Examples of links

<norm="RFC822" section="2">

# Test <anchor="fred"/>

This is a test of the testy test <anchor="fred2"/>. It references <xref="moreFred"/>. It also has an external anchor to <eref="http://example.com/"/>;

# Test More <anchor="moreFred"/>

This references <xref="fred"/> and defines the term <iref>more</iref> It also has an external anchor to <a="http://example.com/"/> it references <xref="figure"/> and <xref="fred2"/>.

<figuresvg="aa.svg" id="figure"/>Alice's personal Mesh.

* r [.] S = Sr mod p
* o = p-1
* *A* [x] *B* = *A* [.] *B* = *AB* mod *p*.

*Proof:*

Let z = x + y

By definition, X = ex mod p, Y = ey mod p, and Z = ez mod p.

Therefore,

Z = ez mod p

= ex+y mod p

= (exey) mod p

= ex mod p.ey mod p

= X.Y

Moreover, A = ea mod p,

Therefore,

(Ax mod p).(Ay mod p)

= (AxAy) mod p)

= (Ax+y) mod p)

= Az mod p

= eaz mod p

= (ez)a mod p

= Za mod p

Since eo mod p = 1, the same result holds for z = (x + y) mod o since ex+y+no = ex+y.eno = ex+y.1 = ex+y.

## 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.

### 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 { S1, s1 }, { S2, s2 }, … { Sn, sn }, 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 sn is given by:

*sn = (x - s1 - s2 - … - sn-1)* mod *o*

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

### Decryption

To encrypt a document, we first generate an ephemeral key pair { Y, y }. The key agreement value exy is calculated from the base public key X = ex 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 ci for the holder of the ith key share is calculated as:

ci = Ysi

The key agreement value is thus

A = c1 . c2 . … . cn

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

## 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.

### 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.*

### 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 { At, at }.
* Generate and sign an end entity certificate Ct for the key B [x] At.
* Transmit Ct, at to the host by means of a secure channel

### 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.

### 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.

## 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 <info="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.

# 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 <norm="RFC7748"/> and <norm="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 <norm="RFC7748"/>, the means of performing point addition is not.

## Implementation for Ed25519 and Ed448

<norm="RFC8032"/> provides all the cryptographic operations required to perform threshold operations and corresponding public keys.

The secret scalars used in <norm="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 <norm="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.

### Ed25519

Let the inputs to the threshold key generation scheme be a set of 32 byte private key values *P1, P2 … Pn*. For each private key value *i* in turn:

1. Hash the 32-byte private key using SHA-512, storing the digest in a 64-octet large buffer, denoted *hi*. Let ni be the first 32 octets of hi and mi be the remaining 32 octets of hi.
2. Prune ni: 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.
3. Interpret the buffer as the little-endian integer, forming a secret scalar si.

The private key values are calculated as follows:

The aggregate secret scalar value *sa = 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 m1, m2, … mn, or
* Taking the SHA256 digest of sa.

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.

### Ed448

Let the inputs to the threshold key generation scheme be a set of 57 byte private key values *P1, P2 … Pn*. For each private key value *i* in turn:

1. Hash the 57-byte private key using SHAKE256(x, 114), storing the digest in a 114-octet large buffer, denoted *hi*. Let ni be the first 57 octets of hi and mi be the remaining 57 octets of hi.
2. Prune ni: 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.
3. Interpret the buffer as the little-endian integer, forming a secret scalar si.

The private key values are calculated as follows:

The aggregate secret scalar value *sa = 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 m1, m2, … mn, or
* Taking the SHAKE256(x, 57) digest of sa.

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.

## Implementation for X25519 and X448

<norm="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 <norm="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 <norm="RFC7748"/> is followed using {u, v} to represent the coordinates on the Montgomery curve and {x, y} for coordinates on the corresponding Edwards curve.

### Point Encoding

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

*v2* = *u3 + Au2 + u*

An algorithm for extracting a square root of a number in a finite field is specified in <norm="RFC8032">.

Since *v2* has a positive (*v*) and a negative solution (*-v*), it follows that *v2* 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*.

### X25519 Point Encoding

The extended point encoding allowing the v coordinate to be recovered is as given in <norm="draft-ietf-lwig-curve-representations"/>

[TBS]

### X448 Point Encoding

The extended point encoding allowing the v coordinate to be recovered is as given in <norm="draft-ietf-lwig-curve-representations"/>

[TBS]

### Point Addition

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

Let P1 = {u1, v1}, P2 = {u2, v2}, P3 = {u3, v3} = P1 + P2

By definition:

u3 = B(v2 - v1)2 / (u2 - u1)2 - A - u1 - u2

= B((u2v1 - u1v2)2 ) / u1u2 (u2 - u1)2

v3 = ((2u1 + u2 + A)(v2 - v1) / (u2 - u1)) - B (v2 - v1)3 / (u2 -u1)3 - v1

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

u3 = ((v2 - v1).(u2 - u1)-1)2 - A - u1 - u2

v3 = ((2u1 + u2 + A)(v2 - v1).(u2 - u1)-1) - ((v2 - v1).(u2 -u1)-1)3 - v1

This may be implemented using the following code: