# DualSalt, distributed signing and decryption for NaCl

Håkan Olsson

2019-02-18

## Abstract

Storing the secret key of key pairs is a security risk. There will never be a way to guarantee that a device is secure. The risk is heightened if the same key pair should be used on multiple devices. This is usually the case if the key pair should represent the identity of a person or a company. One solution is to split up the secret key into parts, but usually it needs to be recombined to do crypto operations. DualSalt is a library that lets you do 2 out of 2 signing and decryption without ever combining the secret key parts. The signature is compatible with EdDSA. The encrypt and decrypt algorithm is based on Ed25519 and the symmetric algorithm used in NaCl. It also features a rotate key functionality where both secret key parts are changed, but still represents the same public key.

## 1 Introduction

Curve25519 used NaCl has quickly become a major player in crypto [1][2]. It is a common choice for both distributed systems and IoT devices. One problem that arises is how to store the secret key if it represents something not associated with just one device e.g. a person or a company. These representations can have multiple devices or zero devices from time to time. To add one extra degree of freedom to solve this problem the secret key can be split up into multiple parts and stored in multiple devices. This without ever having to recombine the key to do decryption and signing. The Schnorr signatures used in EdDSA are so called MPC-friendly and excellent for collective operations compared to ECDSA. The algorithm supports this, but there is no known recognised software library exposing this functionality[1]. DualSalt is an attempt to solve this. DualSalt is open source and can be found at: https://github.com/assaabloy-ppi/dualsalt

Note: Some other attempts exist[4], but they have stagnated and there is still no library out there, until now.

## 1.1 Foundation

The DualSalt implementation builds upon a TweetNaCl [7] and all low level functionality is taken from that implementation. TweetNaCl is implemented in multiple languages which makes porting DualSalt to other languages easier since only the top layer needs to be ported. The reference implementation for DualSalt is written in Java and uses InstantWebP2P/tweetnacl-java.

## 1.2 Functions

The interface to DualSalt should be complete. Create keys, add public key parts, rotate added key parts, create signature, verify signature, encrypt and decrypt.

| |
|---|
| createKeyPair |
| createKeyPart |
| addPublicKeyParts |
| rotateKeyPart |
| signCreate |
| signCreateDual1 |
| signCreateDual2 |
| signCreateDual3 |
| signVerify |
| encrypt |
| decrypt |
| decryptDual1 |
| decryptDual2 |

Table 1: DualSalt interface functions

[1]During 2018 while this report was written a promising library has been released [10], it is GPL

## 1.3 Dual operations flow

DualSalt can be used in many ways but Figure 1 illustrate the case where one of the two devices act as master.

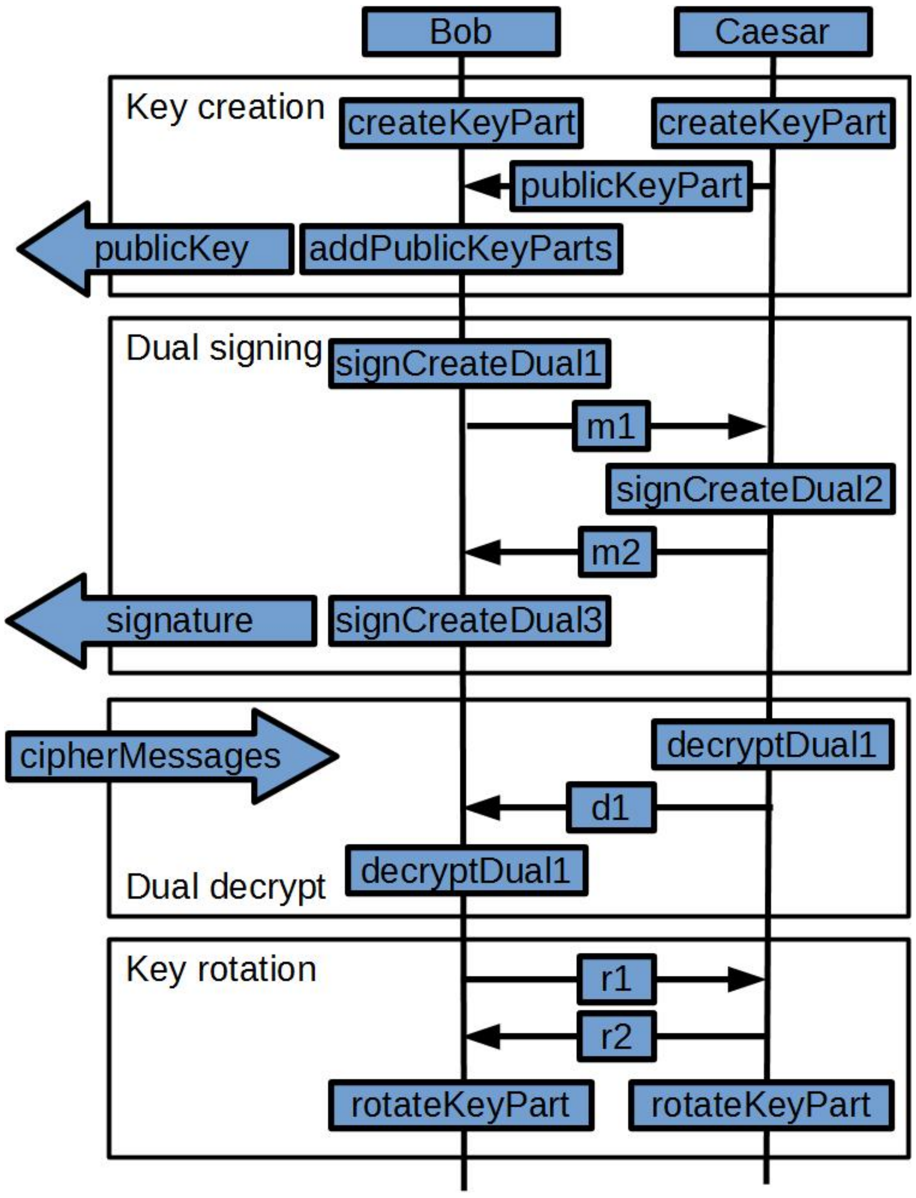

Figure 1: Illustration of typical dual operations with DualSalt.

## 1.4 Security requirements

This section describes the design goals for DualSalt and the limitations.

### 1.4.1 Priorities

The following priorities were used when designing the protocol.

1. Achieve high security.
2. Achieve a low network overhead; that is, few round-trips and a small data overhead.
3. Allow for low code complexity, and low CPU requirements of the communicating peers. Low complexity is in itself, important to achieve high security.

### 1.4.2 Goals

The main goals of the protocol are the following:

1. 128-bit security. The best attack should take at least $2^{128}$ operations to complete. No attack should be feasible until there are (if there ever will be) large-enough quantum computers.
2. Bob and Caesar can do 2 out of 2 signing and decryption.
3. Bob and Caesar can not retrieve the combined secret through use or manipulation of the Dual Salt protocol.
4. Bob and Caesar should be able to be perform key part rotation. There secrets is refreshed but Bob and Caesar still can do dual crypto operations the previous common public key.
5. For an attacker to be successful both Bob and Caesar have to be compromised in the same time window, defined by key part rotations.
6. Signatures are compatible with EdDSA ED25519.
7. Simple protocol. It should be possible to implement in a few lines of code. Should be auditable just like TweetNaCl.
8. Compact messages (few bytes). Designed for Bluetooth Low Energy and other low-bandwidth channels.

### 1.4.3 Limitations

DualSalt is limited in scope in the following ways:

1. Messages between Bob and Caesar should be sent authenticated and encrypted with forward secrecy. E.g. SaltChannel [8].
2. No attempt is made by DualSalt to check what is signed. That functionality could be added on top of DualSalt.
3. No quantum security. The protocol is not intended to be secure for an attacker with a quantum computer with a large number of qubits. Such a computer does not exist. This is a limitation from the underlying TweetNaCl library.

## 1.5 Complexity

NaCl has the nice property that it does not contain many options. This makes it easy to use. It also has a small footprint. DualSalt try to do the same.

## 1.6 Example use case

One use case for DualSalt is unsecure web application. A web page in a browser and web server. E.g. Protonmail [9]. The user do not want the Protonmail server to be able to sign and decrypt messages on the users behalf. The Protonmail team neither want to be able to sign nor decrypt messages on the users behalf. If the Protonmail servers is hacked the risk is that and a large number keys are leaked. The solution in Protonmail today is to recreate the secret key in the web browser with the help of username and password. This solves the security issues above but what if the web browser can not be trusted ether? One solution is to use DualSalt and recreate a secret key part in the web server and storing another key part in the Protonmail server. The two parts together can sign and decrypt for the user. This does not solve all problems. However even after a total compromise of the web browser the attacker can still be blocked out by refreshing the password and rotating the key parts.

## 2 Keys

The key parts used for dual operations are stored in different format than the key pairs used for single operations. Single operations is the classic signing and decryption done by one party. The single key pairs are identical with TweetNaCl. Key parts could have been used as key pairs for single operations too, but the design decision was taken to use the original TweetNaCl key pairs instead. The reason for this is mostly compatibility, but also to prevent reuse of the same key for both single and dual operations.

**Definition table:**

| |
|---|
| $a_s$ = seed of secret key for Alice |
| $a$ = scalar piece of the secret key of Alice |
| $a_r$ = random piece of the secret key of Alice |
| $A$ = public key of Alice$(aP)$ |
| $b, c$ = scalar pieces of the secret key parts of Bob and Caesar |
| $b_r, c_r$ = random pieces of the secret key parts of Bob and Caesar |
| $B, C$ = public key parts of Bob and Caesar $(bP, cP)$ |
| $D$ = Dual public key of Bob and Caesar |
| $P$ = base point |

### 2.1 Standard EdDSA key pair

The key format used in single operations

$$a_s = random(32)$$
$$d = sha512(a_s) = (d_0, d_1, ..., d_{510}, d_{511})$$
$$a = 2^{254} + \sum_{i=3}^{253} 2^i d_i \in \{2^{254} + 0, 2^{254} + 8, ..., 2^{254} + 8 \cdot 2^{251}\}$$

The expression above masks the scalar $a$ in this specific way[5]:

1. **Clears 3 lowest bits** to prevent sub group attacks. A sub group attack is the attempts of an attacker to trick the attacked into operating on group element in a subgroup of the desired group.
2. **Clears the highest bit** to skip modulus to make scalar fit in group
3. **Sets the second highest bit** to mitigate weakness in some scalar multiplication implementations

The masking in code:

```
a=d[0,32];
a[0]&=248;
a[31]&=127;
a[31]|=64;
```

| createKeyPair($a_s$) |
|---|
| $d = sha512(a_s)$ <br> $a = 2^{254} + \sum_{i=3}^{253} 2^i d_i$ <br> $A = aP$ |
| $(a_s, A)$, $A$ |

The secret key has the public key included $(a_s, A)$ which gives a 64 byte array compatible with TweetNaCl. $A$ is the public key.

## 2.2 Key Part

Key parts are almost identical to TweetNaCl key pairs, but with some differences required by a key rotation. The key rotation makes it impossible to only store the pre hash format as is the case with TweetNaCl keys. Also the second top bit is not guaranteed to remain set through a key rotation. Because this bit is not set, the scalar multiplication used has to be constant time even if said bit is not set anymore. Fortunately the scalar multiplication in TweetNaCl has this property. Another effect that occurs is that the 3 lowest bits that are cleared in a TweetNaCl key will not stay cleared after a key rotation. This does not affect signing, but it opens up for small order attacks while decrypting. This is avoided by having group checks on incoming group elements while decrypting. As a consequence of these two effects, only the highest bit is cleared.

$$d = sha512(random(32))$$
$$b = \sum_{i=0}^{255-1} 2^i d_i$$
$$b_r = \sum_{i=256}^{511} 2^{i-256} d_i$$
$$B = bP$$

The secret key part is saved in the format $(b, b_r, B)$ and makes up a 96 byte array.

### 2.2.1 Public key part addition

Adding public key parts together introduces a built-in risk. If Caesar gets hold of $B$ before Caesar has presented its own public key part $C$. Caesar can trick the device that adds the public key parts to eliminate the key part from Bob.

$$B = bP$$

$$C = cP$$

$$C' = C - B$$

$$D = B + C' = B + C - B = C = cP$$

To get around this DualSalt requires each public key part used in a public key part addition to have a signature with the public key part signed by its secret key part. This proves that the device has the corresponding secret key part. It is impossible for $c$ to sign with $c' = c - b$ without knowing the other party's secret key part $b$.

See EdDSA in section 3.1. Use the public key as message. Add the resulting $(R_B, s_B)$ and the public key $B$ to get the public key part $(R_B, s_B, B)$. This is the parameter later used in *addPublicKeyParts*() This is also a valid EdDSA signature

Before the key addition in *addPublicKeyParts*() the two incoming public key parts have to be validated with $signVerify()$. If the two public key parts are correct, add them. $D = B + C$

| createKeyPart($b_s$) |
|---|
| $d = sha512(b_s)$<br>$b = \sum_{i=0}^{254} 2^i d_i$<br>$b_r = \sum_{i=256}^{511} 2^{i-256} d_i$<br>$B = bP$<br>$r = sha512(b_r \vert\vert B)$<br>$R_B = rP$<br>$h = sha512(R_B \vert\vert B \vert\vert B)$<br>$s_B = r + bh$ |
| $(b, b_r, B)$, $(R_B, s_B, B)$ |

| addPublicKeyParts$((R_B, s_B, B),\ (R_C, s_C, C))$ |
|---|
| $signVerify((R_B, s_B, B), B))$<br>$signVerify((R_C, s_C, C), C))$<br>$D = B + C$ |
| $D$ |

### 2.2.2 Key rotation

The secret key parts can be rotated in DualSalt. This is when both secret key parts are refreshed but they together still can do operations for the same common public key.

$$r = random(32)$$

$$(r_s, r_r) = sha512(r)$$

$$D = bP + cP$$

$$b' = b + r_s$$

$$c' = c - r_s$$

$$D == b'P + c'P = bP + cP + (r_s - r_s)P$$

$r$ should be made using random from both Bob and Caesar. Alternatively the functions can be run twice, once with random from Bob and once with random from Caesar. The same function is used by both Bob and Caesar and a boolean parameter $f$ tells the function to make scalar addition or subtraction.

$b_r$ and $c_r$ has to be updated. Just updating with a new random value could be done here. No connection to the random used in the other device is needed. No connection to the random used to update the scalar is needed. But to keep the use of random down and to get a simpler interface DualSalt uses pseudo random. The main reason to keep the use of random down is to avoid attacks that exploits bad random generations.

```
rotateKeyPart($(b, b_r, B)$, $r$, $f$)
-----
$d = sha512(r)$
$r_s = \sum_{i=0}^{254} 2^i d_i$
$r_r = \sum_{i=256}^{511} 2^{i-256} d_i$
$if(f)\{$
        $b' = b + r_s$
$\}else\{$
        $b' = b - r_s$
$\}$
$B' = b'P$
$z = sha512(b_r || r_r)$
$b'_r = \sum_{i=0}^{255} 2^i z_i$
-----
$(b', b'_r, B')$
```

## 3 Signing

The signatures produced by DualSalt are compatible with Ed25519. Signatures can be created with a key pair or with two key parts after the combined public key has been created.

**Definition table:**

| |
|---|
| $m$ = message to sign |
| $r$ = random used in the signature |
| $R$ = corresponding group element to $r$ $(rP)$ |
| $m1$ = first message in a dual sign transaction |
| $m2$ = second message in a dual sign transaction |
| $n_B, n_C$ = signing nonce for Bob and Caesar |
| $(R, s, m)$ = signature |

## 3.1 Standard EdDSA

Standard EdDSA signing and verification is included in the DualSalt library. For the implementation it is recommended to do fall-through functions that uses the corresponding TweetNaCl functions. Single sign

| signCreate$((a_s, A),\ m)$ |
|---|
| $d = sha512(a_s)$ |
| $a = 2^{254} + \sum_{i=3}^{253} 2^i d_i$ |
| $a_r = \sum_{i=256}^{511} 2^{i-256} d_i$ |
| $r = sha512(a_r \|\| m)$ |
| $R = rP$ |
| $h = sha512(R \|\| A \|\| m)$ |
| $s = r + ah$ |
| $(R, s, m)$ |

Validation

| signVerify$((R, s, m),\ A)$ |
|---|
| $h = sha512(R \|\| A \|\| m)$ |
| $R == sP + h(-A)$ |
| true/false |

$$s = r + ha$$

If $s$ is defined as above then the corresponding expression in the group below has to be true.

$$sP == R + hA$$

$$R == sP - hA$$

$$R == sP + h(-A)$$

## 3.2 Dual Sign

The dual signing is implemented in three functions. To create a signature only takes one round trip.

Device Bob　　　　　　　　　　　　　　　　　　　　Device Caesar

| signCreateDual1$(m,\ (b, b_r, B),\ D,\ n_A)\ )$ |
|---|
| $r_B = sha512(b_r \|\| m \|\| n_B)$ |
| $R_B = r_B P$ |
| $m1 = (D, R_B, m)$ |

$$\overrightarrow{\text{send } m1}$$

| signCreateDual2($m1$, $(c, c_r, C)$, $n_C$) |
|---|
| $r_C = sha512(c_r \vert\vert m \vert\vert n_C)$ |
| $R_C = r_C P$ |
| $R = R_B + R_C$ |
| $h = sha512(R \vert\vert D \vert\vert m)$ |
| $s_C = r_C + hc$ |
| $m2 = (R_C, s_C)$ |

$$\overleftarrow{\text{send } m2}$$

| signCreateDual3($m1$, $m2$, $(b, b_r, B)$, $n_B$) |
|---|
| $r_B = sha512(b_r \vert\vert m \vert\vert n_B)$ |
| $R = R_B + R_C$ |
| $h = sha512(R \vert\vert D \vert\vert m)$ |
| $C = D - B$ |
| $R == sP + h(-C)$ |
| $s_B = r_B + hb$ |
| $s = s_B + s_C$ |
| $(R, s, m)$ |

The validation for ordinary EdDSA can validate a dual signature as proven below:

$$sP == R + h(D)$$

$$sP == R + h(B + C)$$

$$sP$$

$$= (s_B + s_C)P$$

$$= (r_B + hb + r_C + hc)P$$

$$= R_B + hB + R_C + hC$$

$$= R + h(B + C)$$

### 3.2.1 Random for signing

EdDSA signatures uses pseudo random generated $r = sha512(a_r||m)$. This has some very attractive properties[3]. By not using random the signing is not vulnerable to potential errors in random generation. No random means that the function is deterministic which is good for testability. It would be nice to implement a similar technique and have a deterministic pseudo random in DualSalt.

$$r_B = ???$$

$$R = r_B P + r_C P = R_B + R_C$$

$$h = sha512(R||D||m)$$

$$s_B = r_B + hb$$

$$s = s_B + s_C$$

1. $h$ is used in the EdDSA verification. To be compatible with EdDSA that expression can not be changed.
2. $h$ depends on $R$. And $R$ depends on both $R_B$ and $R_C$.
3. This means that the attacker (the other signer) can affect the shared value of $h$.
4. If the attacker can change $h$ while the attacked uses the same $r$ the secret scalar is leaked.

$$s_{B1} - h_1 b = s_{B2} - h_2 b$$

$$b = \frac{s_{B1} - s_{B2}}{h_2 - h_1}$$

5. $r$ has to depend on all parameters given by the attacker that affects $h$. If $h$ is affected so must be $r$.
6. This is no problem in ordinary EdDSA. $r$ is reused, but then $h$ is the same, no more information is given.
7. For device Bob that first calculate $r_B$ in function signCreateDual1() before it has the possibility to know the parameters for $h$. Those parameters are given first in function signCreateDual3(). It is simply impossible to make $r_B$ affected by $h$.

8. Another set of functions can be considered, but by checking the two calculations of $r$:

$$r_B = sha512(R_C||D||m||b_s)$$

$$r_C = sha512(R_B||D||m||c_s)$$

$r_B$ depends on $r_C$ and $r_C$ depends on $r_B$ what results in Catch-22 situation. Some kind of uniqueness has to be introduced by Bob or Caesar.

9. Better than to rely on just random data to create $r$ DualSalt has taken the approach to use nonce. The nonce has to be unique for each signature. It could be created from a sequence counter, time or random as long as the result is unique.

$$r_B = sha512(b_r||m||n_B)$$

$$r_C = sha512(c_r||m||n_C)$$

10. Note: Nonce is used both for device Bob and Caesar, but device Caesar has the option to use the parameters sent from Bob that affect $h$ as source for its nonce. This is attractive if Caesar is a shard server that can not rely on time or sequence counters due to parallel executions.

$$n_C = sha512(R_B||D)$$

$$r_C = sha512(b_r||m||n_C)$$

## 4 Decryption

Single and dual decryption is a Diffie Hellman with the encrypting party using an ephemeral key pair and the decrypting party or parties using long term keys.

**Definition table:**

| |
|---|
| $t_s$ = random seed to create ephemeral key pair |
| $T = tP$ ephemeral key used by the sender for a specific message |
| $S$ = shared secret |
| $d1$ = transaction message between Bob and Caesar |
| $m$ = message to encrypt |
| $x$ = cipher text |
| $y$ = cipher message |

## 4.1 Single decryption

| encrypt($m$, $A$, $t_s$) |
|---|
| $d = sha512(t_s)$ |
| $t = 2^{254} + \sum_{i=3}^{253} 2^i d_i$ |
| $T = tP$ |
| $S = tA$ |
| $x = crypto\_secretbox(S, m)$ |
| $y = (T, x)$ |

| decrypt($y$, $(a_s, A)$) |
|---|
| $d = sha512(a_s)$ |
| $a = 2^{254} + \sum_{i=3}^{253} 2^i d_i$ |
| (No need for validation) |
| $S = aT$ |
| $m = crypto\_secretbox\_open(S, x)$ |
| m |

Because the encrypter's key pair is ephemeral and if the key pair is forgotten after the encryption then forward secrecy is ensured.

## 4.2 Dual decryption

Dual decryption builds upon the equation below:

$$S = tD = t(B + C) = bT + cT$$

The encrypt function can be used to encrypt for a dual public key too.

| $y$ =encrypt($m$, $D$, $t_s$) |
|---|

Device Bob

Device Caesar

| decryptDual1($y$, $(c, c_r, C)$) |
|---|
| $groupElementVerify(T)$ see section 4.2.1 |
| $S_C = cT$ |
| $d1 = (S_C)$ |

$\overset{\longleftarrow}{\text{send } d1}$

```
decryptDual2($d1$, $y$, $(b, b_r, B)$)
    $groupElementVerify(T)$
    $S_B = bT$
    $S = S_B + S_C$
    $m = crypto\_secretbox\_open(S, x)$
m
```

Note: Only Bob ends up with the decrypted message and not Caesar.

### 4.2.1 Group element validation

As mentioned in section 2.2 there is a problem with using key rotation and dual decryption.

$$a' = (a + r) \mod n$$

We can not know if $a + r$ is larger or smaller than $n$

- If $a + r > n$ if $n$ is prime and $a$ and $r$ even then $a'$ is odd, not multiple of 8.
- If $a + r < n$ if $n$ is prime and $a$ even and $r$ odd then $a'$ is odd, not multiple of 8.

If the cipher message is tampered with by Bob a faulty $T$ is introduced. Caesar does the scalar multiplication $S_C = cT$ and sends the result back to Bob. Bob will then have information about $c$. After multiple key rotations Bob can also see when $c$ wraps by just looking at the LSB. And because Bob can affect $r$ this is a very powerful attack. The solution is for Bob and Caesar to validate that $T$ is in the group before the scalar multiplication.

Group element validation from[6]. $W$ is a point of order $n$ in $E(\mathbb{F}_q)$.

1. $W \neq \infty$
2. $x_W$ and $y_W$ are properly represented elements of $\mathbb{F}_q$ e.g. integers in the interval $[0, q-1]$ if $\mathbb{F}_q$ has prime order
3. $W$ satisfies the defining equation of the elliptic curve $E$
4. $nW == \infty$

The group elements representation used in DualSalt is the field coordinate $y$ as a 32 byte array with the top bit representing the sign of the $x$ coordinate.

With this information the $x$ coordinate can be calculated. Because the one coordinate is calculated from the other it can not be wrong, the check in point 2 is not needed. Because the coordinate can not be negative it is only needed to check if the coordinate is less then $q$.
$y_W = W_{0,\dots,254}$ (all except the top bit)

1. $y_W \neq y_\infty$
2. $y_W < q$
3. $nW == \infty$

### 4.3 Why Ed25519 and not X25519?

Decryption with two parties is unpractical when using X25519 because the scalar multiplication used in TweetNaCl for X25519 does not keep track of the sign on the $x$ coordinate. While decrypting the sign could be guessed, but this would be more complex than to use the Ed25519 scalar multiplication. Also to be able to use the same public key for both signing and decryption has its upside. Even if DualSalt can have the same key for decryption and signing it shall be said that many argue that separate keys should be used for security reasons

### 4.4 Symmetric crypto

For symmetric encryption and decryption DualSalt uses $Salsa20$ and $Poly130$ through the functions $crypto\_secretbox$ and $crypto\_secretbox\_open$ in TweetNaCl. The nonce used is:

000000000000000000000000000000000000000000000000 (in hex)

This static 24 byte long dummy value is used because a random ephemeral key pair is created and used for each message. No need to change the nonce if the key is used once only.

## References

[1] Things that use X25519,
https://ianix.com/pub/curve25519-deployment.html

[2] Things that use Ed25519,
https://ianix.com/pub/ed25519-deployment.html

[3] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang
High-speed high-security signatures,
https://ed25519.cr.yp.to/ed25519-20110926.pdf

[4] B. Ford, N Gailly, L. Gasser, P. Jovanovic, 2018
Collective Edwards-Curve Digital Signature Algorithm
https://tools.ietf.org/id/draft-ford-cfrg-cosi-00.html

[5] Trevor Perrin, 2017-03-28
Ed25519 "clamping" and its effect on hierarchical key derivation
https://moderncrypto.org/mail-archive/curves/2017/000874.html

[6] Adrian Antipa, Daniel Brown, Alfred Menezes, René Struik, Scott Vanstone
Validation of Elliptic Curve Public Keys
https://iacr.org/archive/pkc2003/25670211/25670211.pdf

[7] Daniel J. Bernstein, Bernard van Gastel, Wesley Janssen, Tanja Lange, Peter Schwabe, Sjaak Smetsers
TweetNaCl: A crypto library in 100 tweets
https://tweetnacl.cr.yp.to/tweetnacl-20140917.pdf

[8] SaltChannel: A simple, light-weight secure channel protocol based on TweetNaCl by Bernstein
https://github.com/assaabloy-ppi/salt-channel

[9] Protonmail: Secure Email Based in Switzerland
https://protonmail.com/

[10] blockchain-crypto-mpc: Protecting cryptographic signing keys and seed secrets with Multi-Party Computation
https://github.com/unbound-tech/blockchain-crypto-mpc