Multi-Party Diffie-Hellman Key Exchange

Perry Kundert

2025-01-07 08:04:28

A critical requirement to support secure decentralized applications is the ability to compute cryptographic secrets amongst multiple parties. The Elliptic Curve Diffie-Hellman algorithm supports this, but it must be implemented within Holochain's `lair-keystore`, which stores and manages all cryptographic material for Holochain applications.

For example, to compute some common data between two or more Agents who only know the other's public keys, the shared data can be derived and used for encryption, but `lair-keystore`'s APIs do not currently allow it to be used to derive shared data for other uses such as independently configuring Holochain DNAs to use a shared DHT, such as required by apps like Volla Messages for two- or multi-party communications. It also prevents the implementation of many types of secure backup schemes involving off-line Ed25519 keypairs held in Crypto wallets or HSMs (Hardware Signing Modules).

In this paper we implement the examples in Wikipedia's Diffie-Hellman and Elliptic-curve Diffie-Hellman (ECDH) using the Python crypto-licensing module's Ed25519/X25519 implementation, and propose enhancements to `lair-keystore` to support Holochain applications using these constructs; including a robust method to ensure that multi-party Diffie-Hellman secret sharing does *not* leak the 2-party D-H secret between every pair of Agents – which is precisely one of the "Intermediate" value shared in the naive multi-party Diffie-Hellman computation! (PDF, Text)

Contents

1 Two-Party Shared Secrets **2**
1.1 Alice and Bob compute a Shared Secret via Diffie-Hellman 3
1.2 Verify Eve's Known Values 3

2 Three-Party Shared Secret Implementation **4**
2.1 Computing Intermediate Values and Shared Secret 4
2.2 What Does Eve Know? 5
2.3 Verify All Parties Have Same Secret 5
2.4 Generalizing to N Counterparies 6
2.5 Only Use Long-Term Keys for Two-Party Shared Secrets 7
2.6 Use Single-Purpose Keys for Multi-Party Shared Secrets 7

3 Implementation Using Ed25519 and X25519 Keypairs **9**
3.1 Computing N-Party Diffie-Hellman Shared Secret 14

4 Implementing in `lair-keystore` **16**
4.1 Computing Common Shared Data Using a Shared Secret 17
4.2 Revealing Intermediate Values for Multi-Party Shared Secrets 18
4.3 Implementing in Holochain 19
4.4 Additional Suggestions 19

1 Two-Party Shared Secrets

In this simple Diffie-Hellman example using integers, a public key is the primitive root g raised to the power of the private key, modulo the prime p:

$$\begin{aligned} A &= g^a \bmod p \\ B &= g^b \bmod p \end{aligned}$$

Where:

- p = prime modulus (public parameter)
- g = primitive root (public parameter); a number whose powers generate all non-zero values modulo p
- a, b = private keys (secret)
- A, B = public keys (derived from private keys, safe to share)

A shared secret calculation has a similar structure. By substitution, we can see that our private key plus the counterparty's public key can derive a common power of g – a "shared" secret value:

$$\begin{aligned} s_{bob} &= A^b \bmod p \\ &= (g^a)^b \\ &= g^{ab} \\ s_{alice} &= B^a \bmod p \\ &= (g^b)^a \\ &= g^{ab} \end{aligned}$$

This is adequate to investigate the properties of Diffie-Hellman; later, we will investigate how this translates to multi-party Diffie-Hellman using ECDH via Ed25519 and X25519 keypairs.

1.1 Alice and Bob compute a Shared Secret via Diffie-Hellman

```
def mod_exp(base, exp, modulus):
    """Calculate modular exponentiation efficiently"""
    result = 1
    base = base % modulus
    while exp > 0:
        if exp & 1:
            result = (result * base) % modulus
        base = (base * base) % modulus
        exp >>= 1
    return result

# Public parameters.  Small for demonstration, but cryptographically correct
g = 5   # primitive root
p = 23  # prime modulus

# Private keys
a = 6   # Alice's private key
b = 15  # Bob's private key

# Calculate public keys
A = mod_exp(g, a, p)  # Alice's public key
B = mod_exp(g, b, p)  # Bob's public key

# Calculate shared secret
s_alice = mod_exp(B, a, p)  # Alice's calculation
s_bob = mod_exp(A, b, p)    # Bob's calculation

[
    [ "Party","Private Key","Public Key", "Shared Secret" ],
    None,
    ["Alice", a, A, s_alice],
    ["Bob", b, B, s_bob],
]
```

Party	Private Key	Public Key	Shared Secret
Alice	6	8	2
Bob	15	19	2

1.2 Verify Eve's Known Values

What does Eve the eavesdropper know during this process?

Parameter	Value	Known to Eve?
g	5	Yes
p	23	Yes
$g^a = A$	8	Yes
$g^b = B$	19	Yes
a	6	No
b	15	No
$g^{ba} = s_{alice}$	2	No
$g^{ab} = s_{bob}$	2	No

2 Three-Party Shared Secret Implementation

For three-party DH, the structure of the intermediate shared secrets is basically the calculation and sharing of values computed by having each party apply their private key exponent to the public keys of each of their counterparies, and share this with the one remaining counterparty.

We can assume in many practical scenarios that each party has access to (or is provided with) the public keys of all desired counterparties.

- Public keys are well known, or
- Someone initiates the process by collecting all counterparties' private keys, and sends them to everyone involved.

However, in this example we'll demonstrate each party creating private keys a, b, c, and transmitting the corresponding public keys A, B, C and all intermediate values to all counterparties.

Let's demonstrates that:

- All parties arrive at the same shared secret
- Eve can see all intermediate values but can't compute the final secret
- The implementation follows the two basic principles for extending to larger groups:
  1. Starting with g and applying each participant's exponent once (ie. uses their public keys)
  2. Each participant applies their private key last to get the final secret

2.1 Computing Intermediate Values and Shared Secret

```
# Private keys
a = 6   # Alice's private key
b = 15  # Bob's private key
c = 13  # Carol's private key

# Calculate public keys (the initial intermediate values)

# Step 1: Alice distributes g^a (her public key, A) to Bob and Carol
A = g_a = mod_exp(g, a, p)
# Bob sends g^b (his public key, B) to Carol and Alice
B = g_b = mod_exp(g, b, p)
# Carol sends g^c to Alice and Bob
C = g_c = mod_exp(g, c, p)

# Step 2: Bob computes (g^a)^b = g^ab and sends to Carol
g_ab = mod_exp(g_a, b, p)
# Carol computes (g^b)^c = g^bc and sends to Alice
g_bc = mod_exp(g_b, c, p)
# Alice computes (g^c)^a = g^ca and sends to Bob
g_ca = mod_exp(g_c, a, p)

# Step 3: Carol computes (g^ab)^c = g^abc = final secret
s_carol = mod_exp(g_ab, c, p)
```

```
# Alice computes (g^bc)^a = g^bca = g^abc = final secret
s_alice = mod_exp(g_bc, a, p)
# Bob computes (g^ca)^b = g^cab = g^abc = final secret
s_bob = mod_exp(g_ca, b, p)

[
    ["Party", "Private Key", "Public Key", "Final Secret"],
    None,
    ["Alice", a, A, s_alice],
    ["Bob", b, B, s_bob],
    ["Carol", c, C, s_carol]
]
```

Party	Private Key	Public Key	Final Secret
Alice	6	8	4
Bob	15	19	4
Carol	13	21	4

2.2 What Does Eve Know?

Intermediate Value	Expression	Value	Known to Eve?
$g^a = A$	g^a mod p	8	Yes
$g^b = B$	g^b mod p	19	Yes
$g^c = C$	g^c mod p	21	Yes
$g^{ab} = s_{alice/bob}$	g^{ab} mod p	2	Yes
$g^{bc} = s_{bob/carol}$	g^{bc} mod p	7	Yes
$g^{ca} = s_{carol/alice}$	g^{ca} mod p	18	Yes
$g^{abc} = s_{alice/bob/carol}$	g^{abc} mod p	4	No

2.3 Verify All Parties Have Same Secret

```
assert s_alice == s_bob == s_carol, "Secrets don't match!"
[
    ["Verification", "Result"],
    None,
    ["All secrets match", "Yes"],
    ["Final shared secret", s_alice]
]
```

Verification	Result
All secrets match	Yes
Final shared secret	4

2.4 Generalizing to N Counterparies

This can extend to as many counterparties as we like. Let's verify this works with 4 parties by adding David (d).

The protocol extends naturally:

- Each party applies their exponent in turn
- The order doesn't matter (verified by calculating two different orders)
- The shared secret remains secure as long as private keys are kept secret

Key mathematical properties:

- The modular exponentiation is associative: $(g^a)^b \bmod p = g^(ab) \bmod p$
  - This allows different computation orders to reach the same final secret
  - The final secret will be $g^{abcd} \bmod p$ regardless of computation order

Security implications:

- Eve would see: $g^a, g^b, g^c, g^d, g^{ab}, g^{bc}, g^{cd}, g^{abc}$
  - But still cannot compute g^{abcd} without knowing at least one private key.

Adding more parties increases the number of visible intermediate values but maintains security *assuming none of the intermediate values are assumed to be secret in any other N-party shared secret computation*!

```
keys = {
    'a': 6,   # Alice
    'b': 15,  # Bob
    'c': 13,  # Carol
    'd': 17   # David (new)
}

# Calculate 4-party shared secret
# Order: Alice -> Bob -> Carol -> David
g_a = mod_exp(g, keys['a'], p)
g_ab = mod_exp(g_a, keys['b'], p)
g_abc = mod_exp(g_ab, keys['c'], p)
secret1 = mod_exp(g_abc, keys['d'], p)

# Alternative order: David -> Carol -> Bob -> Alice
g_d = mod_exp(g, keys['d'], p)
g_dc = mod_exp(g_d, keys['c'], p)
g_dcb = mod_exp(g_dc, keys['b'], p)
secret2 = mod_exp(g_dcb, keys['a'], p)

[
    ["Shared Secret Verification:"],
    None,
    [ "g^a = A", g_a ],
    [ "g^{ab}", g_ab ],
    [ "g^{abc}", g_abc ],
    [ "Secret via A->B->C->D", secret1],
    None,
    [ "g^d = D", g_d ],
    [ "g^{dc}", g_dc ],
    [ "g^{dcb}", g_dcb ],
    [ "Secret via D->C->B->A", secret2],
    None,
    [ "Secrets match:", secret1 == secret2],
]
```

Shared Secret Verification:	
g^{a} = A	8
g^{ab}	2
g^{abc}	4
Secret via A->B->C->D	2
g^{d} = D	15
g^{dc}	5
g^{dcb}	19
Secret via D->C->B->A	2
Secrets match:	True

Great! But there's an obvious problem… Haven't we seen $g^{ab} = 2$ and $g^{abc} = 4$ somewhere before, as the shared secret between Alice, Bob, and between Alice, Bob and Carol?

2.4.1 Shared Secret Exposure Risks!

You'll notice that the shared secret $s_{alice/bob} = g^{ab} = 2$ between Alice and Bob using their keypairs $A = g^a$ and $B = g^b$ is **exposed**, if these *same* keypairs are ever used to compute a shared secret between Alice, Bob and anyone else!

So how may we prevent this from ever happening?

2.5 Only Use Long-Term Keys for Two-Party Shared Secrets

The long-term (eg. Agent) keypairs are too useful for encrypting party-to-party communications to avoid using them. This public key is the well-known identity of the agent, and must be reserved for securing communications to and from Agents.

Any implementation must *prevent* the use of long-term keypairs for computing multi-party group secrets.

2.6 Use Single-Purpose Keys for Multi-Party Shared Secrets

When initiating multi-party group shared secret computation, the initiator (say, Alice) must produce a new "group" keypair private key x and public key $g^x = X$ to use as the basis of identifying the group (by the pubic key), and for securely computing the group shared secret.

By Alice sharing this group-specific public key $g^x = X$, *and* by also computing and sharing the first round of intermediate shared values to each counterparty:

$$g^x = X$$
$$g^{ax} = A^x$$
$$g^{bx} = B^x$$
$$g^{cx} = C^x$$

everyone can then proceed to compute their first round of intermediate shared secret values, just as for the three-party example. However, since all these intermediate values now depend on a group-unique private exponent x, no information is leaked that can affect any other group shared secret, nor any two-party shared secret.

This example demonstrates how Alice initiates the computation of a group shared secret with Bob and Carol using a group-specific keypair. Here's a breakdown of the process:

```
# Long-term private keys
a = 6  # Alice's private key
b = 15 # Bob's private key
c = 13 # Carol's private key

# Calculate/obtain public keys
A = mod_exp(g, a, p) # Alice's public key
B = mod_exp(g, b, p) # Bob's public key
C = mod_exp(g, c, p) # Carol's public key

# Alice generates a new group-specific private key
x = 19 # Alice's group-specific private key
X = mod_exp(g, x, p) # Alice's group-specific public key

# Alice computes and shares initial intermediate values with everyone for group X
g_ax = mod_exp(A, x, p)
g_bx = mod_exp(B, x, p)
g_cx = mod_exp(C, x, p)

# Each party computes their first round of intermediate shared secret values, and shares them with
# all other group X counterparties, ignoring any intermediate values containing their own exponent,
# and only sending to counterparties whose exponent is not already included in the value.  Note that
# Alice may receive a redundanct copy (g_cxb and g_bxc), so one can be ignored.
g_axb = mod_exp(g_ax, b, p) # Bob's computation, send to Carol
g_cxb = mod_exp(g_cx, b, p) # Bob's computation, send to Alice
g_axc = mod_exp(g_ax, c, p) # Carol's computation, send to Bob
g_bxc = mod_exp(g_bx, c, p) # Carol's computation, send to Alice

# Final shared secret computation
s_alice = mod_exp(g_cxb, a, p)
s_bob = mod_exp(g_axc, b, p)
s_carol = mod_exp(g_axb, c, p)
[
    ["Party", "Public Key", "Intermediate Values", "Final Secret"],
    None,
    ["Group-specific public key (X)", X, "", ""],
    None,
    ["Alice", A, (g_cxb, g_bxc), s_alice],
    ["Bob", B, g_axc, s_bob],
    ["Carol", C, g_axb, s_carol],
    None,
    ["Shared secrets match", "", "", s_alice == s_bob == s_carol]
]
```

Party	Public Key	Intermediate Values	Final Secret
Group-specific public key (X)	7		
Alice	8	(11 11)	9
Bob	19	16	9
Carol	21	3	9
Shared secrets match			True

3 Implementation Using Ed25519 and X25519 Keypairs

In `crypto-licensing`, we have a rudimentary pure-python implementation of Ed25519 and X25519 keypair operations. They're so small, I'll include them textually right here.

First, Ed25519 signatures:

```
#
# Edward's Reference Python-only Implementation of ed25519 signatures
#
# From https://ed25519.cr.yp.to/software.html
# Copyrights: The Ed25519 software is in the public domain.
#
# Minimal changes for Python2/3 compatibility by pjkundert
#

try:  # pragma nocover
    unicode
    PY3 = False
    def asbytes(b):
        """Convert array of integers to byte string"""
        return ''.join(chr(x) for x in b)
    def joinbytes(b):
        """Convert array of bytes to byte string"""
        return ''.join(b)
    def bit(h, i):
        """Return i'th bit of bytestring h"""
        return (ord(h[i//8]) >> (i%8)) & 1

except NameError:  # pragma nocover
    PY3 = True
    asbytes = bytes
    joinbytes = bytes
    def bit(h, i):
        return (h[i//8] >> (i%8)) & 1

import hashlib

b = 256
q = 2**255 - 19
l = 2**252 + 27742317777372353535851937790883648493

def H(m):
  return hashlib.sha512(m).digest()

def expmod(b,e,m):
  if e == 0: return 1
  t = expmod(b,e//2,m)**2 % m
  if e & 1: t = (t*b) % m
  return t

def inv(x):
  return expmod(x,q-2,q)

d = -121665 * inv(121666)
I = expmod(2,(q-1)//4,q)

def xrecover(y):
  xx = (y*y-1) * inv(d*y*y+1)
  x = expmod(xx,(q+3)//8,q)
  if (x*x - xx) % q != 0: x = (x*I) % q
  if x % 2 != 0: x = q-x
  return x

By = 4 * inv(5)
Bx = xrecover(By)
B = [Bx % q,By % q]

def edwards(P,Q):
  x1 = P[0]
  y1 = P[1]
  x2 = Q[0]
  y2 = Q[1]
  x3 = (x1*y2+x2*y1) * inv(1+d*x1*x2*y1*y2)
  y3 = (y1*y2+x1*x2) * inv(1-d*x1*x2*y1*y2)
  return [x3 % q,y3 % q]

def scalarmult(P,e):
  if e == 0: return [0,1]
  Q = scalarmult(P,e//2)
  Q = edwards(Q,Q)
  if e & 1: Q = edwards(Q,P)
  return Q

def encodeint(y):
  bits = [(y >> i) & 1 for i in range(b)]
  #return ''.join([chr(sum([bits[i * 8 + j] << j for j in range(8)])) for i in range(b//8)])
  return asbytes([     sum([bits[i * 8 + j] << j for j in range(8)])  for i in range(b//8)])

def encodepoint(P):
  x = P[0]
  y = P[1]
  bits = [(y >> i) & 1 for i in range(b - 1)] + [x & 1]
  #return ''.join([chr(sum([bits[i * 8 + j] << j for j in range(8)])) for i in range(b//8)])
  return asbytes([     sum([bits[i * 8 + j] << j for j in range(8)])  for i in range(b//8)])

def publickey(sk):
  h = H(sk)
  a = 2**(b-2) + sum(2**i * bit(h,i) for i in range(3,b-2))
  A = scalarmult(B,a)
  return encodepoint(A)

def Hint(m):
  h = H(m)
  return sum(2**i * bit(h,i) for i in range(2*b))
```

```
def signature(m,sk,pk):
  h = H(sk)
  a = 2**(b-2) + sum(2**i * bit(h,i) for i in range(3,b-2))
  #r = Hint(''.join([h[i] for i in range(b//8,b//4)]) + m)
  r = Hint(joinbytes( [h[i] for i in range(b//8,b//4)]) + m)
  R = scalarmult(B,r)
  S = (r + Hint(encodepoint(R) + pk + m) * a) % l
  return encodepoint(R) + encodeint(S)

def isoncurve(P):
  x = P[0]
  y = P[1]
  return (-x*x + y*y - 1 - d*x*x*y*y) % q == 0

def decodeint(s):
  return sum(2**i * bit(s,i) for i in range(0,b))

def decodepoint(s):
  y = sum(2**i * bit(s,i) for i in range(0,b-1))
  x = xrecover(y)
  if x & 1 != bit(s,b-1): x = q-x
  P = [x,y]
  if not isoncurve(P): raise Exception("decoding point that is not on curve")
  return P

def checkvalid(s,m,pk):
  if len(s) != b//4: raise Exception("signature length is wrong")
  if len(pk) != b//8: raise Exception("public-key length is wrong")
  R = decodepoint(s[0:b//8])
  A = decodepoint(pk)
  S = decodeint(s[b//8:b//4])
  h = Hint(encodepoint(R) + pk + m)
  if scalarmult(B,S) != edwards(R,scalarmult(A,h)):
    raise Exception("signature does not pass verification")


# Provide the ed25519ll API for compatibility

import warnings
import os

from collections import namedtuple

PUBLICKEYBYTES=32
SECRETKEYBYTES=64
SIGNATUREBYTES=64

Keypair = namedtuple('Keypair', ('vk', 'sk')) # verifying key, secret key

def crypto_sign_keypair(seed=None):
    """Return (verifying, secret) key from a given seed, or os.urandom(32)"""
    if seed is None:
        seed = os.urandom(PUBLICKEYBYTES)
    else:
        warnings.warn("ed25519ll should choose random seed.",
                      RuntimeWarning)
    if len(seed) != 32:
        raise ValueError("seed must be 32 random bytes or None.")
    # XXX should seed be constrained to be less than 2**255-19?
    skbytes = seed
    vkbytes = publickey(skbytes)
    return Keypair(vkbytes, skbytes+vkbytes)
```

```
def crypto_sign(msg, sk):
    """Return signature+message given message and secret key.
    The signature is the first SIGNATUREBYTES bytes of the return value.
    A copy of msg is in the remainder."""
    if len(sk) != SECRETKEYBYTES:
        raise ValueError("Bad signing key length %d" % len(sk))
    vkbytes = sk[PUBLICKEYBYTES:]
    skbytes = sk[:PUBLICKEYBYTES]
    sig = signature(msg, skbytes, vkbytes)
    return sig + msg


def crypto_sign_open(signed, vk):
    """Return message given signature+message and the verifying key."""
    if len(vk) != PUBLICKEYBYTES:
        raise ValueError("Bad verifying key length %d" % len(vk))
    checkvalid(signed[:SIGNATUREBYTES], signed[SIGNATUREBYTES:], vk) # raises Exception on failure
    return signed[SIGNATUREBYTES:]
```

Next, Curve25519 Diffie-Hellman:

```
import hashlib
import random

##########################################################
#
# Curve25519 reference implementation by Matthew Dempsky, from:
# http://cr.yp.to/highspeed/naclcrypto-20090310.pdf

P = 2 ** 255 - 19
A = 486662
G = 9

def expmod(b, e, m):
    if e == 0: return 1
    t = expmod(b, e // 2, m) ** 2 % m
    if e & 1: t = (t * b) % m
    return t

def inv(x):
    return expmod(x, P - 2, P)

def add(n, m, d):
    (xn, zn) = n
    (xm, zm) = m
    (xd, zd) = d
    x = 4 * (xm * xn - zm * zn) ** 2 * zd
    z = 4 * (xm * zn - zm * xn) ** 2 * xd
    return (x % P, z % P)

def double(n):
    (xn, zn) = n
    x = (xn ** 2 - zn ** 2) ** 2
    z = 4 * xn * zn * (xn ** 2 + A * xn * zn + zn ** 2)
    return (x % P, z % P)

def curve25519( n: int, base: int = G ) -> tuple[int, int]:
    one = (base,1)
    two = double(one)
    # f(m) evaluates to a tuple
    # containing the mth multiple and the
    # (m+1)th multiple of base.
    def f(m):
        if m == 1: return (one, two)
        (pm, pm1) = f(m // 2)
        if (m & 1):
            return (add(pm, pm1, one), double(pm1))
        return (double(pm), add(pm, pm1, one))
    ((x,z), _) = f(n)
    return (x * inv(z)) % P

def H( m: bytes ) -> bytes:
    return hashlib.sha512(m).digest()

def curve25519_key( n: int = 0 ) -> int:
    """An curve25519 key as an integer"""
    n = n or random.randint(0,P)
    n &= ~7
    n &= ~(128 << 8 * 31)
    n |= 64 << 8 * 31
    return n

def bytes_to_int(b: bytes) -> int:
    """Convert bytes to integer"""
    return sum( byte << (8 * i) for i,byte in enumerate( b ))

def int_to_bytes(n, length=32):
    """Convert integer to fixed-length bytes"""
    return bytes( (n >> (8 * i)) & 0xff for i in range( length ))

def ed25519_to_curve25519_key( sk: bytes ) -> int:
    """Converts a Ed25519 signing key as bytes to a curve25519 key as an integer"""
    return curve25519_key( bytes_to_int( H( sk )))

class ECDH:
    """Elliptic Curve Diffie-Hellman.

    Computes intermediate secrets for sharing, and the final shared_secret when an intermediate has
    been receive that includes all other desired parties' Ed25519 keys, using X25519 keys derived
    from the supplied Ed25519 key.

    """
    def __init__(self, ed25519_keypair, desired: set[ bytes ] ):
        """Initialize with provided Ed25519 private key bytes.  Add our public key to the desired
        set, which is assumed to contain all other parties' Ed25519 public keys participating in the
        X25519 shared secret calculation.

        """
        # Store original Ed25519 private key
        self.ed25519_public = ed25519_keypair.vk
        self.ed25519_private = ed25519_keypair.sk

        # Convert to X25519 private key and compute X25519 public key
        self.x25519_private = ed25519_to_curve25519_key( self.ed25519_private )
        self.x25519_public = curve25519( self.x25519_private )

        desired |= {self.ed25519_public}
        self.desired = desired
        self.shared_secret = None

    def initial_intermediate_value(self):
        """Initial intermediate value is private key . G, which is equivalent to the X25519 public key."""
        return self.x25519_public, {self.ed25519_public}
    def compute_intermediate_value(self, received_value: int, included: set[ int ]):
        """Compute intermediate value using X25519; add ours unless already included."""
        if self.x25519_public not in included:
            return curve25519(self.x25519_private, received_value), included | {self.ed25519_public}
        return received_value, included

    def compute_final_secret(self, received_value: int, included: set[ int ] ):
        """Compute final shared secret, from value with all other desired keys already included."""
        assert self.ed25519_public not in included and included | {self.ed25519_public} == self.desired, \
            f"Intermediate value is missing keys: {', '.join( vk.hex() for vk in (self.desired - included))}" \
            f"; should only have been missing: {self.ed25519_public.hex()}"
        self.shared_secret = curve25519(self.x25519_private, received_value)
        return self.shared_secret, self.desired
```

Let's demonstrate how these multi-party Diffie-Hellman operations translate.

3.1 Computing N-Party Diffie-Hellman Shared Secret

Let's compute a shared secret amongst 3 agents identified by Ed25519 keypairs. First, get some Ed25519 keypairs and identify all the agents involved:

```
# Step 1: Create participants with Ed25519 keys; all Ed25519 public keys are added to desired
desired         = set()
alice           = ECDH( crypto_sign_keypair(), desired )
bobby           = ECDH( crypto_sign_keypair(), desired )
carol           = ECDH( crypto_sign_keypair(), desired )

def shorten(s, n=16):
    s = str(s)
    if len(s) <= n:
        return s
    half = (n - 3) // 2
    return s[:half] + '...' + s[-half:]

# Step 2: Convert Ed25519 keys to X25519
[
    [ "Agent", "Ed25519 Pubkey"],
    None,
    [ "Alice", shorten( alice.ed25519_public.hex() )],
    [ "Bob",   shorten( bobby.ed25519_public.hex() )],
    [ "Carol", shorten( carol.ed25519_public.hex() )],
    None,
    [ "Agent", "X25519 Pubkey"],
    None,
    [ "Alice", shorten( alice.x25519_public ) ],
    [ "Bob",   shorten( bobby.x25519_public ) ],
    [ "Carol", shorten( carol.x25519_public ) ],
]
```

Now that we have everyone's Ed25519 private keys, we can compute the intermediate values between each 2 parties, and see that adding the missing parties' keys (in any order) leads to the same shared secret values for each party:

```
# Step 3: Alice -> Bob
ag, ag_includes     = alice.initial_intermediate_value()
assert ag_includes == {alice.ed25519_public}

# Step 4: Bob -> Carol
```

Agent	Ed25519 Pubkey
Alice	c98145…d7d043
Bob	51a1ee…f79034
Carol	baa762…9086c6
Agent	**X25519 Pubkey**
Alice	575250…684190
Bob	426081…997028
Carol	140187…798831

```
agb, agb_includes   = bobby.compute_intermediate_value( ag, ag_includes )
assert agb_includes == ag_includes | {bobby.ed25519_public}

# Step 5: Carol computes her secret
agbc, agbc_includes = carol.compute_final_secret( agb, agb_includes )
carol_secret        = agbc
assert agbc_includes == agb_includes | {carol.ed25519_public}

# Step 6: Bob -> Carol
bg, bg_includes     = bobby.initial_intermediate_value()

# Step 7: Carol -> Alice
bgc, bgc_includes   = carol.compute_intermediate_value( bg, bg_includes )

# Step 8: Alice computes her secret
bgca, bgca_includes = alice.compute_final_secret( bgc, bgc_includes )
alice_secret        = bgca

# Step 9: Carol -> Alice
cg, cg_includes     = carol.initial_intermediate_value()

# Step 10: Alice -> Bob
cga, cga_includes   = alice.compute_intermediate_value( cg, cg_includes )

# Step 11: Bob computes his secret
cgab, cgab_includes = bobby.compute_final_secret( cga, cga_includes )
bob_secret          = cgab

[
    [ "Intermediates", "Value" ],
    None,
    ["Alice -> Bob",          shorten( ag )],
    ["Alice -> Bob -> Carol", shorten( agb )],
    ["Bob -> Carol",          shorten( bg )],
    ["Bob -> Carol -> Alice", shorten( bgc )],
    ["Carol -> Alice",        shorten( cg )],
    ["Carol -> Alice -> Bob", shorten( cga )],
    None,
    [ "Agent", "Final Shared Secret" ],
    None,
    [ "Alice", shorten( alice_secret ) ],
    [ "Bob",   shorten( bob_secret ) ],
    [ "Carol", shorten( carol_secret) ],
    None,
    ["Shared secrets match", alice_secret == bob_secret == carol_secret],
]
```

Intermediates	Value
Alice -> Bob	575250. . . 684190
Alice -> Bob -> Carol	427842. . . 945640
Bob -> Carol	426081. . . 997028
Bob -> Carol -> Alice	101923. . . 528839
Carol -> Alice	140187. . . 798831
Carol -> Alice -> Bob	494067. . . 322529
Agent	Final Shared Secret
Alice	399766. . . 063231
Bob	399766. . . 063231
Carol	399766. . . 063231
Shared secrets match	True

3.1.1 Shared Secret Exposure Risks!

Just as with the simple D-H example, computing each 2-party Intermediate value for an N-party shared secret exposes the D-H shared secret used by those 2 parties!

So, we must ensure that our implementation **never** allows this.

We can accomplish this certainty by demanding that:

- exactly one participant must initiate the N-party D-H secret, *and*
- it must identify all the other (N-1) parties by public key, *and*
- it must include an *Ephemeral* (random) private key in the group!

By ensuring that every N-party D-H secret includes a random *Ephemeral* key (so N+1 Ed25519 keypairs), and the initiator always provides a starting Intermediate including both the *Ephemeral* key and *its own* key – *every* D-H shared secret including the same N parties will always be unique.

I suggest that we use the Public Key of *Ephemeral* keypair as the "identity" of the group D-H secret.

For example; if we use this functionality to establish a 5-party encrypted message group, the initiating party's *Ephemeral* public key would be the unique identifier for the group.

4 Implementing in `lair-keystore`

The current implementation of `lair-keystore` is missing a few features required to effectively utilize ECDH (Elliptic Curve Diffie-Hellman) for Two-Party shared secrets, and support for N-party shared secrets is missing entirely.

These capabilities could be implemented *outside* `lair-keystore` (eg. by using `ed25519-dalek` in the Zome's Rust code), but all keys would need to be generated and managed by the Zome code, losing access to the Agent ID private keys (which are never exposed by `lair-keystore`), and much of the valuable security due to the careful cryptographic secret handling provided by `lair-keystore` – it would be easy to bungle the handling of private keys in Zome code, and expose them unintentionally.

Therefore, I propose the following enhancements to `lair-keystore`:

4.1 Computing Common Shared Data Using a Shared Secret

Many situations involving Agent-to-Agent communications require some shared secret to be computed. This shared secret is computed internally by `lair-keystore` for the local Agents private key and any other Agent's public key.

Presently, arbitrary data can be *encrypted* using `LairApiReqCryptoBoxXSalsaBySignPubKey` etc., by one agent, and can be decrypted by the recipient Agent, which is valuable.

However, there is presently no way for two agents to use this shared secret to compute any other shared data – for example, for two agents to agree on a common Holochain DNA metadata value, so they can independently establish Holochain DNA instances that share the same DHT! Presently, the two Agents must come up with some external mechanism to communicate a common DNA metadata value with each-other, and then establish their DNA instances with the shared DHT.

4.1.1 Enhance `...CryptoBox...` APIs to Allow Optional `nonce`

There are 3 ways that ChaCha20Poly1305 may be safely used by two parties that have arrived at a common shared secret encryption key, with certain constraints:

- Hash some fixed known data with the shared secret, or use it directly as the cipher `key`
- Use 0 or some other shared data (eg. the xor or sort+hash of the two public keys) as `nonce`
- Encrypt known plaintext `data` (eg. zeros) of the desired output length to yield a deterministic shared value between the two Agents

Any of these approaches are valid (do not cryptographically reveal the shared secret) – *if* the `nonce` will never again be used with the same cipher `key` and different plaintext `data`!

It is recommended that some fixed data be hashed with the cipher `key` in this construction, so that if the `nonce` is accidentally reused with the same shared secret cipher `key` and different `data`, it only cryptographically compromises this one application's hashed shared secret – not the valuable single underlying Agent-to-Agent shared secret. Therefore, in the APIs allowing a user-supplied `nonce`, a mandatory SHA-256 hash of the shared secret cipher `key` with some user-supplied data (possibly empty) may be advisable. The goal is to provide a means to arrive at some common output ciphertext between two independent parties, depending only on their public keys, and this approach achieves that goal without risking the cryptographic integrity of the shared secret.

This enhancement is simple, and has limited risk – especially if some additional data is required to hash with the computed Diffie-Hellman shared secret when used as the cipher `key`.

4.2 Revealing Intermediate Values for Multi-Party Shared Secrets

For keypairs stored by `lair-keystore` to be used in computing multi-party shared secrets, at the very least we must implement the ability to provide a value to apply modular exponentiation by a keypair's secret key exponent, and return the result.

This is essentially the procedure for producing a public key from a private key: if the primitive root g is provided, and this function is called for a private key x, the public key X is returned.

If it is called with value of the public key $g^a = A$, using private key x, it would return the shared secret $(g^a)^x = g^{ax}$ derivable by holders of the private keys a and x.

Thus, misuse could easily leak the valuable shared secret used by communications between long-term keypairs of Agents, which `lair-keystore` strives to protect!

Furthermore, the creation of intermediate values during the calculation of shared secrets represent a set of private key exponents (identified by their public keys) in the value. Up until *all* counterparties are represented, multiplying by the private key exponent yields yet another intermediate value to be sent to some counterparty not yet represented in the value. This set of represented keys must be returned along with the intermediate value, and sent along so that the counterparties know the keys included in the value.

However, when all counterparties *are* included in the value, the final modular exponentiation with this Agent's private key exponent yields the **final** shared secret! This secret should be stored by `lair-keystore` encrypted at rest, and *not* returned – it must only be used for subsequent `...CryptoBox...` encryption operations, the same as for two-party shared secrets:

- The encryption of data, with a secure random `nonce`, or
- The production of deterministic shared data, with a user-supplied `nonce` and `data`.

4.2.1 Add `...GroupIntermediate...` APIs To Construct Intermediate Values

Receives a value and a set of Public Keys `represented` and `desired`, and the identity of a locally held private keypair (`ByTag`, `BySignPubKey`, etc.), and:

1. If adding private key exponent(s) found locally in set `desired - represented` doesn't satisfy all `desired` keys, return the value with the public key(s) added to `represented`; return an error if none found.

   The caller then forwards the value and `represented` set along to the appropriate counterparties as an intermediate value.

2. If all key(s) required to fulfill the `desired` are found, then apply them and store the shared secret and return a success indicator.

The caller may then use encryption and decryption operations as for any other computed shared secret, eg. `LairApiReqCryptoBoxXSalsaBySignPubKey`. However, the APIs would have to be enhanced to allow the identification of the shared secret by `desired` group, instead of by `sender_pub_key` and `recipient_pub_key`.

4.3 Implementing in Holochain

Additional APIs must be added to Holochain's `hdk` and `hdi` to allow construction and validation of intermediate values. Once implemented in `lair-keystore`, these should be quite simple.

4.4 Additional Suggestions

Remove the dependency on `libsodium` from `lair-keystore`, and instead use Rust-native encryption intrinsics.