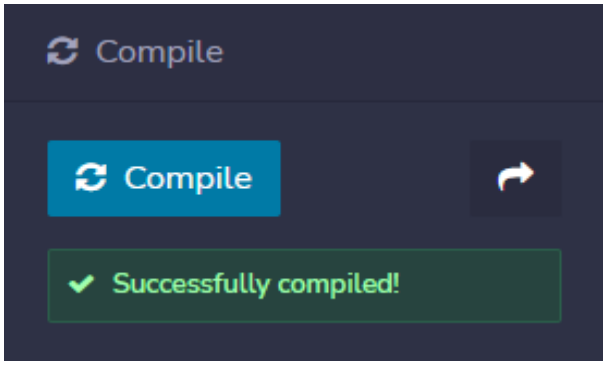
Documentation of zero knowledge proof

First step:

We have created a square.zok file and compiled this file using the ZOKRATES tool.

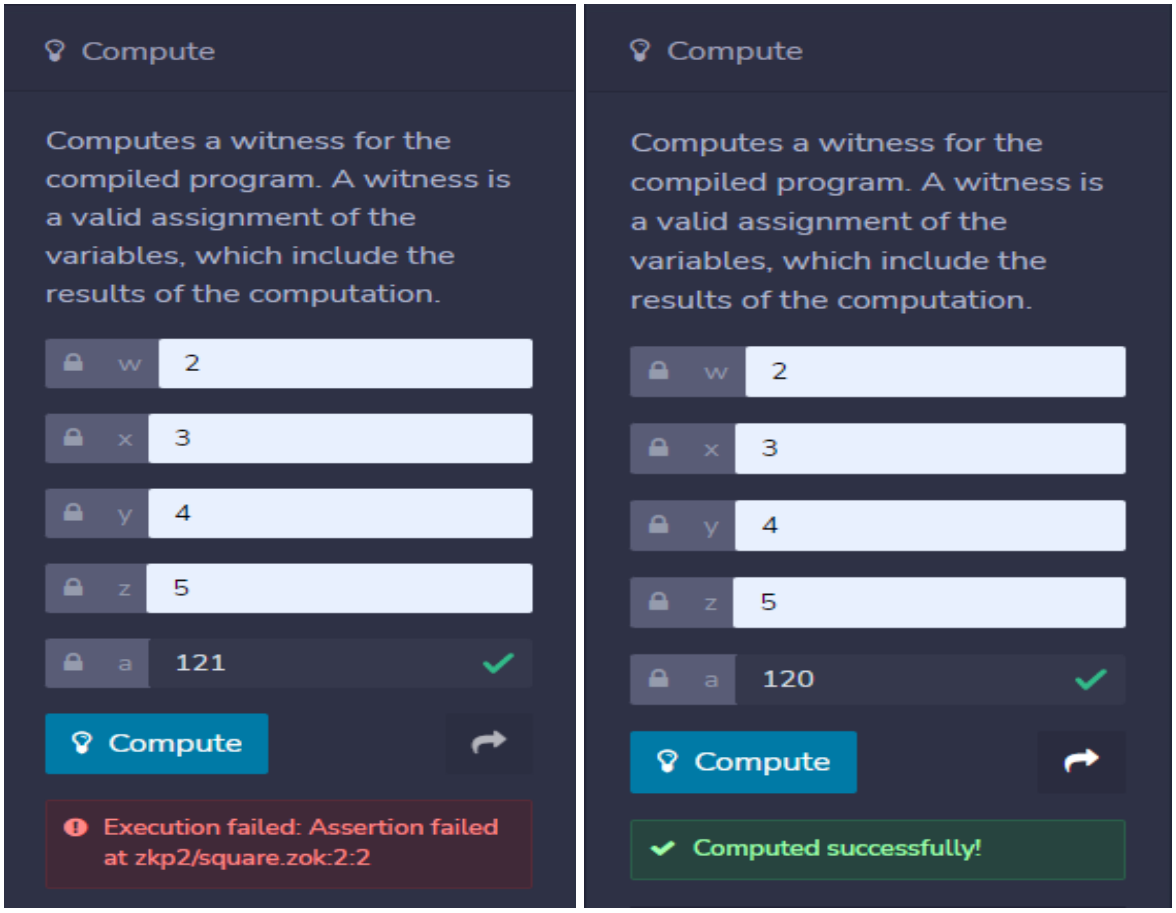


Code of square.zok:

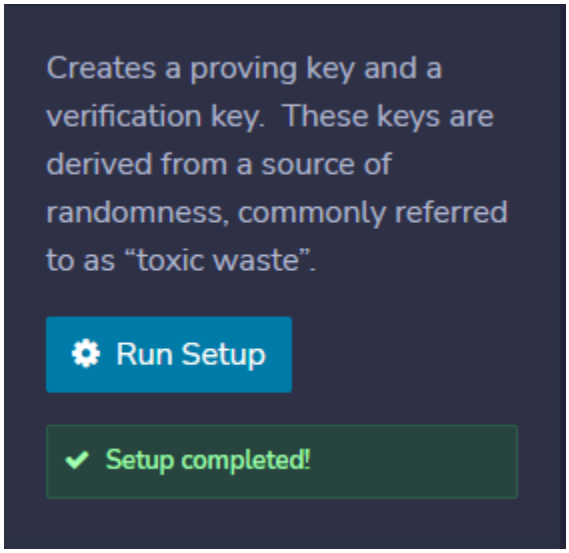
```
def main(private field w, private field x, private field y, private field
z, private field a) {
    assert(w * x * y * z == a);
    return;
}
```

I have kept all fields private, otherwise you can remove private from all fields but then zero-knowledge will be ended. Whose fields are kept private, those data can't be shown otherwise data can be shown. It means which fields are not private that data will not be shown.

So after compilation we have computed this code:
In the first field we have entered 2, then in the next field we have entered 3, then 4, then 5 and in the last field, which will be all field multiplication(120). If I put another number from 120, the computation will fail otherwise. It's mean "a" must be true:



Creates a proving key and a verification key. These keys are derived from a source of randomness, commonly referred to as “toxic waste” using run **“Run Setup”.**



So, after that “**verification_key.json”** file have been created:
In this file the specific values of the parameters in the file would be used in a proof calculation. The parameters include "alpha", "beta", "gamma", "delta", and "gamma_abc", which are used in the proof calculation in various ways. Without more information on the specific system or application these parameters are being used in, it is difficult to provide a detailed explanation of their specific roles in the proof.

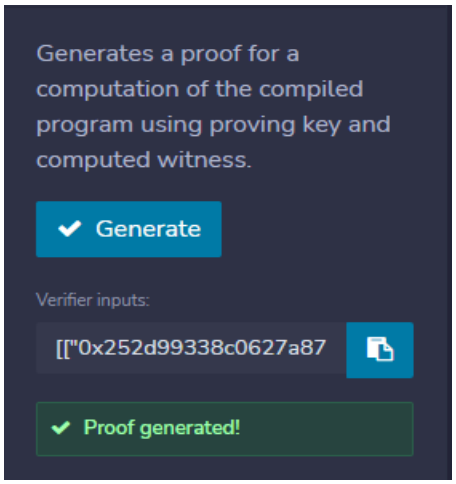

```
{
  "scheme": "g16",
  "curve": "bn128",
  "alpha": [
    "0x04ac649eeb4f358eb6f15251d9fb908e4559be85c8234d5a3234ac461def8b34",
    "0x1c60086908a039add513480da8a2561331408436457aff2129dbe2bba07b40ad"
  ],
  "beta": [
    [
"0x1afdf71742cc8653c06ada6217fa801cf62fe84b070486010dec63d22b0ccf46",
      "0x15083e5923f631baee7467c77d33716524e282b11588b534401e8477a3564ebf"
    ],
    [
"0x168dc03e01493e40db2b07300b60fda0d531003a3ae8e15562d636249eac00bc",
      "0x26b0c8280dbade46089e86e027b2f3af9f5533d9bd68f6bf37f2a4734ea18d2f"
    ]
  ],
  "gamma": [
    [
"0x1cb8354a018eeb7ae1eb12a0ce8c70ac9ceba93d967429f5760ef36bfc80b3e1",
      "0x06690e0e98f8bc2c6f0f1c17e6dde2e4c605ebdaa007984de7d333a11becb412"
    ],
    [
"0x1574f882c4a4be61151acd713a036640bac34e720cbbb94f7a8f8615aaaa65ad",
      "0x2488a238250d2f9b1204ada0e2fd912a29ed904d6ac4b8b0c967e4d284969c23"
    ]
  ],
  "delta": [
```

```
    [

"0x10a5425f7ebf860134f6677d584e721c24e97a7f2d18c97263748dcf667c8ea0",
      "0x0fa0bb00d86c05cbdb2cfbabc06092ca8fc9084b3f24fd42f3fac946bce48bc9"
    ],
    [

"0x0f5353a1bf184072a698b37a7ce6dfee2dd9f2249f93cae5ce1bb5793b042943",
      "0x072ac5bb23b374f43150b464920a311c58a8067443a97b34cc97162a2da5c441"
    ]
  ],
  "gamma_abc": [
    [

"0x1d795c7d9dd65c33b24a714b418947859f7ece97e292f06b9305386087bc1daf",
      "0x0cab3be538ce9758a331a6bb55c7abad992934cdb142f90f45d5e8d54df1e0b5"
    ]
  ]
}
```

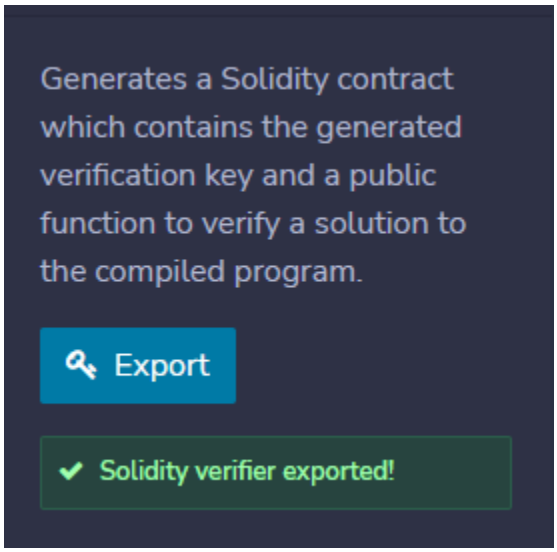

After that we have generate proof and copy that proof:

PROOF:

[["0x252d99338c0627a871e7fd7cfd7ebe252a06dac32c9fda9d54ee2dad4e4de619","0x10dc66cff6b5e2f767719ac5d8e79936652f580b229a6659ea20fa427008538b"],[["0x15c3925a8a998f75c2650138af17bd10bb535f39ccdb6096e633eb8e347e70cc","0x22f0e1f4b0f873b6f68751eeeb254ae63349ffb672256bc5b2850fba0eb3a13d"],["0x0d676ab157480615c3d7732c59260fec9e53698730cbd52e341433389a79b0bb","0x1e87d4f1c8b033d01abf2c141eb9c47c4b54abc46c4e09d243b9190b72b2e2db"]],["0x17cb874a66a8c21adb17c739c6873407fdaa7572a52c201fdf34b1357ac355d4","0x2c0c16c261968de428c4226f95eee60cabbdf9700b6343f362a7f67645a6929c"]]



Then we will verify this proof from verifier using “verifier.sol” file:



Here is the code of “verifier.sol” file:

```
// SPDX-License-Identifier: MIT
// This file is MIT Licensed.
//
// Copyright 2017 Christian Reitwiessner
// Permission is hereby granted, free of charge, to any person obtaining a
copy of this software and associated documentation files (the "Software"),
to deal in the Software without restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following conditions:
// The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN
NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR OTHER DEALINGS IN THE SOFTWARE.
pragma solidity ^0.8.0;
library Pairing {
    struct G1Point {
        uint X;
        uint Y;
```

```
    }
    // Encoding of field elements is: X[0] * z + X[1]
    struct G2Point {
        uint[2] X;
        uint[2] Y;
    }
    /// @return the generator of G1
    function P1() pure internal returns (G1Point memory) {
        return G1Point(1, 2);
    }
    /// @return the generator of G2
    function P2() pure internal returns (G2Point memory) {
        return G2Point(

[10857046999023057135944570762232829481370756359578518086990519993285655852781,

11559732032986387107991004021392285783925812861821192530917403151452391805634],

[84956539231234314176049732474892724384181905872636001487702806493069581019930,

40823678758634336813322034031454355683168513275934012081057410762141200935 31]
        );
    }
    /// @return the negation of p, i.e. p.addition(p.negate()) should be zero.
    function negate(G1Point memory p) pure internal returns (G1Point memory) {
        // The prime q in the base field F_q for G1
        uint q = 21888242871839275222246405745257275088696311157297823662689037894645226208583;
        if (p.X == 0 && p.Y == 0)
            return G1Point(0, 0);
        return G1Point(p.X, q - (p.Y % q));
    }
    /// @return r the sum of two points of G1
```

```
    function addition(G1Point memory p1, G1Point memory p2) internal view
returns (G1Point memory r) {
        uint[4] memory input;
        input[0] = p1.X;
        input[1] = p1.Y;
        input[2] = p2.X;
        input[3] = p2.Y;
        bool success;
        assembly {
            success := staticcall(sub(gas(), 2000), 6, input, 0xc0, r,
0x60)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require(success);
    }


    /// @return r the product of a point on G1 and a scalar, i.e.
    /// p == p.scalar_mul(1) and p.addition(p) == p.scalar_mul(2) for all
points p.
    function scalar_mul(G1Point memory p, uint s) internal view returns
(G1Point memory r) {
        uint[3] memory input;
        input[0] = p.X;
        input[1] = p.Y;
        input[2] = s;
        bool success;
        assembly {
            success := staticcall(sub(gas(), 2000), 7, input, 0x80, r,
0x60)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require (success);
    }
    /// @return the result of computing the pairing check
    /// e(p1[0], p2[0]) *  .... * e(p1[n], p2[n]) == 1
    /// For example pairing([P1(), P1().negate()], [P2(), P2()]) should
    /// return true.
```

```
    function pairing(G1Point[] memory p1, G2Point[] memory p2) internal
view returns (bool) {
        require(p1.length == p2.length);
        uint elements = p1.length;
        uint inputSize = elements * 6;
        uint[] memory input = new uint[](inputSize);
        for (uint i = 0; i < elements; i++)
        {
            input[i * 6 + 0] = p1[i].X;
            input[i * 6 + 1] = p1[i].Y;
            input[i * 6 + 2] = p2[i].X[1];
            input[i * 6 + 3] = p2[i].X[0];
            input[i * 6 + 4] = p2[i].Y[1];
            input[i * 6 + 5] = p2[i].Y[0];
        }
        uint[1] memory out;
        bool success;
        assembly {
            success := staticcall(sub(gas(), 2000), 8, add(input, 0x20),
mul(inputSize, 0x20), out, 0x20)
            // Use "invalid" to make gas estimation work
            switch success case 0 { invalid() }
        }
        require(success);
        return out[0] != 0;
    }
    /// Convenience method for a pairing check for two pairs.
    function pairingProd2(G1Point memory a1, G2Point memory a2, G1Point
memory b1, G2Point memory b2) internal view returns (bool) {
        G1Point[] memory p1 = new G1Point[](2);
        G2Point[] memory p2 = new G2Point[](2);
        p1[0] = a1;
        p1[1] = b1;
        p2[0] = a2;
        p2[1] = b2;
        return pairing(p1, p2);
    }
    /// Convenience method for a pairing check for three pairs.
    function pairingProd3(
            G1Point memory a1, G2Point memory a2,
```

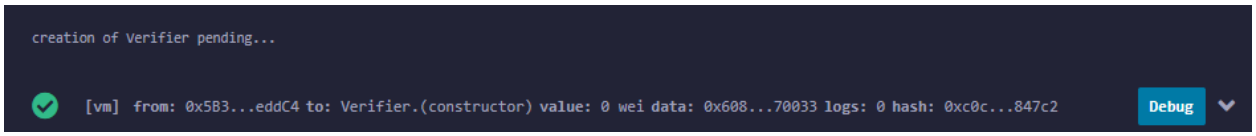
```
            G1Point memory b1, G2Point memory b2,
            G1Point memory c1, G2Point memory c2
    ) internal view returns (bool) {
        G1Point[] memory p1 = new G1Point[](3);
        G2Point[] memory p2 = new G2Point[](3);
        p1[0] = a1;
        p1[1] = b1;
        p1[2] = c1;
        p2[0] = a2;
        p2[1] = b2;
        p2[2] = c2;
        return pairing(p1, p2);
    }
    /// Convenience method for a pairing check for four pairs.
    function pairingProd4(
            G1Point memory a1, G2Point memory a2,
            G1Point memory b1, G2Point memory b2,
            G1Point memory c1, G2Point memory c2,
            G1Point memory d1, G2Point memory d2
    ) internal view returns (bool) {
        G1Point[] memory p1 = new G1Point[](4);
        G2Point[] memory p2 = new G2Point[](4);
        p1[0] = a1;
        p1[1] = b1;
        p1[2] = c1;
        p1[3] = d1;
        p2[0] = a2;
        p2[1] = b2;
        p2[2] = c2;
        p2[3] = d2;
        return pairing(p1, p2);
    }
}

contract Verifier {
    using Pairing for *;
    struct VerifyingKey {
        Pairing.G1Point alpha;
        Pairing.G2Point beta;
        Pairing.G2Point gamma;
```

```
        Pairing.G2Point delta;
        Pairing.G1Point[] gamma_abc;
    }
    struct Proof {
        Pairing.G1Point a;
        Pairing.G2Point b;
        Pairing.G1Point c;
    }
    event boolEvent(bool);
    function verifyingKey() pure internal returns (VerifyingKey memory vk)
{
        vk.alpha =
Pairing.G1Point(uint256(0x04ac649eeb4f358eb6f15251d9fb908e4559be85c8234d5a
3234ac461def8b34),
uint256(0x1c60086908a039add513480da8a2561331408436457aff2129dbe2bba07b40ad
));
        vk.beta =
Pairing.G2Point([uint256(0x1afdf71742cc8653c06ada6217fa801cf62fe84b0704860
10dec63d22b0ccf46),
uint256(0x15083e5923f631baee7467c77d33716524e282b11588b534401e8477a3564ebf
)],
[uint256(0x168dc03e01493e40db2b07300b60fda0d531003a3ae8e15562d636249eac00b
c),
uint256(0x26b0c8280dbade46089e86e027b2f3af9f5533d9bd68f6bf37f2a4734ea18d2f
)]);
        vk.gamma =
Pairing.G2Point([uint256(0x1cb8354a018eeb7ae1eb12a0ce8c70ac9ceba93d967429f
5760ef36bfc80b3e1),
uint256(0x06690e0e98f8bc2c6f0f1c17e6dde2e4c605ebdaa007984de7d333a11becb412
)],
[uint256(0x1574f882c4a4be61151acd713a036640bac34e720cbbb94f7a8f8615aaaa65a
d),
uint256(0x2488a238250d2f9b1204ada0e2fd912a29ed904d6ac4b8b0c967e4d284969c23
)]);
        vk.delta =
Pairing.G2Point([uint256(0x10a5425f7ebf860134f6677d584e721c24e97a7f2d18c97
263748dcf667c8ea0),
uint256(0x0fa0bb00d86c05cbdb2cfbabc06092ca8fc9084b3f24fd42f3fac946bce48bc9
)],
[uint256(0x0f5353a1bf184072a698b37a7ce6dfee2dd9f2249f93cae5ce1bb5793b04294
```

```
3),
uint256(0x072ac5bb23b374f43150b464920a311c58a8067443a97b34cc97162a2da5c441
)]);
        vk.gamma_abc = new Pairing.G1Point[](1);
        vk.gamma_abc[0] =
Pairing.G1Point(uint256(0x1d795c7d9dd65c33b24a714b418947859f7ece97e292f06b
9305386087bc1daf),
uint256(0x0cab3be538ce9758a331a6bb55c7abad992934cdb142f90f45d5e8d54df1e0b5
));
    }
    function verify(uint[] memory input, Proof memory proof) internal view
returns (uint) {
        uint256 snark_scalar_field =
218882428718392752222464057452572750885483644004160343436982041865758084955
617;
        VerifyingKey memory vk = verifyingKey();
        require(input.length + 1 == vk.gamma_abc.length);
        // Compute the linear combination vk_x
        Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
        for (uint i = 0; i < input.length; i++) {
            require(input[i] < snark_scalar_field);
            vk_x = Pairing.addition(vk_x,
Pairing.scalar_mul(vk.gamma_abc[i + 1], input[i]));
        }
        vk_x = Pairing.addition(vk_x, vk.gamma_abc[0]);
        if(!Pairing.pairingProd4(
             proof.a, proof.b,
             Pairing.negate(vk_x), vk.gamma,
             Pairing.negate(proof.c), vk.delta,
             Pairing.negate(vk.alpha), vk.beta)) return 1;
        return 0;
    }
    function verifyTx(
            Proof memory proof
        ) public returns (bool r) {
        uint[] memory inputValues = new uint[](0);

        if (verify(inputValues, proof) == 0) {
            emit boolEvent(true);
            return true;
```

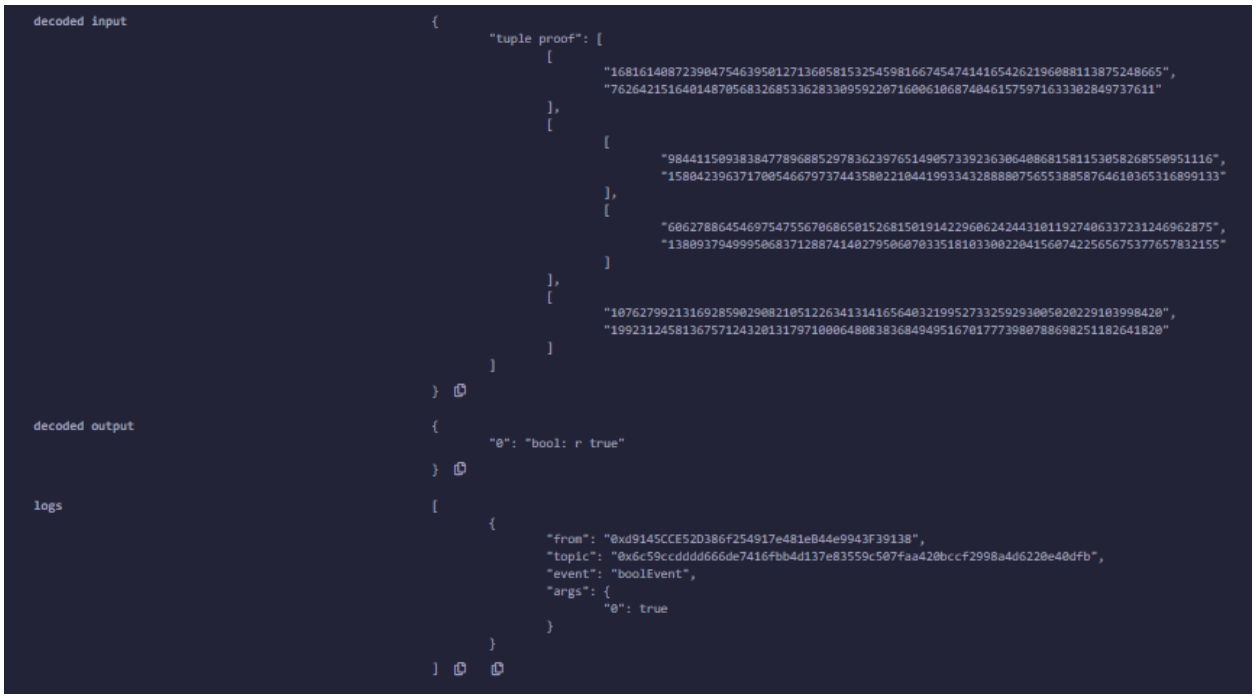
```
        } else {
            emit boolEvent(false);
            return false;
        }
    }
}
```

After the creation of verifier.sol, we will verify myself from proof:
So, lets deploy the contract:

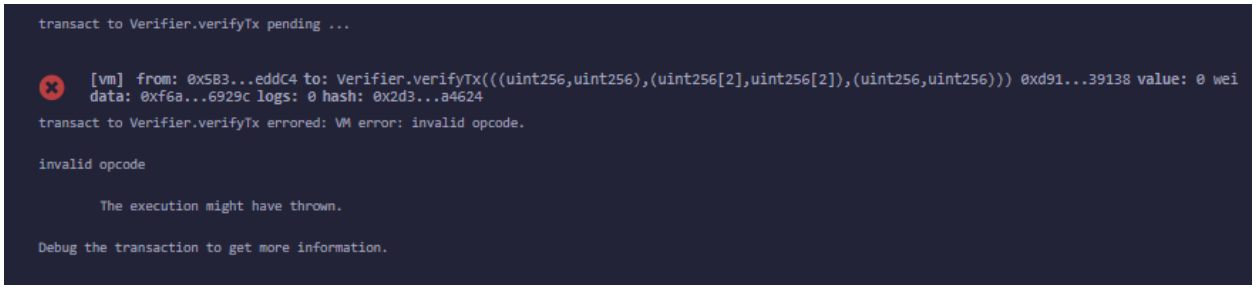


Then put the proof in input:

Then result comes true:



Now we will remove a digit and add another digit and then run this:
So transaction have been failed:




decoded output
{
"0": "bool: r false"
}
logs
[]
val
0 wei
transact to Verifier.verifyTx errored: VM error: invalid opcode.
invalid opcode
The execution might have thrown.
Debug the transaction to get more information.

////////////////////////////////////// END //////////////////////////////////////