Workgroup: Network Working Group
Internet-Draft: draft-hallambaker-threshold
Published: 20 April 2022
Intended Status: Informational
Expires: 22 October 2022
Authors: P. M. Hallam-Baker
ThresholdSecrets.com

Threshold Modes in Elliptic Curves

Abstract

Threshold cryptography operation modes are described with application to the Ed25519, Ed448, X25519 and X448 Elliptic Curves. Threshold key generation allows generation of keypairs to be divided between two or more parties with verifiable security guaranties. Threshold decryption allows elliptic curve key agreement to be divided between two or more parties such that all the parties must co-operate to complete a private key agreement operation. The same primitives may be applied to improve resistance to side channel attacks. A Threshold signature scheme is described in a separate document.

https://mailarchive.ietf.org/arch/browse/cfrg/Discussion of this draft should take place on the CFRG mailing list (cfrg@irtf.org), which is archived at .

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Public key cryptography provides greater functionality than symmetric key cryptography by introducing separate keys for separate roles. Knowledge of the public encryption key does not provide the ability to decrypt. Knowledge of the public signature verification key does not provide the ability to sign. Threshold cryptography extends the scope of traditional public key cryptography with further separation of roles by splitting the private key. This allows greater control of (e.g.) decryption operations by requiring the use of two decryption key shares rather than just the decryption key.

This document describes threshold modes for decryption and key generation for the elliptic curves described in [RFC7748] and [RFC8032]. Both schemes are interchangeable in their own right but require minor modifications to the underlying elliptic curve systems to encode the necessary information in the public (X25519/X448) or private key (Ed25519/Ed448).

In its most general form, threshold cryptography allows a private key to be divided between (n, t) shares such that n is the total number of shares created and t is the threshold number of shares required to perform the operation. For most applications however, the number of shares is the same as the threshold (n = t) and the most common case is (n = t = 2).

This document sets out the principles that support the definition threshold modes in elliptic curve Diffie Hellman system first using simple additive key sharing, an approach which is limited to the case ($n = t$). The use of Shamir secret sharing [Shamir79] is then described to support the case ($n > t$). For convenience, we refer to the non Shamir secret sharing case as 'direct key sharing'.

1.1. Applications

Development of the threshold modes described in this document were motivated by the following applications.

1.1.1. Cloud control of decryption

The security of data at rest is of increasing concern in enterprises and for individual users. Transport layer security such as TLS and SSH provide effective confidentiality controls for data in motion but not for data at rest.

Of particular concern is the case in which a large store of confidential data is held on a server. Encryption provides a simple and effective means of protecting the confidentiality of such data. But the real challenge is how to effect decryption of the data on demand for the parties authorized to access it.

Storing the decryption keys on the server that holds the data provides no real security advantage as any compromise that affects the server is likely to result in disclosure of the keys. Use of end-to-end security in which each document is encrypted under the public key of each person authorized to access it provides the desired security but introduces a complex key management problem and provides no effective means of revoking access after it is granted.

Threshold encryption allows a key service to control decryption of stored data without having the ability to decrypt. The data decryption key is split into two (or more) parts with a different split being created for each user. One decryption share is held at the server allowing it to control access to the data without being able to decrypt. The other decryption share is encrypted under the public encryption key of the corresponding user allowing them to decrypt the stored data but only with the co-operation of the key service.

1.1.2. Device Onboarding

The term 'onboarding' is used to refer to the configuration of a device for use by a particular user or within a particular enterprise. One of the typical concerns of onboarding is to initialize the device with a set of public key pairs for various purposes and to issue credentials (e.g. PKIX certificates) to enable their use.

One of the concerns that arises in such cases is whether keys should be generated on the device on which they are to be used or on another device administering the onboarding process.

Both approaches are unsatisfactory. While generation of keys on the device on which they are to be used is generally preferred, experience has shown that many devices, particularly IoT devices use insufficiently random processes to generate such keys and this has led to numerous cases of duplicate and otherwise weak keys being discovered in running systems.

Generation of keys on an administration device and transferring them to the device on which they are to be used means that the administration device used for key generation represents a single point of failure. Compromise of this device or of the means used to install the keys will lead to compromise of the device.

Threshold key generation allows the advantages of both approaches to be realized avoiding dependence on either the target device or the administration device.

1.1.3. Cryptographic co-processor

Most real-world compromises of cryptographic security systems involve the disclosure of a private key. Common means of disclosure being inadvertent inclusion in backup tapes, keys being stored on public file shares and (increasingly) keys being inadvertently uploaded to source code management systems.

A new and emerging set of key disclosure threats come from the recent generation of hardware vulnerabilities being discovered in CPUs including ROWHAMMER and SPECTRE.

The advantages of Hardware Security Modules (HSMs) for storing and using private keys are well established. An HSM allows a private key to be used in an isolated environment that is designed to be resistant to side channel attacks.

Regrettably, the 'black box' nature of HSMs means that their use introduces a new security concern - the possibility that the device itself has been compromised during manufacture or in the supply chain.

Threshold approaches allows a key exchange or signature operation to be divided between two private keys, one of which is generated by the application that is to use it and the other of which is tightly bound to a specific host such that it cannot be extracted.

1.1.4. Side Channel Resistance

The same techniques that make threshold cryptography possible are the basis for Kocher's side-channel resistance technique [Kocher96]. Differential side channel attacks are a powerful tool capable of revealing a private key value that is used repeatedly in practically any algorithm. The claims made with respect to 'constant time' algorithms such as the Montgomery ladder depend upon the actual implementation performing operations in constant time.

2. Definitions

This section presents the related specifications and standard, the terms that are used as terms of art within the documents and the terms used as requirements language.

2.1. Requirements Language

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in [RFC2119].

2.2. Defined Terms

The following terms are used as terms of art in this document and the accompanying specification [draft-hallambaker-threshold-sigs].

Dealer A party that coordinates the actions of a group of participants performing a threshold operation.

Multi-Encryption The use of multiple decryption fields to allow a document encrypted under a session key to be decrypted by multiple parties under different decryption keys.

Multi-Encryption allows a document to be shared with multiple recipients but does not allow the decryption role to be divided between multiple parties.

Multi-Signatures The use of multiple independently verifiable digital signatures to authenticate a document.

Multi-Signatures allow separation of the signing roles and thus achieve a threshold capability. But they are not true threshold signatures as the set of signers is visible to external parties.

Onboarding The process by which an embedded device is provisioned for deployment in a local network.

Threshold Key Generation An aggregate public, private key pair is constructed from a set of contributions such that the private key is a function of the private key of all the contributions.

A Threshold Key Generation function is auditable if and only if the public component of the aggregate key can be computed from the public keys of the contributions alone.

Threshold Decryption Threshold decryption divides the decryption role between two or more parties.

Threshold Key Agreement A bilateral key agreement exchange in which one or both sides present multiple public keys and the key agreement value is a function of all of them.

This approach allows a party to present multiple credentials in a single exchange, a de

Threshold Signatures Threshold signatures divide the signature role between two or more parties in such a way that the parties and their roles is not visible to an external observer.

2.3. Related Specifications

This document extends the elliptic curve cryptography systems described in [RFC7748] and [RFC8032] to provide threshold capabilities.

This work was originally motivated by the requirements of the Mathematical Mesh [draft-hallambaker-mesh-architecture].

2.4. Implementation Status

The implementation status of the reference code base is described in the companion document [draft-hallambaker-mesh-developer].

3. Threshold Cryptography in Diffie-Hellman

The threshold modes described in this specification are made possible by the fact that Diffie Hellman key agreement and elliptic curve variants thereof support properties we call the Key Combination Law and the Result Combination Law.

Let {X, x}, {Y, y}, {A, a} be {public, private} key pairs and r [.] S represent the Diffie Hellman operation applying the private key r to the public key S.

The Key Combination law states that we can define an operator [x] such that there is a keypair {Z, z} such that:

Z = X [x] Y and $z = (x + y) \bmod o$ (where o is the order of the group)

The Result Combination Law states that we can define an operator [+] such that:

(x [.] A) [+] (y [.] A) = (z [.] A) = (a [.] Z)

It will be noted that each of these laws is interchangeable. The output of the key combination law to a Diffie Hellman key pair is a Diffie Hellman key pair and the output of the result combination law is a Diffie Hellman result. This allows modular and recursive application of these principles.

3.1. Application to Diffie Hellman (not normative)

Diffie Hellman in a modular field provides a concise demonstration of the key combination and result combination laws [RFC2631]. The realization of the threshold schemes in a modular field is outside the scope of this document.

For the Diffie Hellman system in a modular field p, with exponent e:

*r [.] S = S^r mod p

*o = p-1

*A [x] B = A [.] B = $AB \bmod p$.

Proof:

Let z = x + y

By definition, X = e^x mod p, Y = e^y mod p, and Z = e^z mod p.

Therefore,

Z = e^z mod p = e^{x+y} mod p

= $(e^x e^y)$ mod p

= e^x mod p.e^y mod p

$= X.Y$

Moreover, $A = e^a \bmod p$,

Therefore,

$(\mathbf{A^x \bmod p}).(\mathbf{A^y \bmod p}) = (A^xA^y) \bmod p)$

$= (A^{x+y}) \bmod p)$

$= A^z \bmod p$

$= e^{az} \bmod p$

$= (e^z)^a \bmod p$

$= Z^a \bmod p$

Since $e^o \bmod p = 1$, the same result holds for $z = (x + y) \bmod o$ since $e^{x+y+no} = e^{x+y}.e^{no} = e^{x+y}.1 = e^{x+y}$.

3.2. Threshold decryption

Threshold decryption allows a decryption key to be divided into two or more parts, allowing these roles to be assigned to different parties. This capability can be employed within a machine to divide use of a private key between an implementation running in the user mode and a process running in a privileged mode that is bound to the machine. Alternatively, threshold cryptography can be employed to

The key combination law and result combination law provide a basis for threshold decryption.

3.2.1. Direct Key Splitting

We begin by creating a base key pair { X, x }. The public component X is used to perform encryption operations by means of a key agreement against an ephemeral key in the usual fashion. The private component x may be used for decryption in the normal fashion or to provide the source material for a key splitting operation.

To split the base key into n shares { S_1, s_1 }, { S_2, s_2 }, ... { S_n, s_n }, we begin by generating the first n-1 private keys in the normal fashion. It is not necessary to generate the corresponding public keys as these are not required.

The private key of the final key share s_n is given by:

$s_n = (x - s1 - s2 - ... - sn\text{-}1) \bmod o$

Thus, the base private key x is equal to the sum of the private key shares modulo the group order.

3.2.2. Direct Decryption

To encrypt a document, we first generate an ephemeral key pair { Y, y }. The key agreement value e^{xy} is calculated from the base public key $X = e^x$ and the ephemeral private key y. A key derivation function is then used to obtain the session key to be used to encrypt the document under a symmetric cipher.

To decrypt a document using the threshold key shares, each key share holder first performs a Diffie Hellman operation using their private key on the ephemeral public key. The key shares are then combined using the result combination law to obtain the key exchange value from which the session key is recovered.

The key contribution c_i for the holder of the ith key share is calculated as:

$c_i = Y^{si}$

The key agreement value is thus

$A = c_1 . c_2 . ... . c_n$

This value is equal to the encryption key agreement value according to the group law.

3.3. Direct threshold key generation

The key combination law allows an aggregate private key to be derived from private key contributions provided by two or more parties such that the corresponding aggregate public key may be derived from the public keys corresponding to the contributions. The resulting key generation mechanism is thus, auditable and interoperable.

3.3.1. Device Provisioning

Auditable Threshold Key Generation provides a simple and efficient means of securely provisioning keys to devices. This is encountered in the IoT space as a concern in 'onboarding' and in the provisioning of unique keys for use with cryptographic applications (e.g. SSH, S/MIME, OpenPGP, etc.).

Consider the case in which Alice purchases an IoT connected Device D which requires a unique device key pair *{ X , x }* for its operation. The process of provisioning (aka 'onboarding') is performed using an administration device. Traditional key pair generation gives us three options:

* Generate and install a key pair during manufacture.
* Generate a new key pair during device provisioning.
* Generate a key pair on the administration device and transfer it to the device being provisioned.

The first approach has the obvious disadvantage that the manufacturer has knowledge of the private key. This represents a liability for both the user and the manufacturer. Less obvious is the fact that the second approach doesn't actually provide a solution unless the process of generating keys on the device is auditable. The third approach is auditable with respect to the device being provisioned but not with respect to the administration device being used for provisioning. If that device were to be compromised, so could every device it was used to provision.

Threshold key generation allows the administration device and the joining device being provisioned to jointly provision a key pair as follows:

* The joining device has public, private key pair *{ D, d }*.
* A provisioning request is made for the device which includes the joining device public key *D*.
* The administration device generates a key pair contribution *{ A, a }*.
* The administration device private key is transmitted to the Device by means of a secure channel.
* The joining device calculates the aggregate key pair *{ A [x] D, a+d }*.
* The administration device authorizes the joining device to participate in the local network using the public key *A [x] D*.

The Device key pair **MAY** be installed during manufacture or generated during provisioning or be derived from a combination of both using threshold key generation recursively. The provisioning request **MAY** be originated by the device or be generated by a purchasing system.

Note that the provisioning protocol does not require either party to authenticate the aggregate key pair. The protocol provides security by ensuring that both parties obtain the correct results if and only if the values each communicated to the other were correct.

Out of band authentication of the joining device public key *D* is sufficient to prevent Man-in-the-Middle attack. This may be achieved by means of a QR code printed on the device itself that discloses or provides a means of obtaining *D.*

[Note add serious warning that a party providing a private contribution **MUST** make sure they see the public key first. Otherwise a rogue key attack is possible. The Mesh protocols ensure that this is the case but other implementations may overlook this detail.]

3.3.2. Key Rollover

Traditional key rollover protocols in PKIX and other PKIs following the Kohnfelder model, require a multi-step interaction between the key holder and the Certificate Authority.

Threshold key generation allows a Certificate Authority to implement key rollover with a single communication:

Consider the case in which the service host has a base key pair { B , b }. A CA that has knowledge of the Host public key B may generate a certificate for the time period *t* with a fresh key as follows:

* Generate a key pair contribution { A_t, a_t }.
* Generate and sign an end entity certificate C_t for the key B [x] A_t.
* Transmit C_t, a_t to the host by means of a secure channel

3.3.3. Host Activation

Modern Internet service architectures frequently make use of short lived, ephemeral hosts running on virtualized machines. Provisioning cryptographic material in such environments is a significant challenge and especially so when the underlying hardware is a shared resource.

The key rollover approach described above provides a means of provisioning short lived credentials to ephemeral hosts that potentially avoids the need to build sensitive keys into the service image or configuration.

3.3.4. Separation of Duties

Threshold key generation provides a means of separating administration of cryptographic keys between individuals. This allows two or more administrators to control access to a private key without having the ability to use it themselves. This approach is of particular utility when used in combination with threshold decryption. Alice and Bob can be granted the ability to create key contributions allowing a user to decrypt information without having the ability to decrypt themselves.

3.4. Side Channel Resistance

Side-channel attacks, present a major concern in the implementation of public key cryptosystems. Of particular concern are the timing attacks identified by Paul Kocher [Kocher96] and related attacks in the power and emissions ranges. Performing repeated observations of the use of the same private key material provides an attacker with considerably greater opportunity to extract the private key material.

A simple but effective means of defeating such attacks is to split the private key value into two or more random shares for every private key operation and use the result combination law to recover the result.

The implementation of this approach is identical to that for Threshold Decryption except that instead of giving the key shares to different parties, they are kept by the party performing the private key operation.

While this approach doubles the number of private key operations required, the operations **MAY** be performed in parallel. Thus avoiding impact on the user experience.

4. Shamir Secret Sharing

The direct threshold modes described above may be extended to support the case ($n > t$) through application of Shamir secret sharing and the use of the Lagrange basis to recover the secret value.

Shamir Secret Sharing makes use of the fact that a polynomial of degree t-1 is defined by t points on the curve. To share a secret s, we construct a polynomial of degree $t-1$ in the modular field L where $L > s$.

$f(x) = s + a_1.x + a_2.x^2 + ... a_{t-1}.x^{t-1} \bmod L$

The shares p_1, p_2 ... p_n are given by the values $f(x_1)$, $f(x_2)$, ... , $f(x_n)$ where x_1, x_2 ... x_n are in the range 1 x_i L.

We can use the Lagrange basis function to construct a set of coefficients l_1, l_2, ... l_t from a set of t shares p_1, p_2 ... p_t such that:

$s = l_1p_1 + l_2p_2 + ... + l_tp_t \bmod L$

Thus, if we choose the sub-group order of the curve as the value of *L*, the formula used to recover the secret *s* is a sum of terms as was used for the case where $n = t$. We can thus apply a set of Lagrange coefficients provided by the dealer to the secret shares to extend the threshold operations to the case where $n > t$.

4.1. Shamir secret share generation

To create *n* shares for the secret *s* with a threshold of *t*, the dealer constructs a polynomial of degree *t* in the modular field *L*, where *L* is the order of the curve sub-group:

$f(x) = a_0 + a_1.x + a_2.x^2 + ... a_t.x^{t-1} \bmod L$

where $a_0 = s$

$a_1 ... a_t$ are randomly chosen integers in the range 1 a_i L

The values of the key shares are the values $f(x_1)$, $f(x_2)$, ... , $f(x_n)$. That is

$p_i = f(x_i)$

where $p_1 ... p_n$ are the private key shares

$x_1 ... x_n$ are distinct integers in the range 1 x_i L

The means of constructing distinct values $x_1 ... x_n$ is left to the implementation. It is not necessary for these values to be secret.

4.2. Lagrange basis recovery

Given *n* shares (x_0, y_0), (x_1, y_1), ... (x_{n-1}, y_{n-1}), The corresponding the Lagrange basis polynomials l_0, l_1, .. l_{n-1} are given by:

$l_m = ((x - x_{m0}) / (x_m - x_{m0})) . ((x - x_{m1}) / (x_m - x_{m1})) . ... . ((x - x_{mn-2}) / (x_m - x_{mn-}2))$

Where the values m_0, m_1, ... m_{n-2}, are the integers 0, 1, .. n-1, excluding the value *m*.

These can be used to compute *f(x)* as follows:

$f(x) = y_0 l0 + y1l1 + ... yn\text{-}1ln\text{-}1$

Since it is only the value of *f(0)* that we are interested in, we compute the Lagrange basis for the value $x = 0$:

$lz_m = ((x_{m1}) / (x_m - x_{m1})) . ((x_{m2}) / (x_m - x_{m2}))$

Hence,

$a_0 = f(0) = y_0 lz0 + y1lz1 + ... yn-1ln-1$

4.3. Verifiable Secret Sharing

The secret share generation mechanism described above allows a private key to be split into *n* shares such that *t* shares are required for recovery. While this supports a wide variety of applications, there are many cases in which it is desirable for generation of the private key to be distributed.

Feldman's Verifiable Secret Sharing (VSS) Scheme provides a mechanism that allows the participants in a distributed generation scheme to determine that their share has been correctly formed by means of a commitment.

To generate a commitment the dealer generates the polynomial $f(x)$ as before and in addition selects a generator g

[TBS]

4.4. Distributed Key Generation

[TBS]

5. Application to Elliptic Curves

For elliptic curve cryptosystems, the operators [x] and [.] are point addition.

Implementing a robust Key Co-Generation for the Elliptic Curve Cryptography schemes described in [RFC7748] and [RFC8032] requires some additional considerations to be addressed.

* The secret scalar used in the EdDSA algorithm is calculated from the private key using a digest function. It is therefore necessary to specify the Key Co-Generation mechanism by reference to operations on the secret scalar values rather than operations on the private keys.

* The Montgomery Ladder traditionally used to perform X25519 and X448 point multiplication does not require implementation of a function to add two arbitrary points. While the steps required to create such a function are fully constrained by [RFC7748], the means of performing point addition is not.

5.1. Implementation for Ed25519 and Ed448

[RFC8032] provides all the cryptographic operations required to perform threshold operations and corresponding public keys.

The secret scalars used in [RFC8032] private key operations are derived from a private key value using a cryptographic digest function. This encoding allows the inputs to a private key combination to be described but not the output. Contrawise, the encoding allows the inputs to a private key splitting operation to be described but not the output

It is therefore necessary to provide an alternative representation for the Ed25519 and Ed448 private keys. Moreover, the signature algorithm requires both a secret scalar and a prefix value as inputs.

Since threshold signatures are out of scope for this document and [RFC8032] does not specify a key agreement mechanism, it suffices to specify the data formats required to encode private key values generated by means of threshold key generation.

5.1.1. Ed25519

Let the inputs to the threshold key generation scheme be a set of 32 byte private key values P_1*, P2 ... Pn. For each private key value i in turn:*

0. Hash the 32-byte private key using SHA-512, storing the digest in a 64-octet large buffer, denoted h_i. Let n_i be the first 32 octets of h_i and m_i be the remaining 32 octets of h_i.

1. Prune n_i: The lowest three bits of the first octet are cleared, the highest bit of the last octet is cleared, and the second highest bit of the last octet is set.

2. Interpret the buffer as the little-endian integer, forming a secret scalar s_i.

The private key values are calculated as follows:

The aggregate secret scalar value s_a *= s1 + s2 + ... sn mod L, where L is the order of the curve.*

The aggregate prefix value is calculated by either

*Some function TBS on the values m_1, m_2, ... m_n, or

*Taking the SHA256 digest of s_a.

The second approach is the simplest and the most robust. It does however mean that the prefix is a function of the secret scalar rather than both being functions of the same seed.

5.1.2. Ed448

Let the inputs to the threshold key generation scheme be a set of 57 byte private key values P_1, *P2 ... Pn. For each private key value i in turn:*

0. Hash the 57-byte private key using SHAKE256(x, 114), storing the digest in a 114-octet large buffer, denoted h_i. Let n_i be the first 57 octets of h_i and m_i be the remaining 57 octets of h_i.

1. Prune n_i: The two least significant bits of the first octet are cleared, all eight bits the last octet are cleared, and the highest bit of the second to last octet is set.

2. Interpret the buffer as the little-endian integer, forming a secret scalar s_i.

The private key values are calculated as follows:

The aggregate secret scalar value s_a *= s1 + s2 + ... sn mod L, where L is the order of the curve.*

The aggregate prefix value is calculated by either

*Some function TBS on the values m_1, m_2, ... m_n, or

*Taking the SHAKE256(x, 57) digest of s_a.

The second approach is the simplest and the most robust. It does however mean that the prefix is a function of the secret scalar rather than both being functions of the same seed.

5.2. Implementation for X25519 and X448

[RFC7748] defines all the cryptographic operations required to perform threshold key generation and threshold decryption but does not describe how to implement them.

The Montgomery curve described in [RFC7748] allows for efficient scalar multiplication using arithmetic operations on a single coordinate. Point addition requires both coordinate values. It is thus necessary to provide an extended representation for point encoding and provide an algorithm for recovering both coordinates from a scalar multiplication operation and an algorithm for point addition.

The notation of [RFC7748] is followed using {u, v} to represent the coordinates on the Montgomery curve and {x, y} for coordinates on the corresponding Edwards curve.

5.2.1. Point Encoding

The relationship between the u and v coordinates is specified by the Montgomery Curve formula itself:

$v^2 = u^3 + Au2 + u$

An algorithm for extracting a square root of a number in a finite field is specified in . [RFC8032]

Since v^2 has a positive (v) and a negative solution ($-v$), it follows that v^2 mod p will have the solutions v, p-v. Furthermore, since p is odd, if v is odd, p-v must be even and vice versa. It is thus sufficient to record whether v is odd or even to enable recovery of the v coordinate from u.

5.2.2. X25519 Point Encoding

The extended point encoding allowing the v coordinate to be recovered is as given in [draft-ietf-lwig-curve-representations]

A curve point (u, v), with coordinates in the range 0 = u,v p, is coded as follows. First, encode the u-coordinate as a little-endian string of 57 octets. The final octet is always zero. To form the encoding of the point, copy the least significant bit of the v-coordinate to the most significant bit of the final octet.

5.2.3. X448 Point Encoding

The extended point encoding allowing the v coordinate to be recovered is as given in [draft-ietf-lwig-curve-representations]

A curve point (u, v), with coordinates in the range 0 = u,v p, is coded as follows. First, encode the u-coordinate as a little-endian string of 57 octets. The final octet is always zero. To form the encoding of the point, copy the least significant bit of the v-coordinate to the most significant bit of the final octet.

5.2.4. Point Addition

The point addition formula for the Montgomery curve is defined as follows:

Let $P_1 = \{u_1, v_1\}$, $P_2 = \{u_2, v_2\}$, $P_3 = \{u_3, v_3\} = P_1 + P_2$

By definition:

u_3

$= B(v_2 - v_1)^2 / (u_2 - u_1)^2 - A - u_1 - u_2$

$= B((u_2 v_1 - u_1 v_2)^2) / u_1 u_2 (u_2 - u_1)^2$

$v_3 = ((2u_1 + u_2 + A)(v_2 - v_1) / (u_2 - u_1)) - B (v_2 - v_1)^3 / (u_2 - u_1)^3 - v_1$

For curves X25519 and X448, B = 1 and so:

$u_3 = ((v_2 - v_1).(u_2 - u_1)^{-1})^2 - A - u_1 - u_2$

$v_3 = ((2u_1 + u_2 + A)(v_2 - v_1).(u_2 - u_1)^{-1}) - ((v_2 - v_1).(u_2 - u_1)^{-1})^3 - v_1$

This may be implemented using the following code:

```
B = v2 - v1
C = u2 - u1
CINV = C^(p - 2)
D = B * CINV
DD = D * D
DDD = DD * D

u3 = DD - A - u1 - u2
v3 = ((u1 + u1 + u2 + A) * B * CINV) - DDD - v1
```

Performing point addition thus requires that we have sufficient knowledge of the values v_1, v_2. At minimum whether one is odd and the other even or if both are the same.

5.2.5. Montgomery Ladder with Coordinate Recovery

As originally described, the Montgomery Ladder only provides the u coordinate as output. L?pez and Dahab [Lopez99] provided a formula for recovery of the v coordinate of the result for curves over binary fields. This result was then extended by Okeya and Sakurai [Okeya01] to prime field Montgomery curves such as X25519 and X448. The realization of this result described by Costello and Smith [Costello17] is applied here.

The scalar multiplication function specified in [RFC7748] takes as input the scalar value k and the coordinate u_1 of the point $P_1 = \{u_1, v_1\}$ to be multiplied. The return value in this case is u_2 where $P_2 = \{u_2, v_2\} = k.P_1$.

To recover the coordinate v_2 we require the values x_2, z_2, x_3, z_3 calculated in the penultimate step:

```
x_1 = u
x_2 = 1
z_2 = 0
x_3 = u
z_3 = 1
swap = 0

For t = bits-1 down to 0:
    k_t = (k >> t) & 1
    swap ^= k_t
    // Conditional swap as specified in RFC 7748
    (x_2, x_3) = cswap(swap, x_2, x_3)
    (z_2, z_3) = cswap(swap, z_2, z_3)
    swap = k_t

    A = x_2 + z_2
    AA = A^2
    B = x_2 - z_2
    BB = B^2
    E = AA - BB
    C = x_3 + z_3
    D = x_3 - z_3
    DA = D * A
    CB = C * B
    x_3 = (DA + CB)^2
    z_3 = x_1 * (DA - CB)^2
    x_2 = AA * BB
    z_2 = E * (AA + a24 * E)

(x_2, x_3) = cswap(swap, x_2, x_3)
(z_2, z_3) = cswap(swap, z_2, z_3)
Return x_2, z_2, x_3, z_3
```

The values x_2, z_2 give the projective form of the u coordinate of the point $P_2 = \{u_2, v_2\} = k.P_1$ and the values x_3, z_3 give the projective form of the u coordinate of the point $P_3 = \{u_3, v_3\} = (k+1).P_1 = P_1 + k.P_1 = P_1 + P_2$.

Given the coordinates $\{u_1, v_1\}$ of the point P1 and the u coordinates of the points P_2, $P_1 + P_2$, the coordinate v_2 **MAY** be recovered by trial and error as follows:

```
v_test = SQRT (u3 + Au2 + u)
u_test = ADD_X (u, v, u_2, v_test)
if (u_test == u_3)
   return u_test
else
   return u_test +p
```

Alternatively, the following **MAY** be used to recover {u_2, v_2} without the need to extract the square root and using a single modular exponentiation operation to convert from the projective coordinates used in the calculation. As with the Montgomery ladder algorithm above, the expression has been modified to be consistent with the approach used in [RFC7748] but any correct formula may be used.

```
x_p = u
y_p = v

B = x_p * z_2    //v1
C = x_2 + B      //v2
D = X_2 - B      //v3
DD = D^2         //v3
E = DD. X_3      //v3
F = 2 * A * z_2  //v1

G = C + F        //v2
H = x_p * x_2    //v4
I = H + z_2      //v4
J = G * I        //v2
K = F * z_2      //v1
L = J - K        //v2
M = L * z_3      //v2

yy_2 = M - E     //Y'
N = 2 * y_p      //v1
O = N * z_2      //v1
P = O * z_3      //v1
xx_2 = P * x_q   //X'
zz_2 = P * z_ q  //Z'

ZINV = (zz_2^(p - 2))
u2 = xx_2 * ZINV
v2 = yy_2 * ZINV

return u2, v2
```

6. Test Vectors

6.1. Threshold Key Generation

6.1.1. X25519

The key parameters of the first key contribution are:

```
X25519Key1 (X25519)
    UDF:            ZAAA-CTKG-X255-XXKE-YX
    Scalar:         32485880484394401977535777134446830499336694251
        10150162410603197194664525072
    Encoded Private
  10 BD E5 52  D6 AF 62 BE  E4 5B F3 30  B8 FC 1C 51
  B3 1B 10 9D  1E E9 D7 8D  04 23 39 08  55 5B D2 47
    U:              394710109806205653786046834135930103397328759791
        49327589691528787260753559967
    V:              356194396126197144337653540363972416928636805167
        03007342935403505222303874766
    Encoded Public
  9F C1 03 BF  A0 E6 6F C7  F1 98 4F 11  99 6E 35 E8
  E0 12 0A 0A  D0 0D 79 97  4E 8A 1C 08  EF CC 43 57
  00
```

The key parameters of the second key contribution are:

```
X25519Key2 (X25519)
    UDF:            ZAAA-CTKG-X255-XXKE-Y2
    Scalar:         411580799970764177643732652839689210097433579803
        82961631181135147895746569008
    Encoded Private
  30 A3 31 35  93 F6 AD C9  AC 13 1C 27  15 83 C8 1B
  00 EF 48 B9  52 14 8D 4D  3C F0 A3 C1  D2 A5 FE 5A
    U:              459625327459773177107083316486045330014029630693
        84886488866639114650950362503
    V:              257984254735960195350757294576288849253536777584
        92280923754924184339462178503
    Encoded Public
  87 E5 CC DD  1D AA 42 EA  6F E8 6F 70  71 EE CF 86
  45 52 48 50  9D B2 6A 76  3B 7A 21 A0  23 DF 9D 65
  80
```

The composite private key is:

```
Scalar_A = (Scalar_1 + Scalar_2) mod L
    =127390470814819760217717736698366165110586381169403573357222896
        2235868584190

Encoded Composite Private Key:

  FE 18 7D E6  61 C7 58 17  32 4F 63 FA  1A BD 2F 9C
  B2 0A 59 56  71 FD 64 DB  40 13 DD C9  27 01 D1 02
```

The composite public key is:

```
Point_A = Point_1 + Point_2

    U:              150135516006055775898023592854502286813423441552
        78788763949643404481150980325
    V:              415138854802975146804397164494309448482487190852
        96478905918078322266869330796

Encoded Public
  E5 10 7A CA  6D 63 5F 0B  96 8D C1 FF  03 88 6A 9F
  5E 39 FB C7  7D 4E 0C 8F  B9 BE 02 68  7B 5E 31 21
```

Note that in this case, the unsigned representation of the key is used as the composite key is intended for unsigned CurveX key agreement. If the result is intended for use as a key contribution, the signed representation is required.

6.1.2. X448

The key parameters of the first key contribution are:

```
X448Key1 (X448)
    UDF:            ZAAA-ETKG-X44X-KEYX
    Scalar:         481994585826426647968618231378947367458943870175
        360121942970840151506728554061197387112897908361553737348667
        886278259299907241847731316
    Encoded Private
  74 B4 D2 F1  12 CC E7 DD  F8 1A 30 80  1F 2C 19 EA
  EF E2 B3 8A  84 AF 60 11  0C 12 ED C3  B7 59 AE CC
  C9 B4 E4 9D  39 26 7C 61  5F 18 F1 24  FE 63 D6 4B
  BB 90 58 16  43 6E C3 A9
    U:              394700438080601820949554957897111201031639217153
        649316073420498416350673492846326387358441405527682364389676
        084099279128049619119543974
    V:              444540392869323915210590844990511294596507999517
        821659576820658489772434608466545900017841967298236181114781
        72629203404123869477697007
    Encoded Public
  A6 96 1A 77  DC 39 41 5F  D7 DA A5 07  45 AC 8E A4
  3E AE 8C 77  BD 50 4A B0  24 64 CD EA  58 0A A3 C7
  A7 80 BA A6  10 BD 57 9A  FA 0C E3 EB  2F C8 BB 52
  36 42 B2 58  C3 7B 04 8B  80
```

   The key parameters of the second key contribution are:

```
X448Key2 (X448)
    UDF:            ZAAA-ETKG-X44X-KEY2
    Scalar:         511754555230466033071173155845990436633003100190
        696628114856620555834938157360893794661792647517417178355617
        271188892590944570884738624
    Encoded Private
  40 CE 77 E2  F2 EC 9B 7D  3E F4 62 C6  F9 99 81 B4
  19 E5 4B 18  48 54 13 C9  79 D4 FF 3C  ED 3B 9C A1
  FE 10 7E DC  1F 56 BD 4D  27 7F 9C 70  4B 30 BE 0A
  86 2A 01 3D  2A C3 3E B4
    U:              150272226357947871808446816866811163361018047865
        977226556625643162666620657310046951510401422900774703631401
        927908349274354369426223715
    V:              366137359317179726859107301681665650104046728426
        432019544955478508004916807825041417161908471339553361130707
        683387754680120674193097983
    Encoded Public
  63 F2 0D 66  B0 F9 43 1C  58 AD 56 2B  C7 9A D5 83
  B0 B5 B1 73  9A BE B9 1E  72 5D 4A F7  8D 45 00 A6
  B3 7F AA 27  BE B4 72 44  EE D6 AA 24  5B BE B9 92
  F8 8D 63 CC  A1 6A ED 34  80
```

The composite private key is:

```
Scalar_A = (Scalar_1 + Scalar_2) mod L
    =852007356873840678531366273649321361498952695069091747059647117
        316116469037241626007959140974170910991528166120088403669830
        33434221045
```

```
Encoded Composite Private Key:

  F5 29 91 7B  28 EC 27 AA  8D 42 B7 81  DC F9 7A F7
  38 B7 D0 38  5C BB E9 04  F5 32 FA 90  A7 95 4A 6E
  C8 C5 62 7A  59 7C 39 AF  86 97 8D 95  49 94 94 56
  41 BB 59 53  6D 31 02 1E
```

The composite public key is:

```
Point_A = Point_1 + Point_2

    U:              992302561314692402358105531637482692313485564268
        0122693128385333032001352304577859762322215798453360214908 78
        1124991183380489710579618 7
    V:              456075089062293144799088747749397731190530292504
        2033435256956482434476720233184620803975661201956170497623 03
        280841894269351301464260878
```

```
Encoded Public
  5B DC 74 39  94 08 79 2C  D5 F0 F1 E0  5F 7F 87 4D
  4D 3B 92 96  AB 62 FF EC  CB 3C 74 42  48 D2 D0 30
  95 45 37 89  5E 53 5D 47  72 DD D8 1A  24 2C 65 76
  1F 7A FB 2E  15 2D F3 22
```

Note that in this case, the unsigned representation of the key is used as the composite key is intended for unsigned CurveX key agreement. If the result is intended for use as a key contribution, the signed representation is required.

6.1.3. Ed25519

The key parameters of the first key contribution are:

```
ED25519Key1 (ED25519)
    UDF:            ZAAA-GTKG-ED25-5XXK-EYX
    Scalar:         566707288876750955042011684517553468890396806254
        32020135051411766560411209648
    Encoded Private
  1D 04 C0 18  98 F0 31 CA  3B A1 F0 C3  AD 2B FC 3B
  CF F1 DC DC  07 FC 61 5F  B1 63 75 35  CE C4 EA B4
    X:              102242812372232316143929541945425934190366349837
        479493246432658075153118475011602747352787206173426322413798
        120490099709524898827760941579520013611119144
    Y:              789031640540855004178762236065881312872127689877
        983340977132004355159711145586361923666161872909051598545593
        272855769333002825303166428484255810207910288
    Encoded Public
  D8 4E 98 3E  F7 21 87 CC  C6 47 CB 6E  5A 57 1F 79
  F5 B9 22 D9  A8 00 D3 69  91 E1 B6 E6  10 FF 59 08
```

The key parameters of the second key contribution are:

```
ED25519Key2 (ED25519)
    UDF:            ZAAA-GTKG-ED25-5XXK-EY2
    Scalar:         566707288876750955042011684517553468890396806254
        32020135051411766560411209648
    Encoded Private
  1D 04 C0 18  98 F0 31 CA  3B A1 F0 C3  AD 2B FC 3B
  CF F1 DC DC  07 FC 61 5F  B1 63 75 35  CE C4 EA B4
    X:              102242812372232316143929541945425934190366349837
        479493246432658075153118475011602747352787206173426322413798
        120490099709524898827760941579520013611119144
    Y:              789031640540855004178762236065881312872127689877
        983340977132004355159711145586361923666161872909051598545593
        272855769333002825303166428484255810207910288
    Encoded Public
  D8 4E 98 3E  F7 21 87 CC  C6 47 CB 6E  5A 57 1F 79
  F5 B9 22 D9  A8 00 D3 69  91 E1 B6 E6  10 FF 59 08
```

The composite private key is:

```
Scalar_A = (Scalar_1 + Scalar_2) mod L
    =478637411536625779880453845786578016522261586016542618007355945
        8839008654461

Encoded Composite Private Key:

  7D 34 94 4B  39 80 D9 34  DB EB E1 7C  6C 7E BB FA
  77 03 B0 DC  59 BD DB 30  E9 EC 02 15  E3 FD 94 0A
```

The composite public key is:

```
Point_A = Point_1 + Point_2

    X:            156803891315589086286215914260988096641777056734
        949191292807298262090581551508321205906817723758419038014448
        221785557483058278412455441230622087321788265
    Y:            797171707400790781155409731019930644260359827339
        627382830182820599849038258772654275070450097319504001761725
        123426865448110684459259776487353068850001481

Encoded Public
  7B 1C 48 02  66 17 79 32  B3 02 7B 21  8E D8 FD 6C
  A1 D5 EC 8E  28 5D E8 D3  E2 08 1A F9  EB FA AC 32
```

6.1.4. Ed448

The key parameters of the first key contribution are:

```
ED448Key1 (ED448)
    UDF:            ZAAA-ITKG-ED44-XKEY-X
    Scalar:         627183439506120484725959826569914487299049496696
        595523009635802846606657882532934711603764178690431435900965
        134411043888412886366713268
    Encoded Private
  03 81 D6 5A  17 53 20 CB  F4 02 40 66  C8 28 A5 E7
  85 C6 87 7D  41 3A 33 CD  5B 09 9F E5  DF 1A 80 4D
  4F 6E B1 35  F7 46 8B 1C  46 99 6E 6D  A4 B5 23 EC
  FD 9C DF A4  9A E5 17 C6
    X:              583262368398068306064924227289526726945089383036
        870654148794281596473976374037255611400384372179423639836667
        405050261630569976923704901
    Y:              394808842674474575092248856523729981587605779158
        130628847253048358283876706462306075444313878262092034880682
        915931230840640589150609812
    Encoded Public
  BB 4B F3 61  B5 1F 95 1E  88 04 A8 5E  48 77 4C A5
  25 F0 D6 49  41 07 12 C1  15 05 25 78  60 FC D9 A8
  CC DB F9 6D  63 6B C3 F7  63 F2 BB 00  A0 7C 13 E1
  C6 4B 72 08  EA C8 87 11  00
```

The key parameters of the second key contribution are:

```
ED448Key2 (ED448)
    UDF:            ZAAA-ITKG-ED44-XKEY-2
    Scalar:         627183439506120484725959826569914487299049496696
        595523009635802846606657882532934711603764178690431435900965
        134411043888412886366713268
    Encoded Private
  03 81 D6 5A  17 53 20 CB  F4 02 40 66  C8 28 A5 E7
  85 C6 87 7D  41 3A 33 CD  5B 09 9F E5  DF 1A 80 4D
  4F 6E B1 35  F7 46 8B 1C  46 99 6E 6D  A4 B5 23 EC
  FD 9C DF A4  9A E5 17 C6
    X:              583262368398068306064924227289526726945089383036
        870654148794281596473976374037255611400384372179423639836667
        405050261630569976923704901
    Y:              394808842674474575092248856523729981587605779158
        130628847253048358283876706462306075444313878262092034880682
        915931230840640589150609812
    Encoded Public
  BB 4B F3 61  B5 1F 95 1E  88 04 A8 5E  48 77 4C A5
  25 F0 D6 49  41 07 12 C1  15 05 25 78  60 FC D9 A8
  CC DB F9 6D  63 6B C3 F7  63 F2 BB 00  A0 7C 13 E1
  C6 4B 72 08  EA C8 87 11  00
```

The composite private key is:

```
Scalar_A = (Scalar_1 + Scalar_2) mod L
    =164108792568830633627933941307822173067636952362213955597036306
        922337291995828355126032996607226607091940168014272113948183
        237575527862

Encoded Composite Private Key:

  B6 4D CF FC  C2 D8 08 4F  08 3C 03 5E  C5 68 95 D2
  65 F8 1D F2  C6 DC 4D 2E  23 B5 D5 02  AC 7A C5 A8
  77 E9 D0 6E  A2 1D 3D 11  91 49 AC 00  BF 0A 5B 95
  59 BD 0D 94  6E 00 CD 39
```

The composite public key is:

```
Point_A = Point_1 + Point_2

    X:            397684546797270425386442556916724605113037846866
        341266059847555740358632067326446186925590144430667066483990
        646036946928867316187394724
    Y:            408779958726533701154676738846429933260841277918
        126357328937237395286337382114316760480663335443832148985549
        99820255282659972439867001

Encoded Public
  39 DE 52 BB  22 79 72 DD  40 82 BB 45  7B FA 61 AD
  D0 BA 02 FF  8A FB 62 62  39 C6 C0 46  6B 35 93 E4
  CD 1B 5F 79  FB D7 68 87  4C E7 D8 09  B1 3C ED 5A
  DE 61 5A D2  DC 19 C6 D3  00
```

6.2. Threshold Decryption

6.2.1. Direct Key Splitting X25519

The encryption key pair is

```
X25519KeyA (X25519)
    UDF:          ZAAA-CTHD-X255-XXKE-YA
    Scalar:       326889380688366386055157824948497405399138729191
        89762489343300449845480879296
    Encoded Private
  C0 74 51 B1  0A 11 F3 AA  E9 E8 5C 99  A2 29 2F 78
  88 A8 FC 3D  09 69 06 60  C2 B4 95 71  85 48 45 48
    U:            298393977624490693814310584567182927861271505374
        9632477801696949179285366587
    V:            469514996865823666093464626932614325688577702009
        92824380735949162568370305003
    Encoded Public
  3B E7 D1 11  EA 09 02 81  C7 88 E9 59  7A 44 D1 D5
  34 AE 12 E2  3C 59 32 99  41 D1 99 B6  9D D9 98 06
```

To create n key shares we first create n-1 key pairs in the normal fashion. Since these key pairs are only used for decryption operations, it is not necessary to calculate the public components:

```
X25519Key1 (X25519)
    UDF:            ZAAA-CTHD-X255-XXKE-YX
    Scalar:         312348810422743662022326371802074910867528782383
        29365764492453739245942799272
    Encoded Private
  A8 FB C2 FD  62 20 CA 30  DA 44 87 38  BA 30 BE 5E
  FD 71 8F 48  33 E2 D2 9E  3F 28 AA C7  F0 50 0E 45
```

The secret scalar of the final key share is the secret scalar of the base key minus the sum of the secret scalars of the other shares modulo the group order:

```
Scalar_2 = (Scalar_A - Scalar_1) mod L
    =602777449245290709596292717071327769980982028247986740582014668
        2807789670656
```

```
This is encoded as a binary integer in little endian format:

  00 D1 65 C7  9A 18 2A 1B  11 47 27 BA  67 8B F5 2F
  85 1A 8C 86  3C 4B D9 FE  01 DD 3F 39  76 99 53 0D
```

6.2.2. Direct Decryption X25519

The means of encryption is unchanged. We begin by generating an ephemeral key pair:

```
X25519KeyE (X25519)
    UDF:            ZAAA-CTHD-X255-XXKE-YE
    Scalar:         493149316568141341116267515722816577238360781396
        63006284230452497981236822048
    Encoded Private
  20 C0 8B F4  BA DB D2 9A  69 47 45 73  4F 34 8E 35
  B5 78 24 AB  F6 85 29 51  37 0A CB 38  1E 43 07 6D
    U:              262957444446660172926134873812719769018477396438
        13954077731613449391499377029
    V:              275785345363973047453636076008194379259667671218
        204709190728344437212344040
    Encoded Public
  85 F9 AB 1E  1F 07 0F F9  9A 61 9F 3A  C8 34 C5 A2
  44 20 2A 92  7C 06 D8 54  E7 56 83 4F  2A DD 22 3A
```

The key agreement result is given by multiplying the public key of the encryption pair by the secret scalar of the ephemeral key to obtain the u-coordinate of the result.

```
 U:              2887877815485253696100618938370253328910292376544
     14471063569609087123262768472
```

The u-coordinate is encoded in the usual fashion (i.e. without specifying the sign of v).

```
58 85 FB 70  25 DB ED FB  F4 3F C2 11  65 A7 B6 FA
1B 2F 02 B7  36 34 A3 7B  F3 A0 2B 90  27 CF D8 3F
```

The first decryption contribution is generated from the secret scalar of the first key share and the public key of the ephemeral.

The outputs from the Montgomery Ladder are:

```
 x_2:            4920982389058477796216122916884880178090492296602
     94749204467459229896958710447
 z_2:            4295277578832279553752374567691230247398693682639
     13473544994514492134676680 91
 x_3:            1166052722679984495241928957075571148826293637988
     64825629417902239234188402 85
 z_3:            1957323558903499628199153847475120663027798992833
     56079998002895600508725693 34
```

The coordinates of the corresponding point are:

```
 u:              5367308943048082829305289884366605178672981287090
     47534335333617667296254533 82
 v:              4216320786006978219497487573828823097486360686160
     50214867098907422360742104 97
```

The encoding of this point specifies the u coordinate and the sign (oddness) of the v coordinate:

```
85 F9 AB 1E  1F 07 0F F9  9A 61 9F 3A  C8 34 C5 A2
44 20 2A 92  7C 06 D8 54  E7 56 83 4F  2A DD 22 3A
```

The second decryption contribution is generated from the secret scalar of the second key share and the public key of the ephemeral in the same way:

```
 u:             315780533103181510115520285714808517940341134033
    73677773373235419322267220550
 v:             182446295500312360012159151093139263903764865208
    52094490828880182704089261185
```

```
85 F9 AB 1E  1F 07 0F F9  9A 61 9F 3A  C8 34 C5 A2
44 20 2A 92  7C 06 D8 54  E7 56 83 4F  2A DD 22 3A
```

To obtain the key agreement value, we add the two decryption contributions:

```
 u:             400218389629441160369977262251603578789352483113
    44007312711045713754930418024
 v:             365543819982499491694061844647668737870082451479
    1236363503671566229693675370
```

This returns the same u coordinate value as before, allowing us to obtain the encoding of the key agreement value and decrypt the message.

6.2.3. Direct Key Splitting X448

The encryption key pair is

```
X448KeyA (X448)
    UDF:            ZAAA-ETHD-X44X-KEYA
    Scalar:         5130610704516282430400389743751911434509066008961
        18032807854320837117042898069268053810784334921351781007752
        474853979794048171909720757
    Encoded Private
  18 AB BD 69  F6 B7 16 23  72 4E B5 28  7E F8 F1 4E
  DB B5 6C EF  00 CD 51 4A  AD F6 24 AF  73 0B CC 37
  E4 66 01 C0  B4 35 18 99  CA 31 D0 7E  5D C6 86 9F
  4F 33 33 95  BB 90 B4 B4
    U:              19331338888441320215435003747749086893725402266O
        93636866811377797899223556623062585606490597078280448063871
        8961661755983597338734305565
    V:              63251206326449461009545592892127805892033877015
        23312210194943932254316016400401127656121982960128441713445
        59006776860014410518640582570
    Encoded Public
  1D 21 53 89  F7 D8 78 AD  F5 4F 66 AE  F6 E4 35 57
  A4 2D 0F 29  D7 ED 64 13  5A 15 5D 0C  5A 9D 78 8E
  30 AA D7 ED  94 D3 0A FD  5F C9 EB C4  6E 78 CB EC
  67 10 DE 1A  F7 41 16 44
```

To create n key shares we first create n-1 key pairs in the normal fashion. Since these key pairs are only used for decryption operations, it is not necessary to calculate the public components:

```
X448Key1 (X448)
    UDF:            ZAAA-ETHD-X44X-KEYX
    Scalar:         5847331912910601716145156574748319053529009968155
        38008733617256668598608739673264046230908386003505289747870
        06868637482183424893090556 4
    Encoded Private
  DC CD 9E 38  94 A1 EA CA  7D 41 FA B5  FB D3 01 43
  4A FB F9 1D  AE 73 82 3B  4C 11 A8 F2  2A 36 87 AB
  3F EB EE E8  DC AB A7 30  5E 26 9C BC  4C FC D0 C8
  48 F1 D3 78  A1 F0 F2 CD
```

The secret scalar of the final key share is the secret scalar of the base key minus the sum of the secret scalars of the other shares modulo the group order:

```
Scalar_2 = (Scalar_A - Scalar_1) mod L
    =753617529927807883056892001801624727334555668074124638992516626
        889301395112843028716421998506276731996363256267619893367343
        2870214466
```

```
This is encoded as a binary integer in little endian format:
```

```
  42 DB 4A 9E  1A CA 2C 19  F1 33 0E 8C  CA 3D 67 C9
  C4 69 61 F4  F4 1C FB EB  7E 30 10 B5  A1 41 53 E3
  23 52 F0 A8  91 E1 BF C9  28 58 6C 3B  AA C2 57 68
  98 24 07 0E  5D 81 A7 02
```

6.2.4. Direct Decryption X448

The means of encryption is unchanged. We begin by generating an ephemeral key pair:

```
X448KeyE (X448)
    UDF:            ZAAA-ETHD-X44X-KEYE
    Scalar:         375092101281685084138772040470324089521485504818
        151786170217259465369930000693812877111168523934570244151290
        043203531788355356164832452
    Encoded Private
  C4 3C 47 59  CD E7 17 95  B4 7B 93 AA  69 B8 B6 B7
  ED FE 18 D7  F4 7F 60 65  F1 89 C7 35  8D B5 43 37
  1B 7F 29 3E  C2 DE EF 30  B6 C6 B5 53  17 C5 53 34
  E1 86 A9 88  60 7B 1C 84
    U:              563581233819606639218664767140161390782939076635
        909204332105344559772692562144684584035704830171693935746990
        510749935485351255733775569
    V:              361864251157310605646771236064439317468860292821
        265311098908535377847605233578670208219924720551821972711559
        36724947885353521079579125
    Encoded Public
  D1 2C A9 6B  5E 97 F8 F0  18 2A BF 33  E8 14 65 23
  A9 F1 06 9B  D5 F0 DB 06  01 E5 1F 87  07 7D 69 63
  0A FD 05 FB  7A 65 4C D5  81 FC 63 11  5B D6 40 A1
  40 2F A5 FE  B3 C1 7F C6
```

The key agreement result is given by multiplying the public key of the encryption pair by the secret scalar of the ephemeral key to obtain the u-coordinate of the result.

```
    U:              467011616546325364170545331693527825162957894564
        752019329166269756954567903847269575078419988391203956276277
        502444715801151733990260662
```

The u-coordinate is encoded in the usual fashion (i.e. without specifying the sign of v).

```
B6 7F 79 43  2A 13 43 58  EB A5 F5 7E  0E 58 9B AA
BB D7 B1 7E  07 3E 42 F1  ED F4 C0 09  0C 5C 4E 88
C9 81 21 E5  31 53 40 2F  DE 7B 91 FE  E4 47 A2 A7
9B F8 E8 B0  AC 7A 7C A4  00
```

The first decryption contribution is generated from the secret scalar of the first key share and the public key of the ephemeral.

The outputs from the Montgomery Ladder are:

```
  x_2:            5065583485630979783123903130481133451680222127 11
      84934943515022463118399040333505232491239144312349970276 4248
      849278356671151441957790278
  z_2:            2131251809064129170646017094912422587817966802 09
      49698738627474492982070825757614708184082505313608222349 8470
      326539519004870996749115274
  x_3:            2034977525320085925292758468768893073928551300 50
      26658452525647612190782311421562636004801487502101626466 8899
      873822741730779444877210235
  z_3:            5710826122715861025754740479749799306267980156 01
      68073202599304857354201488134340888066971031506623204971 5159
      504508304710290054762104158
```

The coordinates of the corresponding point are:

```
  u:              6121129211606489390917112803975228316086493756 44
      85548462768557357479609757332494806024622185472869033216 8796
      528561330174015687677222868
  v:              2010434084114527683249927019519003811858696869 31
      75005610975799394781626263408814297959235561330883946921 9652
      899581533430916970635983432
```

The encoding of this point specifies the u coordinate and the sign (oddness) of the v coordinate:

```
D1 2C A9 6B  5E 97 F8 F0  18 2A BF 33  E8 14 65 23
A9 F1 06 9B  D5 F0 DB 06  01 E5 1F 87  07 7D 69 63
0A FD 05 FB  7A 65 4C D5  81 FC 63 11  5B D6 40 A1
40 2F A5 FE  B3 C1 7F C6
```

The second decryption contribution is generated from the secret scalar of the second key share and the public key of the ephemeral in the same way:

```
 u:             290543592387303668402000444540573819834556168336
     146133899156492196788195733788135568819938481333767701019721
     625502303783518942583538850
 v:             545084635544186338165791882421333945771369831404
     654834318355031722653832417877619017774120725462178582001609
     731098147709911715676858744
```

```
D1 2C A9 6B  5E 97 F8 F0  18 2A BF 33  E8 14 65 23
A9 F1 06 9B  D5 F0 DB 06  01 E5 1F 87  07 7D 69 63
0A FD 05 FB  7A 65 4C D5  81 FC 63 11  5B D6 40 A1
40 2F A5 FE  B3 C1 7F C6
```

To obtain the key agreement value, we add the two decryption contributions:

```
 u:             654406645663157043297342343331709836245150043735
     702788665901049252553535973169012625989449576860527067131717
     971914114816832035824532119
 v:             104146947548847304025795777575846818531726293762
     244429364569547834686063656820529540685030846449787127578128
     365084621097748733067553687
```

This returns the same u coordinate value as before, allowing us to obtain the encoding of the key agreement value and decrypt the message.

6.2.5. Shamir Secret Sharing X448

[TBS]

6.2.6. Lagrange Decryption X448

[TBS]

7. Security Considerations

All the security considerations of [RFC7748] and [RFC8032] apply and are hereby incorporated by reference.

7.1. Complacency Risk

Separation of duties can lead to a reduction in overall security through complacency and lack of oversight.

Consider the case in which a role that was performed by A alone is split into two roles B and C. If B and C each do their job with the same diligence as A did alone, risk should be reduced but if B and C each decide they can be careless because security is the responsibility of the other, the risk of a breach may be substantially increased.

It is therefore important that each of the participants in a threshold scheme perform their responsibilities with the same degree of diligence as if they were the sole control and for those responsible for oversight to treat single point failures that do not lead to an actual breach with the same degree of concern as if a breach had occurred.

Use of threshold operation modes mitigates but does not eliminate security considerations relating to private key operations of the underlying algorithm. For example, use of threshold key generation to generate a composite keypair {b+c, B+C} from key contributions {b, B} and {c, C} produces a strong composite private key if either of the key contributions *a*, *b* are strong. But the composite key will be weak if neither contribution is strong.

7.2. Rogue Key Attack

In general, threshold modes of operation provide a work factor that is at least as high as that of the work factor of the strongest private key share. The karmic tradeoff for this benefit is that the trustworthiness of a composite public key is that of the least trustworthy input.

For example, consider the case in which a client with keypair {c, C} generates an ephemeral keypair {e, E} for use in an authentication algorithm. We might decide to create an 'efficient' proof of knowledge of c and e by using the composite public key A = C+E to test for knowledge of both at the same time.

This approach fails because an attacker with knowledge of C can generate a keypair {a, A} and calculate the corresponding public key E = A-C. The attacker can then use the value a in the authentication protocol.

8. IANA Considerations

This document requires no IANA actions (yet).

It will be necessary to define additional code points for the signed version of the X25519 and X448 public key and the threshold decryption final private keys.

9. Acknowledgements

Rene Struik, Tony Arcieri, Scott Fluhrer, Scott Fluhrer, Dan Brown, Mike Hamburg

10. Appendix A: Calculating Lagrange coefficients

The following C# code calculates the Lagrange coefficients used to recover the secret from a set of shares.

[TBS]

11. Normative References

[draft-ietf-lwig-curve-representations]
Struik, R., "Alternative Elliptic Curve Representations", Work in Progress, Internet-Draft, draft-ietf-lwig-curve-representations-23, 21 January 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-lwig-curve-representations-23>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

12. Informative References

[Costello17] Costello, C. and B. Smith, "Montgomery curves and their arithmetic", 2017.

[draft-hallambaker-mesh-architecture]
Hallam-Baker, P., "Mathematical Mesh 3.0 Part I: Architecture Guide", Work in Progress, Internet-Draft, draft-hallambaker-mesh-architecture-19, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-architecture-19>.

[draft-hallambaker-mesh-developer]
Hallam-Baker, P., "Mathematical Mesh: Reference Implementation", Work in Progress, Internet-Draft, draft-hallambaker-mesh-developer-10, 27 July 2020, <https://datatracker.ietf.org/doc/html/draft-hallambaker-mesh-developer-10>.

[draft-hallambaker-threshold-sigs]
Hallam-Baker, P., "Threshold Signatures in Elliptic Curves", Work in Progress, Internet-Draft, draft-hallambaker-threshold-sigs-06, 13 January 2021, <https://datatracker.ietf.org/doc/html/draft-hallambaker-threshold-sigs-06>.

[Kocher96] Kocher, P. C., "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems.", 1996.

[Lopez99] L?opez, J. and R. Dahab, "Fast multiplication on elliptic curves over GF(2m) without precomputation.", 1999.

[Okeya01] Okeya, K. and K. Sakurai, "Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a Montgomeryform elliptic curve.", 2001.

[RFC2631] Rescorla, E., "Diffie-Hellman Key Agreement Method", RFC 2631, DOI 10.17487/RFC2631, June 1999, <https://www.rfc-editor.org/rfc/rfc2631>.

[Shamir79] Shamir, A., "How to share a secret.", 1979.