secp256k1-zkp Bulletproof format

Grin uses mimblewimble/secp256k1-zkp, a fork of the Blockstream library ElementsProject/secp256k1-zkp, which has Bulletproofs implemented but not yet merged into the main branch.

I have been trying to understand the proof output and recover the proof parameters from it.
Here is a sample proof output:


```
0b1bdf235e9c438aab5c6d02d3fe8173304bc528a3330825fb2311fa60fcdd6bfb92a248e26f849aebd511
d2b326fa34b7f3030517d2f8e08a9b3cac7fa9fd200d07a46ca6ec5af30ce569b1e5faf2acf525cf1ed90c
bed74ab7378b9b3957f28635fe7440aac2dc2c4bf43265b6ad1bfa82fddd9a827c4e97a913ce451b9a66bb
06d3c08e03e85e98c581bbdf8c852796371a4603b8d52b80a1f2e95bd5e2a91c7a00b4d4564d9586235a78
58d9ce8a8888bead7d51be2dd802de5af2921e079586817fce16d36c7764af8b4bf133b56b39970d6a568b
f9ff101e6d33409e7c3bb081df7425b276655be611941245ceaad529495a86bc0e3d0f8634a8acf65c34c4
e244959a5098bd58285408945a247d2fd894e5b18027d698c7e494e4256110553df54babf90592fbdffa01
38b6b5a2a423ea5e2ec4d8f852a33c271a73b10fed9e1ee8cb2db1e71311cacd9e1d0b6dbf6cfab15723ec
3cac4cc52154fc9d532202a085238e756ad1fa804cce2a634decc1b348f6ff939f9f80187d85aa5c308224
a505c75e7f58fc7f35424276db7956474c1895e23ac55f864f4177b59f3ce92ef8c99e011cf55e0cefc563
5d2eaf573df29af057a19bb209392a8c0e29a4b77adad76d385422e7f1d06de2d4f14e61ac3619aa22ae5b
c288bb41cb56ddb70bb39ae84d00eb0cb34b4063bb55a83b9fe52604e545adcd41beb6ce14cdff73a21beb
7493fa443a34585b7d2927f608cad17aa5f0e8e154b14d35315f63dd3580e80d06d8be4039f58778967f7b
f2cdd9020fbcc9fed799b8159814f6a261c568e8b59c59df3180efb9cc13c576bf313248c96fa867aba43a
80e799ff19ac685d7208cea7944dc9dcba7a61f2809540ecd0711e76b601969bdc551845e0b11fb821871d
00e417ad002a70353867db25fa647e98a0db4c3bbaf828d97fc66079ef0d
```


First, we have two 32 byte scalars. These are the already negated versions of taux and mu. We negate them such that the verifier doesn't have to do it. (*rangeproof_impl.h 701-702*)

`0b1bdf235e9c438aab5c6d02d3fe8173304bc528a3330825fb2311fa60fcdd6b` `(5024686248162052924872973414517693136231035491146611931625298995470137089387)` taux (negated)

`fb92a248e26f849aebd511d2b326fa34b7f3030517d2f8e08a9b3cac7fa9fd20` `(113789604713728301456840843635921464549630649029317112794749678552821986360608)` mu (negated)

After that, we have 4 points, which represent commitments A, S, T1, T2. Points are encoded in a very smart way. We have one offset byte. We use this offset byte to determine
if the points y value needs to be negated when recovered. If this is the case, the bit is set to 1 (starting at the LSB); otherwise, it is left at 0. If we have more then
eight points, we need 2 bytes offset, if we have more then 16, then three, and so forth. (*rangeproof_impl.h 703*)

`0d` offset

`0000 1101` offset in binary

From this, we can recover the 4 points. (I am using the standard compressed point version here with leading 02 or 03)

`0307a46ca6ec5af30ce569b1e5faf2acf525cf1ed90cbed74ab7378b9b3957f286` A (03 because of the 1 bit in the offset)

`0235fe7440aac2dc2c4bf43265b6ad1bfa82fddd9a827c4e97a913ce451b9a66bb` S (02 here because of the 0 bit in the offset)

`0306d3c08e03e85e98c581bbdf8c852796371a4603b8d52b80a1f2e95bd5e2a91c` T1

`037a00b4d4564d9586235a7858d9ce8a8888bead7d51be2dd802de5af2921e0795` T2

Next, we have the final value of the dot product which again is a 32-byte scalar (*inner_product_impl.h 811*)

`86817fce16d36c7764af8b4bf133b56b39970d6a568bf9ff101e6d33409e7c3b (60838727059453008536034129618950719358562694528830851223208761064459354405947)` dot

Then we have the final values (32-byte scalars) of the shrunk vectors a, b used in the inner product protocol. The library does not do the last round of the protocol, meaning it will stop when the vectors are of
length two instead of length one. This is because every round creates two commitments Li and Ri. If we don't do the last round, we spare two commitments with the cost that our two vectors are of size two instead of one, which is more space-efficient, and we save computing time. (*inner_product_impl.h 835-836*)

`b081df7425b276655be611941245ceaad529495a86bc0e3d0f8634a8acf65c34 (79836526842770413616887368822368313168206709119259360230886972660827215518772)` a1 `c4e244959a5098bd58285408945a247d2fd894e5b18027d698c7e494e4256110` `(89053099110995010594661038229216983605420219413380810817771304480647904846096)` b1

`553df54babf90592fbdffa0138b6b5a2a423ea5e2ec4d8f852a33c271a73b10f` `(38556062768490931671602594328406809964645337276375001909352144464132590252303)` a2

`ed9e1ee8cb2db1e71311cacd9e1d0b6dbf6cfab15723ec3cac4cc52154fc9d53` `(107477520278964342277912932357487306000871347661927764278313323679782451060051)` b2

And last we have the commitments Li and Ri of every round. In Grin we have 64-bit range proofs. This means we have six rounds (log(64) = 6); however, since we stop early, we only do five rounds, so 10 points instead of 12. The implementation always computes L before R. (*inner_product_impl.h 627*)
Again we have an offset in which we specify how to recover y values. Now since we have more than eight points we need two bytes offset. (*inner_product_impl.h 839*)

`2202` offset

`0010 0010 0000 0010` offset in binary

`02a085238e756ad1fa804cce2a634decc1b348f6ff939f9f80187d85aa5c308224` L1

`03a505c75e7f58fc7f35424276db7956474c1895e23ac55f864f4177b59f3ce92e` R1

`02f8c99e011cf55e0cefc5635d2eaf573df29af057a19bb209392a8c0e29a4b77a` L2

`02dad76d385422e7f1d06de2d4f14e61ac3619aa22ae5bc288bb41cb56ddb70bb3` R2

`029ae84d00eb0cb34b4063bb55a83b9fe52604e545adcd41beb6ce14cdff73a21b` L3

`02eb7493fa443a34585b7d2927f608cad17aa5f0e8e154b14d35315f63dd3580e8` R3

`020d06d8be4039f58778967f7bf2cdd9020fbcc9fed799b8159814f6a261c568e8` L4

`02b59c59df3180efb9cc13c576bf313248c96fa867aba43a80e799ff19ac685d72` R4

`0208cea7944dc9dcba7a61f2809540ecd0711e76b601969bdc551845e0b11fb821` L5

03871d00e417ad002a70353867db25fa647e98a0db4c3bbaf828d97fc66079ef0d R5