // Determine if the signs are the same
51   var sameSign byte
52   if a.Sign() == b.Sign() {
53     sameSign = 1
54   } else {
55     sameSign = 0
56   }
57
58   // Report the conjunction
59   return sameBytes & sameSign
60 }
```

- The `Ep` function `Add()` (pkg/core/curves/pallas_curve.go, line 772), which adds two points in projective coordinates, is not constant-time: it contains several occurrences of secret-dependent branching. These may leak, via timing side-channels, whether either of the operands is the identity, whether the two operands are equal, or whether they are inverses of each other.

```go
772 func (p *Ep) Add(lhs *Ep, rhs *Ep) *Ep {
773         if lhs.IsIdentity() {
774                 return p.Set(rhs)
775         }
776         if rhs.IsIdentity() {
777                 return p.Set(lhs)
778         }
779         z1z1 := new(fp.Fp).Square(lhs.z)
780         z2z2 := new(fp.Fp).Square(rhs.z)
781         u1 := new(fp.Fp).Mul(lhs.x, z2z2)
782         u2 := new(fp.Fp).Mul(rhs.x, z1z1)
783         s1 := new(fp.Fp).Mul(lhs.y, z2z2)
784         s1.Mul(s1, rhs.z)
785         s2 := new(fp.Fp).Mul(rhs.y, z1z1)
786         s2.Mul(s2, lhs.z)
787
788         if u1.Equal(u2) {
789                 if s1.Equal(s2) {
790                         return p.Double(lhs)
791                 } else {
792                         return p.Identity()
793                 }
794         } else {                                                // [snip...]
```

- The `ScalarPallas` function `Div()` (pkg/core/curves/pallas_curve.go, line 356) is not constant-time. The `if` statement on line 360 contains a check that, if triggered, makes the function return more quickly and without reading the `ScalarPallas` value `s`. This may leak whether the argument to `Div` was a scalar equal to `0` modulo the group order.

```
356 func (s *ScalarPallas) Div(rhs Scalar) Scalar {
357       r, ok := rhs.(*ScalarPallas)
358       if ok {
359           v, wasInverted := new(fq.Fq).Invert(r.value)
360           if !wasInverted {
361               return nil
362           }
363           v.Mul(v, s.value)
364           return &ScalarPallas{value: v}
365       } else {
366           return nil
367       }
368   }
```

(iv) Data-dependent Function API Conformity

For code to be constant-time, there must be no memory accesses made that depend on the values of potentially secret data. This principle applies to function APIs and is one of the reasons why any function that inputs or outputs a `big.Int` cannot be constant-time: when reading or writing a `big.Int`, the memory locations accessed depend on the integer's size, and its size depends on its value. More generally, any function that returns a value whose size depends on a potentially secret input is not constant-time. In particular, if a function returns `nil` in certain special cases, but returns a non-`nil` pointer to additional memory in other cases, this may leak through a timing side-channel.

For example, this pattern was observed in `kryptology`:

- The `ScalarPallas` function `Div()` (pkg/core/curves/pallas_curve.go, copied in the previous section) returns `nil` (line 361) if the scalar argument `rhs` had no inverse, while it allocates a scalar and returns a pointer to it (line 364) otherwise. This could leak whether the scalar `rhs` was `0`. This `Div()` function should instead return (a fixed-size representation of) `0`, such as what is returned by the `Fq` function `Invert()`.

(v) Short-Circuit Optimizations

Writing constant-time code is made more difficult by certain common optimizations built into Go. The Go Programming Language Specification[27] states:

> Logical operators apply to boolean values and yield a result of the same type as the operands. **The right operand is evaluated conditionally.**

Conditionally evaluating the right operand of a logical operator is known as short-circuiting, which allows for faster computation of the expression. Short-circuiting allows prematurely evaluating a logical AND (`&&`) expression to FALSE as soon as its first argument is computed to be FALSE, or prematurely evaluating a logical OR (`||`) expression to TRUE as soon as its first argument is TRUE. This short-circuiting results in shorter execution time and potentially fewer steps based on the first term of the logical operator.

A common technique for preventing such short-circuiting is using *bitwise* operators (`&` and `|`) on integer types instead of logical operators (`&&` and `||`) on booleans. One way to ensure

[27] https://golang.org/ref/spec#Logical_operators

these operators are not used is to remove all booleans (`bool`) and use integer types instead, with `0` for false and 1 for true. The `crypto/subtle` functions `ConstantTimeCopy()` and `ConstantTimeSelect()` are made to use on such integers representing boolean values.

Short-circuiting may affect the following functions in `kryptology`.

- The `Ep` (Pasta point) `Equal()` function (pkg/core/curves/pallas_curve.go, line 847), which checks whether two points are equal, could return FALSE as soon as `lhs.x.Equal(rhs.x)` is computed to be FALSE. This may leak whether two points have the same (affine) x-coordinate, i.e., whether the two points are inverses or are identical.

```
func (p *Ep) Equal(other *Ep) bool {
  // warning: requires converting both to affine
  // could save slightly by modifying one so that its z-value equals the other
  // this would save one inversion and a handful of multiplications
  // but this is more subtle and error-
  ➜ prone, so going to just convert both to affine.
  lhs := new(Ep).Set(p)
  rhs := new(Ep).Set(other)
  lhs.toAffine()
  rhs.toAffine()
  return lhs.x.Equal(rhs.x) && lhs.y.Equal(rhs.y)
}
```

- The `PallasCurve` `Add()` function, which adds two points specified by their `big.Int` coordinates, may leak whether either of the operands is the identity.
  Consider using a *complete* formula for elliptic curve with no branches instead. The point addition formulas for prime-order short Weierstrass curves in "Complete addition formulas for prime order elliptic curves" by Renes, Costello, and Batina[28] would be suitable.

```
func (curve *PallasCurve) Add(x1, y1, x2, y2 *big.Int) (*big.Int, *big.Int) {
      p := new(Ep)
      p.x = new(fp.Fp).SetBigInt(x1)
      p.y = new(fp.Fp).SetBigInt(y1)
      p.z = new(fp.Fp).SetOne()
      if p.x.IsZero() && p.y.IsZero() {
          p.z.SetZero()
      }

      q := new(Ep)
      q.x = new(fp.Fp).SetBigInt(x2)
      q.y = new(fp.Fp).SetBigInt(y2)
      q.z = new(fp.Fp).SetOne()
      if q.x.IsZero() && q.y.IsZero() {
          q.z.SetZero()
      }
      p.Add(p, q)
      p.toAffine()
      return p.x.BigInt(), p.y.BigInt()
  }
```

- The `EcPoint` function `IsValid()` (pkg/core/curves/ec_point.go, line 103), which checks if a point is on a curve, may leak whether the given point is on the curve, but not equal to the identity.

```
func (a EcPoint) IsValid() bool {
      return a.IsOnCurve() || a.IsIdentity()
```

[28] https://eprint.iacr.org/2015/1060

```
  }

```

(vi) Architecture-Dependent Optimizations

Constant-time code is often written for specific platform architectures, e.g. 32- or 64-bit. What is constant-time on one platform may not be constant-time on another.

This may affect `kryptology` in the following locations.

- The implementation of the Pasta curves in (pkg/core/curves/native/pasta) includes the types `Fp` and `Fq` for field elements modulo the Pallas and Vesta curve orders respectively. These types are implemented as length-4 arrays of `uint64`s. Operations on `uint64`s that are constant-time on a 64-bit system may no longer be constant-time if the code were to be compiled on a 32-bit system. For example, consider the Pasta scalar `Equal()` function (pkg/core/curves/native/pasta/fp/fp.go, line 57), which checks whether two scalars are equal.

```go
// Equal returns true if fp == rhs
func (fp *Fp) Equal(rhs *Fp) bool {
  t := fp[0] ^ rhs[0]
  t |= fp[1] ^ rhs[1]
  t |= fp[2] ^ rhs[2]
  t |= fp[3] ^ rhs[3]
  return t == 0
}
```

  When this code is compiled on a 32-bit system, the check of whether the `uint64` value `t` equals `0` will be converted into an operation on two `uint32`s, which must both be compared against `0`. If the first of these comparisons returns FALSE, then the function may return sooner. To prevent this potential timing side channel, consider folding the 64-bit result into two 32-bit halves and ORing them before checking for equality with `0`: `return uint(x | (x >> 32)) == 0`.

(vii) Non-Constant Time Dependencies

Many of the supported curves in `kryptology` use third-party backends for elliptic curve computations, some of which claim to be constant-time and some of which make no claims.

Some general observations apply to these libraries:

- Using `big.Int` to model scalars or point coordinates is not constant-time. Any function that accepts a `big.Int` as input or output, or uses a `big.Int` for intermediate values in a computation is not constant-time.
- Using look-up tables to speed up computations is not constant-time if the index of the look-up (and therefore the resulting memory access read) is secret-dependent.

This information is summarized here.

- **BLS12-381**: The backend is proprietary (github.cbhq.net/c3/bls12-381); its constant-time properties are unknown.
- **Ed25519**: Multiple backends are used:
  - filippo.io/edwards25519: appears to be **constant-time**.
  - filippo.io/edwards25519/field: appears to be **constant-time**.
  - github.com/bwesterb/go-ristretto: **partly constant-time** due to `Scalar` functions `BigInt()` and `SetBigInt()`, which convert to/from `big.Int`s. However, these functions do not appear to be used in `kryptology`.
  - github.com/bwesterb/go-ristretto/edwards25519 (as "ed"): **partly constant-time** due to

`FieldElement` functions `BigInt()`, `SetBigInt()`, and `String()`, which all convert to/from `big.Int`s. These conversion functions are used in `kryptology`, e.g. in the `PointEd25519` function `Set()`, which calls `new(ed.FieldElement).SetBigInt(x)` (ed25519_curve.go, line 495).

- **K256 (secp256k1)**: The backend is github.com/btcsuite/btcd/btcec, which is **not constant-time**.
  - It extensively uses `big.Int`s for scalars, point coordinates, and intermediate values during computations.
  - Its `ScalarMult()` function[29] performs some optimizations on the scalar (via the `splitK()` function) that depend on its value.
  - Its `ScalarBaseMult()` function[30] uses a look-up table of pre-computed values and the access to this table depend on the value of the scalar.

  Coinbase could consider using Go bindings[31] to the (C) bitcoin-core secp256k1 library.[32]

- **P256 (secp256r1)**: The backend is crypto/elliptic, which claims in p256.go to be constant-time,[33] but is only **partly constant-time**.
  - The `p256GetScalar()` function uses `math/big`'s comparison and modular reduction functions on a `big.Int` representing the scalar.[34] This function is used in `ScalarBaseMult()` and `ScalarMult()`, the two scalar multiplication functions provided for P256.
  - The `p256ToBig()` function uses `math/big`'s multiplication and modular reduction functions on a `big.Int` representing the scalar.[35]
  - The `p256FromBig()` function, which converts a scalar to Montgomery form, uses `math/big`'s modular reduction function on the scalar.[36]
  - Many functions provided by `crypto/elliptic` do not yet have constant-time versions when operating on the curve P256, e.g. `IsOnCurve()`, `Add()`, and `Double()`. While the main body of the scalar multiplication functions are constant-time, they can take as input or output `big.Int`s representing scalars or point coordinates. (Note that this library uses a look-up table of pre-computed values for scalar multiplications of the base point. However, its accesses to this table do not depend on the scalar's value, so this is constant-time.)

- **Pallas/Vesta**: The backend is proprietary, but included in `kryptology` (in core/curves/native/pasta). It is **partly constant-time**. While the main bodies of most functions (with some exceptions, as covered by the patterns explained in this finding) are constant-time, they often take as input or output `big.Int`s representing scalars or point coordinates.
  - Various functions may leak when a coordinate is `0` or when two coordinates are equal.
  - Certain functions use `big.Int`s for potentially secret-dependent intermediate values, e.g. `sumOfProductsPippengerPallas()` (pkg/core/curves/pallas/pallas_curve.go, line `1037`, copied below) creates `big.Int`s from the (also `big.Int`) scalars it takes as input (line `1037`). Additionally, the result of this computation, `index`, which is based on a potentially secret scalar, is used to conditionally perform several memory accesses and scalar additions. To make this algorithm constant-time, the memory accesses and addition must always be performed (and the intermediate results always set with bitwise operations).

[29] https://github.com/btcsuite/btcd/blob/master/btcec/btcec.go#L765, `splitK()` on line 771
[30] https://github.com/btcsuite/btcd/blob/master/btcec/btcec.go#L870, secret-dependent look-ups on line 883
[31] https://github.com/btccom/secp256k1-go
[32] https://github.com/bitcoin-core/secp256k1
[33] https://github.com/golang/go/blob/master/src/crypto/elliptic/p256.go#L9
[34] https://github.com/golang/go/blob/master/src/crypto/elliptic/p256.go#L54--L55
[35] https://github.com/golang/go/blob/master/src/crypto/elliptic/p256.go#L1192-L1193
[36] https://github.com/golang/go/blob/master/src/crypto/elliptic/p256.go#L1152-L1153

```go
func sumOfProductsPippengerPallas(points []*Ep, scalars []*big.Int) *Ep {
	if len(points) != len(scalars) {
		return nil
	}

	const w = 6

	bucketSize := (1 << w) - 1
	windows := make([]*Ep, 255/w+1)
	for i := range windows {
		windows[i] = new(Ep).Identity()
	}
	bucket := make([]*Ep, bucketSize)

	for j := 0; j < len(windows); j++ {
		for i := 0; i < bucketSize; i++ {
			bucket[i] = new(Ep).Identity()
		}

		for i := 0; i < len(scalars); i++ {
			index := bucketSize & int(new(big.Int).Rsh(scalars[i], uint(w*j))
			➜ .Int64())
			if index != 0 {
				bucket[index-1].Add(bucket[index-1], points[i])
			}
		}                                                    // [snip...]
```

Recommendation First, consider documenting the code to identify which values are considered secret (or not), and which functions are guaranteed to run in constant time (or variable time). Then, for all code that may be run on secret values and should be constant-time, address the points from this finding:

- Do not use the `big.Int` type provided by the `math/big` package. Consider using an alternative library designed to be constant-time, and/or use curve-specific constant-time implementations for each supported curve group.
- Ensure that the lengths of arguments to comparison functions from the `crypto/subtle` package do not depend on their values (i.e. that no leading zero bytes are removed).
- Remove any instances of secret-dependent branching; use the `crypto/subtle` functions `ConstantTimeCopy()` and `ConstantTimeSelect()` where appropriate.
- Ensure that functions' APIs are used uniformly to avoid leaking information about the return values (e.g., by sometimes returning `nil` instead of allocating an array).
- Avoid using boolean types and logical operators, which may result in short-circuit optimizations; use bitwise functions on integer types instead.
- Be aware of differences due to code compilation on different architectures.
- Ensure that no non-constant time functions are used from dependency packages and/or consider replacing non-constant time dependency packages with suitable replacements.

Coinbase Category Security / Implementation issues

Finding	**Implementation Does Not Identify Misbehaving Participants**
Risk	**Low** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-E002578-006
Status	Reported
Category	Cryptography
Component	DKG, Signing
Location	• kryptology/pkg/dkg/frost/dkg_round2.go • kryptology/pkg/ted25519/frost/round2.go • kryptology/pkg/ted25519/frost/round3.go

Impact Misbehaving participants can corrupt the DKG and signing protocols anonymously, which may lead to an indefinite denial of service since the caller is not given any information about which participants to eject.

Description The FROST protocol is a *non-robust* signature scheme, designed to abort upon detection of misbehaving participants. The FROST eprint paper[37] states:

> FROST achieves its efficiency improvements in part by allowing the protocol to abort in the presence of a misbehaving participant (who is then identified and excluded from future operations)

In the `frost` package in `kryptology`, when validation of another party's values fails during DKG or signing, no information about the identity of the misbehaving participant is supplied as output, making it impossible to eject badly behaving participants in the future. The error messages included in the list below refer only to the *existing* checks in the code. Note that additional required checks (see finding NCC-E002578-010 on page 13) should also provide such information.

- **DKG, round 2:** Each party verifies the `Round1Bcast` values (`verifiers`, `wi`, `ci`) and Shamir share (`fji`) it received from each other party. The following two checks are included in `kryptology`'s implementation:
  - First, the party verifies the other party's Schnorr Proof of Knowledge (PoK) of its share of the secret key. This is done by checking that the received hash value `cj` equals the locally computed hash of `id`, `ctx`, `Aj0`, and `prod`. In terms of notation from FROST's pseudocode, this check corresponds to

    $$c_j \overset{?}{=} H(j, CTX, A_{j,0}, w_j \cdot G + (-c_j) \cdot A_{j,0}).$$

    This equation is checked on line 70 of pkg/dkg/frost/dkg_round2.go:

    ```
    70  if cj.Cmp(bcast[id].ci) != 0 {
    71    return nil, fmt.Errorf("Hash check fails")
    72  }
    ```

    There is no identification of the responsible participant (`id`), only an error message that something went wrong with the comparison.
  - Second, the party verifies the correctness of their assigned Shamir share using Feldman verifiable secret sharing's verification algorithm. This is done on line 76 of pkg/dkg/frost/dkg_round2.go:

[37] https://eprint.iacr.org/2020/852

```go
// Step 5 - FeldmanVerify
fji := p2psend[id]
if err = bcast[id].verifiers.Verify(fji); err != nil {
  return nil, err
}
```

Again, if this verification fails, the protocol simply returns `nil` without identifying the misbehaving participant, `id`.

- **Signing, round 2:** Each party verifies that it received non-`nil` `Round1Bcast` values (`Di`, `Ei`) from each other party.

```go
// Step 2 - Check Dj, Ej on the curve and Store round2Input
for _, input := range round2Input {
  if input == nil || input.Di == nil || input.Ei == nil {
    return nil, fmt.Errorf("some round2Input is nil")
  }
}
```

Again, there is no identification of the participant (`id`) who broadcasted invalid commitments. Also note that, contrary to the comment, there is no check that the points are on the curve. (This is addressed in finding NCC-E002578-010 on page 13.)

- **Signing, round 3:** Each party verifies that the `Round2Bcast` values (`zi`, `vki`) it received from each other party represent a valid share of the signature. This is done on line 90 of pkg/ted25519/frost/round3.go, where, again, the returned error message does not highlight which user submitted an invalid response.

```go
89 // Check equation
90 if !zjG.Equal(right) {
91   return nil, fmt.Errorf("zjG != right")
92 }
```

In all of these instances, it is recommended to include the identity of the party responsible for the error in the error message returned to the calling function. It is also important to check *all* parties' values before returning an error, otherwise, only the first misbehaving party will be identified.

Recommendation

- Ensure that the DKG and signing protocol APIs return informative error messages when failing due to another party's inputs, so that the misbehaving participant(s) can be ejected or excluded from future protocol instances.
  - The additional required checks described in finding NCC-E002578-010 on page 13 should also return error messages identifying misbehaving parties.
- Ensure that *all* misbehaving parties, not just the first, are identified. Do not immediately return an error after the first misbehaving party is detected; continue processing other parties' messages and return a *list* of the identities of misbehaving parties.

Coinbase Category Security / Implementation issues

Finding	**Minor Deviations from FROST Specification**
Risk	**Low** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-E002578-009
Status	Updated
Category	Configuration
Component	Signing
Location	• kryptology/pkg/ted25519/frost/round2.go • kryptology/pkg/ted25519/frost/challenge_derive.go
Impact	Deviating from the FROST specification may cause compatibility issues with other implementations. In the context of blockchains, if one implementation considers a signature valid and another implementation considers it invalid, this may result in a fork.

Description Two small deviations were noted from the FROST specification in CFRG Internet Draft draft-komlo-frost-00,[38] apart from the intentionally made changes around the removal of the Signature Aggregator role. Both differences relate to the inputs of hash functions: in one instance, `kryptology` includes additional input, while in the other, `kryptology` omits some input. In general, including more input when hashing a bytestring improves security: it makes the hash value more specific, which may rule out classes of attacks that rely on the re-use of ambiguous hash values. (See also the note about including context strings in hash inputs to ensure domain separation in finding NCC-E002578-003 on page 17.) In this finding, the differences are pointed out simply for awareness about potential incompatibilities with other implementations of draft-komlo-frost-00. (Note that this specification is no longer the most recent one; see finding NCC-E002578-007 on page 36.)

First, in `kryptology`, the challenge c computed during the signing protocol additionally includes the public key: it is computed as $c = H(R, Y, m)$. This is done in the `DeriveChallenge()` function (pkg/ted25519/frost/challenge_derive.go):

```go
func (ed Ed25519ChallengeDeriver) DeriveChallenge(msg []byte, pubKey
➜ curves.Point, r curves.Point) (curves.Scalar, error) {
	h := sha512.New()
	_, _ = h.Write(r.ToAffineCompressed())
	_, _ = h.Write(pubKey.ToAffineCompressed())
	_, _ = h.Write(msg)
	return new(curves.ScalarEd25519).SetBytesWide(h.Sum(nil))
}
```

In draft-komlo-frost-00, the public key is not included and the arguments are in a different order:

```
>   4.  Each P_(i) then computes the set of binding values r_(p) =
>      H_(1)(p, m, B), p in S.  Each P_(i) then derives the group
>      commitment R = PROD(D_(pj) * (E_(pj))^{r_(p)}, p in S), and the
>      challenge c = H_(2)(m, R).
```

Second, also in the signing protocol, each party computes a binding value $\rho_i = H(i, m, B)$ for each other party i in the set of signers S, where $B = \langle(i, D_i, E_i)\rangle_{i \in S}$. The relevant excerpt from draft-komlo-frost-00 is:

[38] https://www.ietf.org/archive/id/draft-komlo-frost-00.txt

```
>   selected for use for this signing operation.  Let B = < (i, D_(ij),
>   E_(ij)) for i in S> denote the ordered list of participant indices
>   corresponding to each participant P_(i), and L_(i) be the set of
>   available commitment values for P_(i) that were published during the
>   Preprocess stage.  Each identifier i is coupled with the jth
>   commitments (D_(ij), E_(ij)) published by P_(i) that will be used for
>   this particular signing operation.  Let H_(1), H_(2) be hash
>   functions whose outputs are in Z_(q)^(*).
>
>   1.  SA begins by fetching the next available commitment for each
>      participant P_(i) in S from L_(i) and constructs B.
>
>   2.  For each i in S, SA sends P_(i) the tuple (m, B).
>
>   3.  After receiving (m, B), each P_(i) first validates the message m,
>      and then checks D_(p j), E_(p j) in G^(*) for each commitment in
>      B, aborting if either check fails.
>
>   4.  Each P_(i) then computes the set of binding values r_(p) =
>      H_(1)(p, m, B), p in S.  Each P_(i) then derives the group
>      commitment R = PROD(D_(pj) * (E_(pj))^{r_(p)}, p in S), and the
>      challenge c = H_(2)(m, R).
```

This is done on line 68 of pkg/ted25519/frost/round2.go with the `concatHashArray()` function:

```
66    for id, data := range round2Input {
67      // Construct the blob (j, m, {Dj, Ej})
68      blob := concatHashArray(id, msg, round2Input, signer.cosigners)
```

The `concatHashArray()` function is defined as follows, in the same file:

```
func concatHashArray(id uint32, msg []byte, round2Input map[uint32]*Round1Bcast,
 ➜  cosigners []uint32) []byte {
  var blob []byte
  // Append identity id
  blob = append(blob, byte(id))

  // Append message msg
  blob = append(blob, msg...)

  // Append (Dj, Ej) for all j in [1...t]
  for i := 0; i < len(cosigners); i++ {
    id := cosigners[i]
    bytesDi := round2Input[id].Di.ToAffineCompressed()
    bytesEi := round2Input[id].Ei.ToAffineCompressed()
    blob = append(blob, bytesDi...)
    blob = append(blob, bytesEi...)
  }
  return blob
}
```

The two highlighted lines show that only D_i (`bytesDi`) and E_i (`bytesEi`) are included, not the identifier i (`id`) of the party. To comply with the FROST specification, each party's identifier should also be included.

Recommendation If compatibility with other implementations of draft-komlo-frost-00 is desired, then make the

	following changes: • In the `DeriveChallenge()` function in pkg/ted25519/frost/challenge_derive.go, remove the group public key from the bytestring to be hashed. • In the `concatHashArray()` function in pkg/ted25519/frost/round2.go, include each party's ID in the bytestring to be hashed.
Coinbase Category	Cryptographic / Mathematical

Finding	**FROST Implementation Does Not Follow Most Recent Specification**
Risk	**Informational** Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-E002578-007
Status	New
Category	Cryptography
Component	FROST
Location	• kryptology/pkg/dkg/frost/ • kryptology/pkg/ted25519/frost/
Impact	Basing the FROST implementation on an expired specification (draft-irtf-cfrg-frost-00) may prevent it from benefiting from security enhancements in later drafts.

Description

FROST is a relatively new protocol: the paper by Komlo and Goldberg that introduced it was first published on eprint[39] in July 2020. The `kryptology` implementation of FROST is based on the CFRG (Crypto Forum Research Group) Internet Draft draft-komlo-frost-00[40] from August 2020. When the CFRG adopted it as a work item, the draft expired and was replaced by draft-irtf-cfrg-frost-00[41] in February 2021. Drafts under consideration to be adopted as RFCs (Request For Comments) are updated at least once every 6 months; the FROST draft was updated to draft-irtf-cfrg-frost-01[42] in August 2021. The latest available version of the paper on eprint[43] (which was published in the proceedings of SAC 2020) is from December 2020.

Later versions of the FROST specification may introduce security enhancements, therefore, it is recommended to keep the implementation up to date with respect to the latest Internet Draft. Currently, there appear to be only a few small differences between the FROST implementation in `kryptology` and the latest available FROST paper and specification.

Specification of Domain-Separated Hashes

The latest Internet Draft, draft-irtf-cfrg-frost-01, includes details about how to use a general-purpose cryptographic hash function to implement the hash functions $H_1()$, used to derive the binding factors ρ_i during signing, and $H_2()$, used to derive the Schnorr challenge c during signing:

```
6.2.  Cryptographic Hash Function

   FROST requires the use of a cryptographically secure hash function,
   generically written as H, which functions effectively as a random
   oracle.  For concrete recommendations on hash functions which SHOULD
   BE used in practice, see Section 9.

   Using H, we introduce two separate domain-separated hashes, H1 and
   H2, where H1(m) = H("rho" || len(m) || m) and H2(m) = H("chal" ||
   len(m) || m).
```

[39] https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2020/852&version=20200712:124135&file=852.pdf
[40] https://www.ietf.org/archive/id/draft-komlo-frost-00.txt
[41] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-frost-00
[42] https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-frost-01
[43] https://eprint.iacr.org/eprint-bin/getfile.pl?entry=2020/852&version=20201222:190959&file=852.pdf

Schnorr Proof of Knowledge (PoK) of Secret (a_{i0}) During DKG

FROST's DKG protects against rogue key attacks by including a proof of knowledge (PoK) of secret values a_{i0}. The older versions of FROST use Schnorr signatures, where the proof is in $\mathbb{Z}_q \times \mathbb{Z}_q$. The newer version of FROST uses the honest-verifier zero-knowledge (HVZK) proof derived from Schnorr's identification protocol, where the proof is in $\mathbb{G} \times \mathbb{Z}_q$.

In more detail, the differences are the following:

- In `kryptology`, which follows the eprint paper from July 2020 and draft-komlo-frost-00 from August 2020:
  - The proof for $g^{a_{i0}}$ is generated during round 1 of DKG as $\sigma_i = (\mu_i, c_i) \in \mathbb{Z}_q \times \mathbb{Z}_q$, where $\mu_i = k_i + a_{i0} \cdot c_i \bmod q$ for a random $k_i \in \mathbb{Z}_q$, and $c_i = H(i, \Phi, g^{a_{i0}}, R_i)$, for $R_i = g^{k_i}$ and a context string Φ.
  - The proofs are verified during round 2 of DKG by computing $X_i = g^{\mu_i} \cdot (g^{a_{i0}})^{-c_i}$ and checking $c_i \overset{?}{=} H(i, \Phi, g^{a_{i0}}, X_i)$.
- In the eprint paper from December 2020[44]:
  - The proof for $g^{a_{i0}}$ is generated during round 1 of DKG as $\sigma_i = (R_i, \mu_i) \in \mathbb{G} \times \mathbb{Z}_q$, where $R_i = g^{k_i}$ for a random $k_i \in \mathbb{Z}_q$, and $\mu_i = k_i + a_{i0} \cdot c_i \bmod q$, where $c_i = H(i, \Phi, g^{a_{i0}}, R_i)$, for a context string Φ.
  - The proofs are verified during round 2 of DKG by computing the hash $c_i = H(i, \Phi, g^{a_{i0}}, R_i)$ and checking $R_i \overset{?}{=} g^{\mu_i} \cdot (g^{a_{i0}})^{-c_i}$.

Recommendation

- Keep the implementation of FROST in `kryptology` up to date with the latest available draft specification from https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/. The next update is expected by February 2022.
  - Rewrite the proof of knowledge following the protocol as specified in the updated paper and Internet Draft.
- Consider monitoring the issues reported on the draft author's GitHub repository (https://github.com/chelseakomlo/frost-spec/issues).

Coinbase Category Security / Implementation issues

[44] Note that the latest Internet Draft of FROST does not include DKG.

Finding `ScalarP256`'s `SetBytesWide()` Method May Return Incorrect or Non-Canonical Scalars

Risk	**Informational** Impact: Low, Exploitability: None
Identifier	NCC-E002578-011
Status	New
Category	Cryptography
Component	curves
Location	kryptology/pkg/core/curves/p256_curve.go

Impact

- The `ScalarP256` value returned by the `SetBytesWide()` function may return scalars that are not equal to the input `bytes` (interpreted as a big-endian unsigned integer) modulo the order of the parent curve secp256r1 (N_{P256}).
- The returned value s may be larger than N_{P256}, which could result in an otherwise honest party mistakenly being identified as malicious for sending a non-canonical scalar (i.e. one not satisfying $0 \leq s < N_{P256}$).

Description

In `kryptology`, the `ScalarP256` method `SetBytesWide()` reduces an integer modulo the order of the wrong curve — secp256k1, instead of secp256r1. This error is similar to the one in the function `IsOnCurve()` (see finding NCC-E002578-004 on page 7) and could also have happened when copy-pasting code from pkg/core/curves/k256_curve.go.

The error is highlighted in the code snippet below from pkg/core/curves/p256_curve.go:

```go
func (s *ScalarP256) SetBytesWide(bytes []byte) (Scalar, error) {
        if len(bytes) < 32 || len(bytes) > 128 {
                return nil, fmt.Errorf("invalid byte sequence")
        }
        value := new(big.Int).SetBytes(bytes)
        value.Mod(value, btcec.S256().N)
        return &ScalarP256{
                value,
        }, nil
}
```

The order of `secp256k1` (the Bitcoin curve) is
$N_{S256} = $ `FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141`,

while the order of `secp256r1` is smaller:
$N_{P256} = $ `FFFFFFFF00000000FFFFFFFFFFFFFFFFBCE6FAADA7179E84F3B9CAC2FC632551`.

Using the wrong group order has two implications.

First, if the input `bytes`, when interpreted as a big-endian unsigned integer, correspond to a value greater than N_{S256}, then at least one modular reduction will occur, and the output value will not be equal to the input modulo N_{P256}, as the caller of the function would expect. The correctness of other computations may be affected.

Second, if the input `bytes` correspond to a value greater than N_{P256}, then it is possible that the output is in the range $[N_{P256}, N_{S256}-1]$. The number of values in this range is about 2^{224}, which means that the probability that the output of `SetBytesWide()` on a random, sufficiently long bytestring is in this range is about $2^{224}/2^{256} = 1/2^{32}$, or about 1 in 4.2 billion. If the output scalar s is then sent to other parties as part of a protocol, and those parties validate the scalar by verifying that it is canonically represented (i.e. that $0 \leq s < N_{P256}$), this validation would not succeed. If this were to happen in a protocol like FROST that is *non-robust* and designed to abort when it identifies misbehaving participants, then it could result in an honest participant being identified as misbehaving and causing the protocol to abort, despite them not intentionally having misbehaved.

Currently in the FROST implementation, only the `SetBytesWide()` method for Ed25519 scalars appears to be used (in pkg/ted25519/frost/challenge_derive.go, line 25), so this vulnerability is not exploitable.

Recommendation Fix the function `SetBytesWide()` in kryptology/pkg/core/curves/p256_curve.go by replacing `btcec.S256().N` with `elliptic.P256().Params().N`.

Coinbase Category Security / Implementation issues

Appendix A: Notes

This appendix contains additional notes and observations that did not warrant findings, but were deemed worthy of pointing out.

- **Misleading function name** in pkg/sharing/shamir.go: the `Shamir` method `getPolyAndShares()` returns `shares, poly` rather than `poly, shares`.
- **Typo** in pkg/core/curves/curve.go: `BLS12831Name` has the `8` and `3` transposed: `BLS12831Name    = "BLS12831"`.
- **`ScalarBls12381Gt` struct is incorrect** in pkg/core/curves/bls12381_curve.go. The target group G_T should have the same order (q, 255 bits) as G_1 and G_2, however, the implementations of many of its methods appear to use 576-byte values, as if they were constructing field elements of the extension $\mathbb{F}_p^{12}$ (for p of 381 bits).
- **Hard-to-read code** due to design choices. Structs implementing the `Scalar` interface define implementations of a `Random()` method. Although this is a method of a `Scalar` object, it does not at all use the member variable, `value`, of the `Scalar`. Instead, it returns a new `Scalar`. This results in confusing code, such as the following, from pkg/sharing/polynomial.go:

```
18 func (p *Polynomial) Init(intercept curves.Scalar, degree uint32, reader io.Reader) *Polynomial {
19   p.Coefficients = make([]curves.Scalar, degree)
20   p.Coefficients[0] = intercept.Clone()
21   for i := 1; i < int(degree); i++ {
22     p.Coefficients[i] = intercept.Random(reader)
23   }
24   return p
25 }
```

  The generation of random scalars on line 22 has nothing to do with the secret `intercept` which is used as the constant term in the polynomial. A more appropriate design could be to have a function `RandomScalar()` as a method of the curve.
- **`xor()` function may silently fail** in pkg/core/hash.go. The `xor()` function (copied below) expects both byte arrays to be the same length, but does not implement any handling when this is not the case. If this function were used in a situation where `len(b2) < len(b1)`, then the code would panic with an "index out of range" runtime error. If it were used with `len(b1) < len(b2)`, then the last byte(s) of `b2` would simply be ignored. Currently, the function is used only in the `core` package, in `ExpandMessageXmd()`, where its usage appears safe.

```
func xor(b1, b2 []byte) []byte {
  // b1 and b2 must be same length
  result := make([]byte, len(b1))
  for i := range b1 {
    result[i] = b1[i] ^ b2[i]
  }

  return result
}
```

- **Left-over code** that does not appear to be used anymore still exists in a few places. Removing dead code will improve ease of maintenance and readability of the codebase, and prevent any vulnerabilities arising from its accidental use.
  - pkg/core/curves/pallas_curve.go: The old, `big.Int`-based scalar implementation (`PallasScalar`) is still in the file (lines 158–206), alongside the new implementation (`ScalarPallas`).
  - pkg/core/curves/ec_scalar.go: The `EcScalar` interface defined here (and the structs that implement it — `K256Scalar`, `P256Scalar`, `Bls12381Scalar`, and `Ed25519Scalar`) are not used in FROST; FROST uses the `Scalar` interface defined in pkg/core/curves/curve.go (and the types `ScalarK256`, `ScalarP256`, `ScalarBls12381`, and `ScalarEd25519`).
  - pkg/core/curves/ec_point.go: The `EcPoint` struct defined here is not used in FROST; FROST uses the `Scalar` interface defined in pkg/core/curves/curve.go and curve-specific structs.
  - pkg/sharing/v1/: The 19 files in this folder all appear to have been replaced elsewhere.
- **Verifiable Secret Sharing will not split `0`** in pkg/sharing/feldman.go. Note that, in the future, it is possible that `kryptology` requires creating secret shares of `0`, e.g. if a protocol requires secret-sharing a sign bit. The check that the secret is not zero (on line 65, copied below) could be moved outside of the function wherever it is used, e.g.

when splitting a_{i0} in pkg/dkg/frost/dkg_round1.go, line 56. There, the check for whether the secret is `0` could occur directly after sampling it (on line 44).

```go
func (f Feldman) Split(secret curves.Scalar, reader io.Reader) (*FeldmanVerifier, []*ShamirShare, error) {
  if secret.IsZero() {
    return nil, nil, fmt.Errorf("invalid secret")
  }
```

- **Assumption about imported `FieldElement` type from edwards25519 package** in `cselect()` in pkg/core/curves/ed25519_curve.go. The dependency github.com/bwesterb/go-ristretto/edwards25519 uses build constraints to provide two different implementations of the `FieldElement` type:
  - field_generic.go[45]: `type FieldElement [10]int32`. The build constraint `// +build !amd64,!go1.13 forcegeneric` indicates that this file will be included if either (i) the target architecture is not `amd64` and the version of Go is earlier than 1.13, or (ii) the `forcegeneric` build tag is supplied.
  - field_radix51.go[46]: `type FieldElement [5]uint64`. The build constraint `// +build amd64,!forcegeneric go1.13,!forcegeneric` indicates this file will be included if (i) the target architecture is `amd64` and the `forcegeneric` build tag is not supplied, or (ii) the version of Go is at least 1.13 and the `forcegeneric` build tag is not supplied.

  The `cselect()` conditional select function in ed25519_curve.go assumes `ed.FieldElement` is an array of length 5, as when the ed25519 dependency is built with field_radix51.go.

```go
542 // cselect sets v to a if cond == 1, and to b if cond == 0.
543 func cselect(v, a, b *ed.FieldElement, cond bool) *ed.FieldElement {
544   const mask64Bits uint64 = (1 << 64) - 1
545 
546   m := uint64(0)
547   if cond {
548     m = mask64Bits
549   }
550 
551   v[0] = (m & a[0]) | (^m & b[0])
552   v[1] = (m & a[1]) | (^m & b[1])
553   v[2] = (m & a[2]) | (^m & b[2])
554   v[3] = (m & a[3]) | (^m & b[3])
555   v[4] = (m & a[4]) | (^m & b[4])
556   return v
557 }
```

  If `kryptology` were ever built for a 32-bit architecture or a non-`amd64` system, with a version of Go older than 1.13, this code would fail to compile, since Go does not perform any implicit conversions of numeric types. (Also note that there is secret-dependent branching on line 547 – see the recommendations in section (iii) of finding NCC-E002578-005 on page 21, which also suggest replacing boolean types and logical operators with bitwise functions on integer types.)

[45] https://github.com/bwesterb/go-ristretto/blob/master/edwards25519/field_generic.go#L7
[46] https://github.com/bwesterb/go-ristretto/blob/59c0de2354cd7e534807bba5f608e4ee1933082e/edwards25519/field_radix51.go#L7

Appendix B: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

Appendix C: Project Contacts

The team from NCC Group has the following primary members:

- Marie-Sarah Lacharité — Consultant
  marie-sarah.lacharite@nccgroup.com
- Giacomo (Jack) Pope — Consultant (Shadow)
  giacomo.pope@nccgroup.com
- Javed Samuel — Practice Director, Cryptography Services
  javed.samuel@nccgroup.com

The team from Coinbase, Inc. has the following primary members:

- Jeff Barksdale
  jeff.barksdale@coinbase.com
- Michael Lodder
  mike.lodder@coinbase.com
- Luis Ocegueda
  luis.ocegueda@coinbase.com
- Daniel Zhou
  daniel.zhou@coinbase.com