ZKSNARK (Zokrates example)

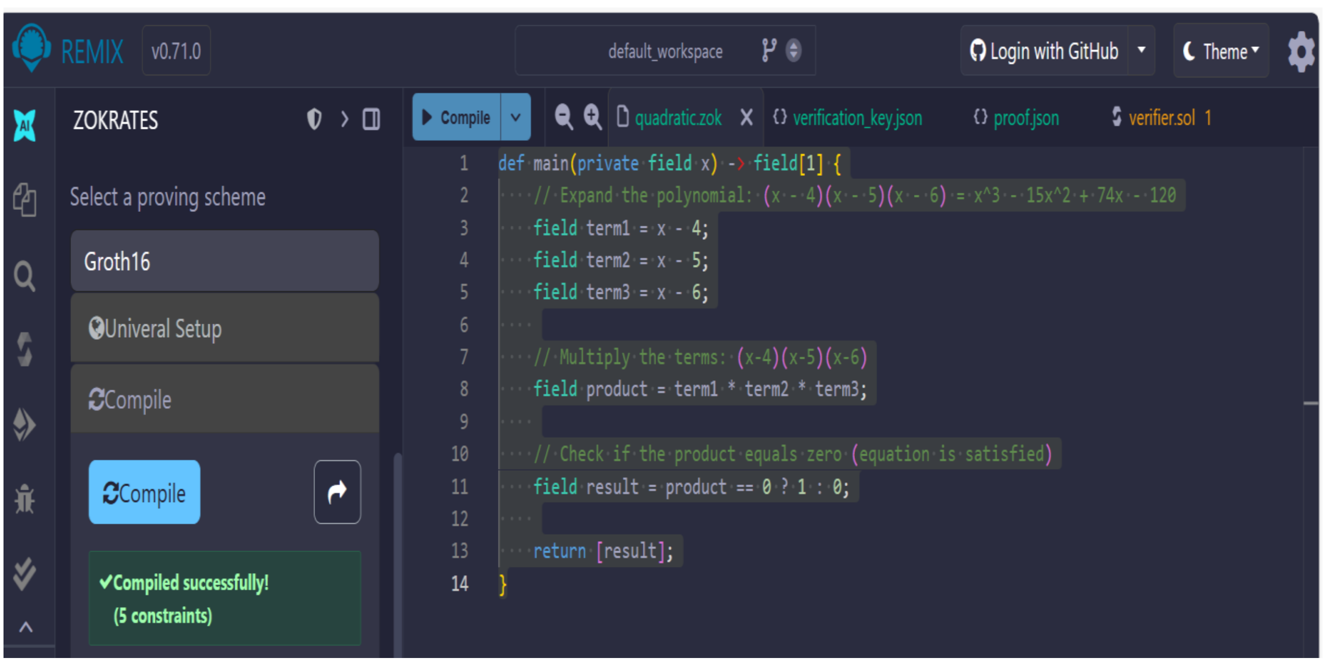
Let's explore how our previous ZKSNARK example is implemented using ZoKrates. We'll use Remix, the online IDE, for this purpose.

This is related Zokrate code

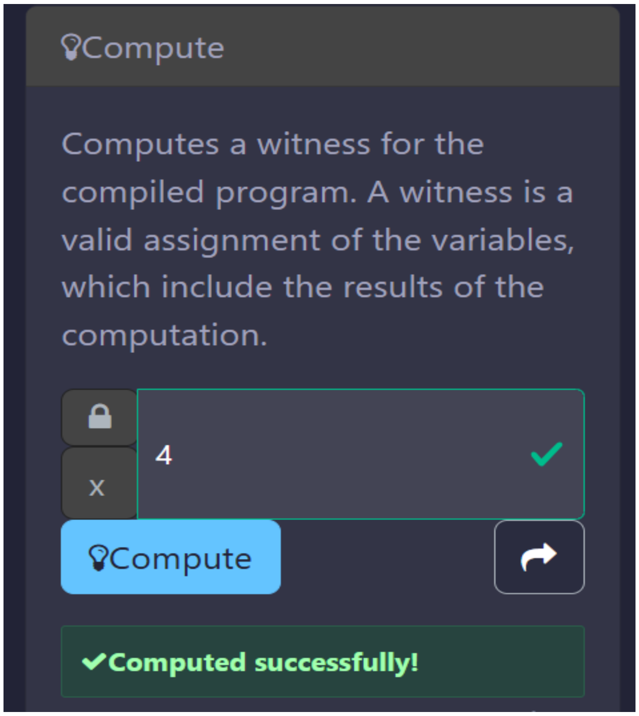
```
def main(private field x) -> field[1] {
   // Expand the polynomial: (x - 4)(x - 5)(x - 6) = x^3 - 15x^2 + 74x - 120
   field term1 = x - 4;
   field term2 = x - 5;
   field term3 = x - 6;

   // Multiply the terms: (x-4)(x-5)(x-6)
   field product = term1 * term2 * term3;

   // Check if the product equals zero (equation is satisfied)
   field result = product == 0 ? 1 : 0;

   return [result];
}
```

All you just need to do is paste this program on creating a new .zok file on the file expoler in the remix studio. Then paste above code. Here make sure to activate Zokaretes plugin before that.

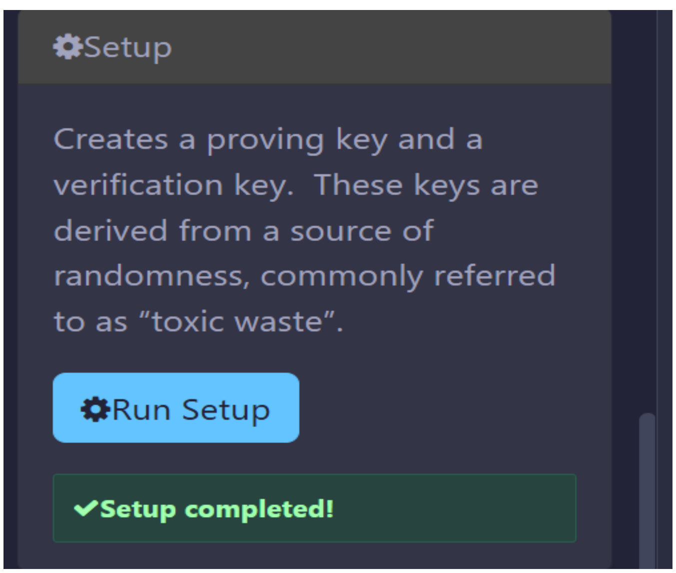
Then compile it



Compute the witness as follows. Here you just need to put the public and private field related to our program. Here we used only 1 private field for x . so put x as 4.



Click Run Setup to generate all the keys. Meaning alpha, beta, gamma , Kpub points. I think you remember this Kpub . We split the equation as privet and pub. Here We calculated Kpub points for the polynomial commitments for the public set of equation. Here Kpub is known by gamma_abc.

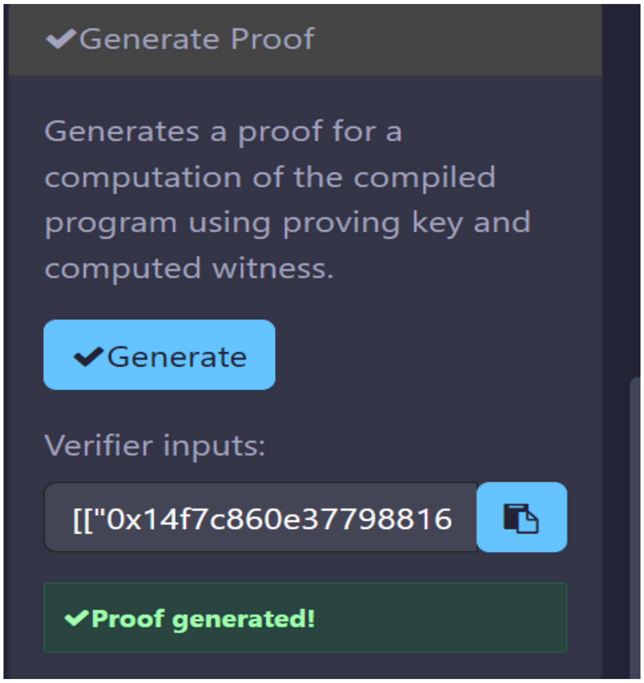


These are the generated values

{
  "scheme": "g16",
  "curve": "bn128",
  "alpha": [
   "0x0356dc98841e4b2877ba81de9fa9e52386abb4eba8419265417b11f7097df9c5",
   "0x1cf6f49a5f20a43ed1af11aa6f2d61ebd43fa0eef822f1145363fcca26dfe857"
  ],
  "beta": [
   [
    "0x22dafb88834a88901285087060dc89a5911da9a56baef199e59b93b00a0b85ae",
    "0x1287dc0fb504ab2e01417700b288325477ba252fb040921792c91222419ea899"
   ],
   [
    "0x28b05f79c892d9ed2d13ca92fb957a2ec837f9e89b9049dec3b2fd048c2bb307",
    "0x1c7a97171627c9353af94fff09327ab19551040a9ecbeebad7859d40c0301427"
   ]
  ],
  "gamma": [
   [
    "0x07b3d46d69b5e124c840a40579edd5d8ea10fac1fd2dc5b20aec3363ab5c0a5a",
    "0x1bb50cbc85b50d5485a12d6442564bf7666518695ea1cd48185123cb49f535ae"
   ],
   [
    "0x19f65ae0dd9139c07edae895d679e1b42e8fdc4434170a6006b39aef696448bf",
    "0x1a36af03273421225a59d4a8966fed4f9b793107d42c8b0a59a4d3801ad1bab1"
   ]
  ],
  "delta": [
   [
    "0x2e9d5bb43da1be6f92a71833d53fa1199cc86224801e2412a6e9f7d40c1e5f45",
    "0x000cd9a2c3c03ca136b3af4b66b44150656c48c51933b8b2515af9de48d8994a"
   ],
   [
    "0x29afd09c7de49ee8624e6c142d9e4c9da5b1993c2d13b6b75156c7293d32b1cc",
    "0x23561d4c0b9870a3553b6974c5be61364e26efdfdcac18a0da55a28568dccba5"
   ]
  ],
  "gamma_abc": [
   [
    "0x29c4371ce5932c83dd365b407c1af6a140ac445a5494fcced3453395402a4b25",
    "0x07bbf0e0b04b9815c4182b5978caf85c6c3a7c95d76a926beb1de6ee542f7a6a"
   ],
   [
    "0x18d87de16ac9f29ab4de32659d032b45a55f1ba9d7c22b256a219c129ae907ec",
    "0x1db5b018041c6f82c5ca9adb7eab8cb7d4ea06fa0d4cbd35ff261d3cfcf0e25e"
   ]
  ]

}

Next step is Generate the Proof, Just click the Generate button as shown below. It generate elliptic curve points used for the paring verification. (A , B , C)




```
{
 "scheme": "g16",
 "curve": "bn128",
 "proof": {
  "a": [
   "0x14f7c860e37798816c8005f3c9d2e7ff5ac81fb3f636d6eb803f43f9b6893762",
   "0x19f0492a4f2f5bb4385b8e9225857ce02af8c49f90d9a6cec3d2a96389273131"
  ],
  "b": [
   [
    "0x1f3b22991eff290d69253fde73dc8ffec21f54b1610088c9e4c3b8df716c0e8a",
    "0x04168da3fbc0df1af66df1c5af53786bee00ba26f0fecc596c45af9549791d8d"
   ],
   [
    "0x1a59d777eefd7804023accd6c2ac3a355696df44efa7ea19cd15b861283218d0",
    "0x1e7efa48f5af84b49d32ef137aeadd56b52754414b6c890aab8287b0b06ed92a"
   ]
  ],
  "c": [
   "0x1834a3d39ac270ea6042dabe1813b919d20d7aa9b70629a947d661297ab82b3b",
   "0x2c42281801a0807f7bb58177d1d7193b007196b85aea168b0fc9869a3370c98c"
  ]
```

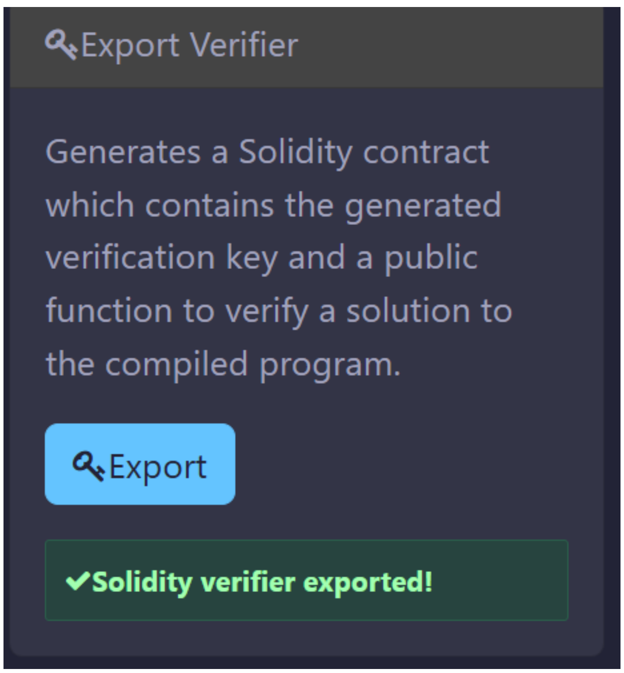
```
 },
 "inputs": [
  "0x0000000000000000000000000000000000000000000000000000000000000001"
 ]
}
```

Here inputs means public inputs we used. Our program set output as this

```
field result = product == 0 ? 1 : 0;
```

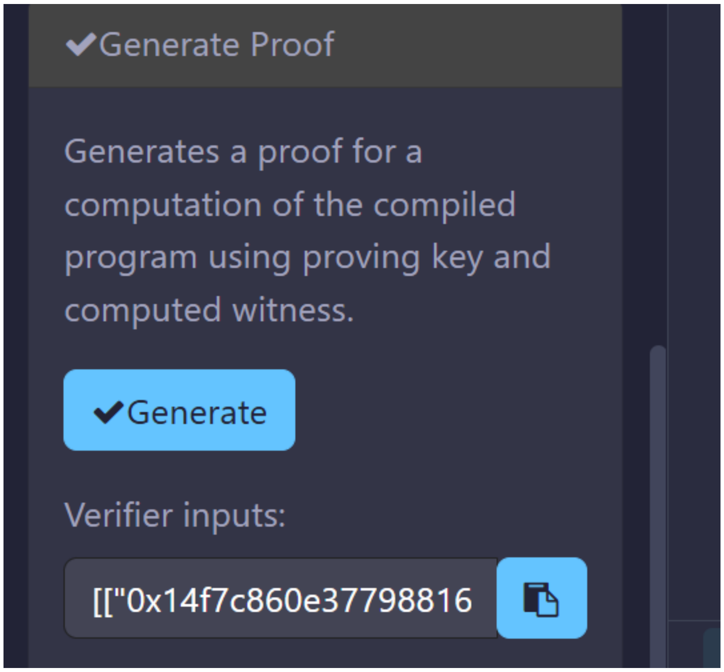
Means if product is 0 then it should be return 1 . That's the only public output we used here. Here x is considered private value, private values no need to used as inputs array as above shown. Above generated inputs array is only for public input/output values.

Then click Export to generate solidity contract for the paring verification.

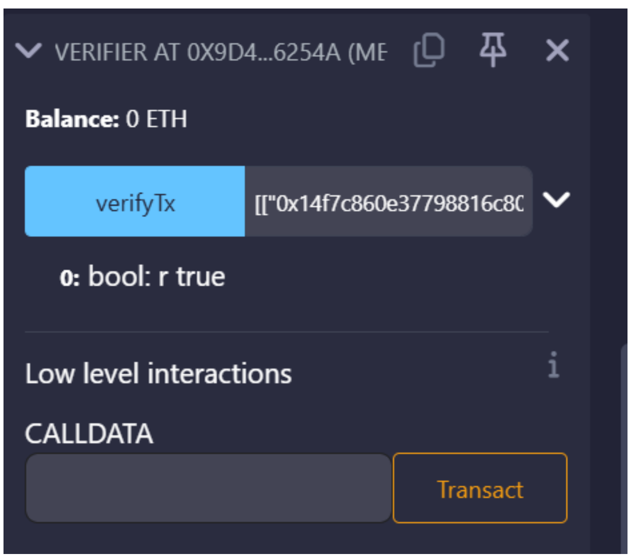


[Here](#) is solidity contract generated.

Now you need to deploy the verifier contract and use this proof to verify the it.

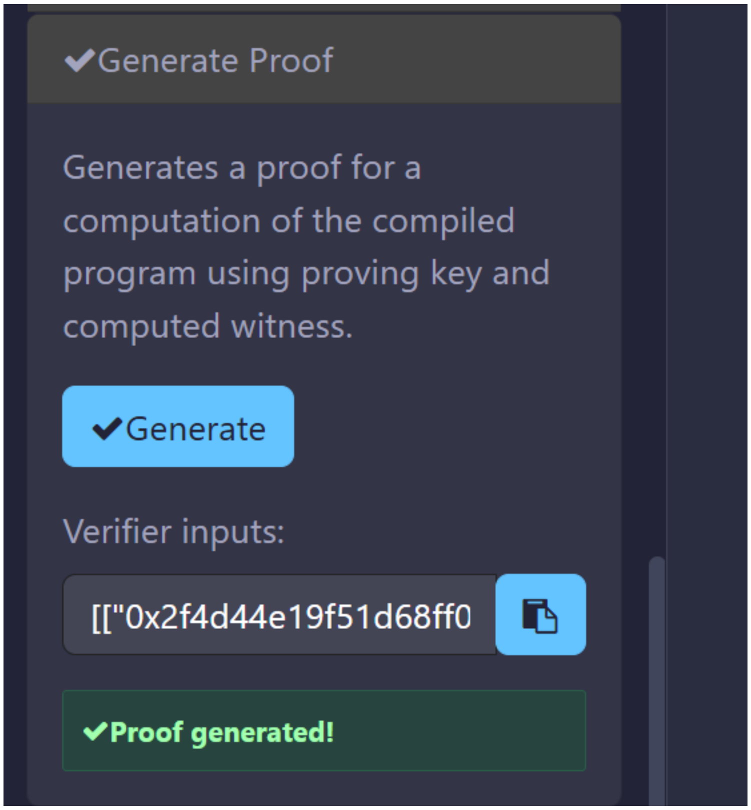


Deploy the above generated verifier contract. Copy the Verifier inputs and paste it to verifyTx function as shown below. You can see it returns true.



Now lets check proof are applied with user randomizations (r and s , here r and s not shown in the keys generated above, but it actually included)

Lets generate another proof.



Paste this it , it also return true on verifiyTx function. That shows user able to generate multiple proof for the verification. But only issue here is , User able to the generate multiple proof but , but developer should keep tack of r and s , may be on another solidity contract which does not allow user to replay attack.

ZKSNARK Proof Verification(Pairing Equation Explained)

The Core Pairing Equation

The verification checks this fundamental equation:

e(A, B) = e(α, β) · e(vk_x, γ) · e(C, δ)

Where:

- A, B, C are the proof components
- α, β, γ, δ are the verification key components
- vk_x is the linear combination of public inputs

Step-by-Step Verification Process

1. Public Input Processing

solidity

*// Compute vk_x = gamma_abc[0] + input[0]*gamma_abc[1] + input[1]*gamma_abc[2] + ...*

```
Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);

for (uint i = 0; i < input.length; i++) {

   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]);
```

For your case with input [1]:

- vk_x = gamma_abc[0] + 1 * gamma_abc[1]

2. The Pairing Check

solidity

```
if(!Pairing.pairingProd4(

    proof.a, proof.b,          // e(A, B)

    Pairing.negate(vk_x), vk.gamma,  // e(-vk_x, γ)

    Pairing.negate(proof.c), vk.delta, // e(-C, δ)

    Pairing.negate(vk.alpha), vk.beta)) return 1; // e(-α, β)
```

This checks: **e(A, B) · e(-vk_x, γ) · e(-C, δ) · e(-α, β) = 1**

Which is equivalent to: **e(A, B) = e(α, β) · e(vk_x, γ) · e(C, δ)**

3. Mathematical Foundation

The pairing equation verifies that:

A = α + ∑(witness_i * u_i(x)) + r * δ

B = β + ∑(witness_i * v_i(x)) + s * δ

C = ∑(witness_i * (β*u_i(x) + α*v_i(x) + w_i(x))) + h(x)*t(x) + s*A + r*B - r*s*δ

Where the pairing properties ensure these relationships hold.

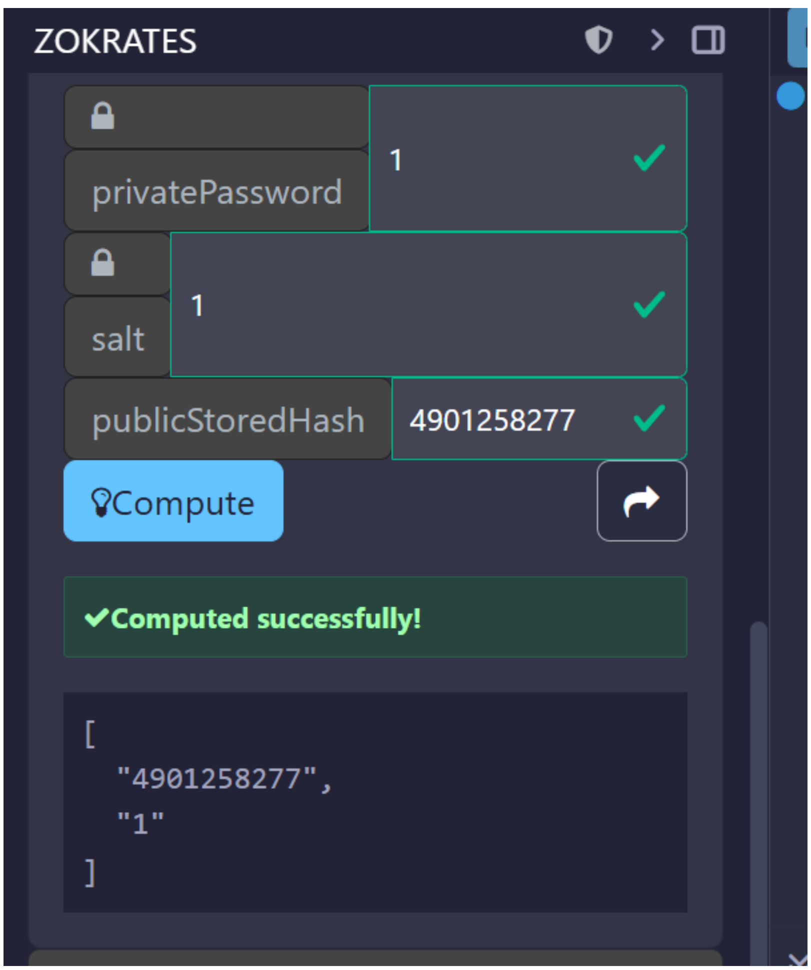
User Password Authentication Program

Here user create password to log in to the system. Here user able to select salt and privatePassword to generate his hash password which is used to log in to they system. Here is proving he knows actually salt and privatePassword when logging . Here used zksnark to prove that hash password came across his salt and privatePassword without showing those.

This is related Zokrates program

```
def main(private field privatePassword, private field salt, field publicStoredHash) -> (field, field) {
   // Field modulus (using p-1 since p itself might cause issues)
   field p = 21888242871839275222246405745257275088548364400416034343698204186575808495616;

   // Use coefficients that are multiples of modulus plus something
   field coeff1 = p + 2654435761;  // p + a
   field coeff2 = 2 * p + 2246822519; // 2p + b

   // computedHash = (privatePassword * (p+a) + salt * (2p+b)) mod p
   // = (privatePassword*a + salt*b) mod p

   field computedHash = privatePassword * coeff1 + salt * coeff2;
   field isValid = computedHash == publicStoredHash ? 1 : 0;

   return (computedHash, isValid);
}
```

For the simplicity, lets think user select salt = 1 , publicStoredHash =1 . Then this is publicStoredHash

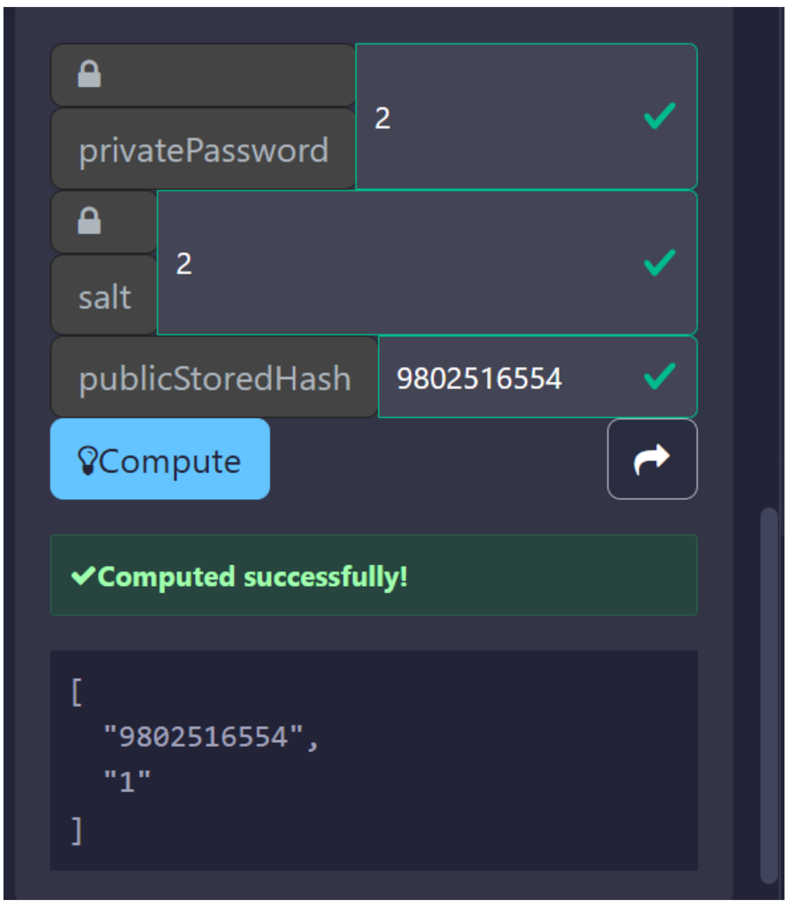


Here is generated verifier.sol contract

This is the user1 proof (salt = 1 , publicStoredHash =1)

[["0x0c4628a9f68307b8ba77e500369da6b573e8998a9d647f509e419364e6c873bf","0x1e97f1f2a3ddd81d633beeba93b87cf3d0e0f42d86f63270667ca300f587c7c1"],[["0x0f3a8b002aae5dc5df146d762f49df874a3f4ae0483e1371bc9baafdd33febbe","0x0e3bf6a6ed9775ae8faa520ff35fdbe3a114fd5f9f35f8bae9e1d17a6e705641"],["0x1cf7ae350c426321adf921fb1a418c0cd041580bc0130c414a9694733c8cdaee","0x024397d42abd876115430112117ae8566851b404d2a1f07cf5a7d54e6a1a3e12"]],["0x09c1a1ae22956abcaf313fa18053f2960851696da46ee8fe1e1ed23d200a1de6","0x173df2d3e092f8a4e67cdfd672ea685640e734f33f24d1be9ceaad3c408525fd"]],["0x000000000000000000000000000000000000000000000000000000124234425","0x000000000000000000000000000000000000000000000000000000124234425","0x0000000000000000000000000000000000000000000000000000000000000001"]

User 2 proof. (salt = 2 , publicStoredHash =2)



This is proof for user 2

[["0x234c2417d8da0e5e967cd42f5961b8cfba6baf50e624bb81d7c0f346ff45e756","0x006ca11884bf74ff795947cb50e5fa9cf3928bb30679b6396fdbf5981cb3b43a"],[["0x15d9e1aed6877e5104846a0201a7f78bdb4c76f2a2c9957d26f4f57cbc92e0db","0x150788505ea486d690877d27afeefd726cb24cfc8b420e50fc6d35645e6284d0"],["0x104b1365

307fe33605122dbff497ec4bdafb111534b2ee0c1e901b11fc79eb95","0x2aae435759ad9666284012cf58efbad9492
e24e68212377c6d9304d435ee3529"]],["0x26163c4a6e69d8cc48fbf32e5978909b4870cbb649b6da00e2449129b5a
fcf3e","0x23e0336618eed18f4d045e407c09b4febdf9bca6a06ea5b16546c69311a4f581"]],["0x0000000000000000
00000000000000000000000000000000000000024846884a","0x000000000000000000000000000000000000000
000000000000000024846884a","0x000000000000000000000000000000000000000000000000000000000000001
"]

[Here](#) is relevant password authentication solidity contract**,**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "./verifier.sol";

contract PasswordAuthentication {
  using Pairing for *;

  Verifier public verifier;

  // Store user password hashes
  mapping(address => uint256) public userPasswordHashes;

  // Track login attempts and timestamps
  mapping(address => uint256) public lastSuccessfulLogin;
  mapping(address => uint256) public failedAttempts;

  event UserRegistered(address indexed user, uint256 passwordHash);
  event LoginSuccessful(address indexed user);
  event LoginFailed(address indexed user);

  constructor(address _verifierAddress) {
    verifier = Verifier(_verifierAddress);
  }

  // Register a new user with password hash
  function registerUser(uint256 passwordHash) public {
    require(userPasswordHashes[msg.sender] == 0, "User already registered");
    userPasswordHashes[msg.sender] = passwordHash;
    emit UserRegistered(msg.sender, passwordHash);
  }

  // Login using zkSNARK proof (password remains private)
  function login(
    uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c,
    uint[3] memory input // [publicStoredHash, computedHash, isValid] - 3 inputs!
  ) public returns (bool) {
```

```
    require(userPasswordHashes[msg.sender] != 0, "User not registered");
    require(input[0] == userPasswordHashes[msg.sender], "Invalid stored hash");

    // Create Proof struct in the format expected by the verifier
    Verifier.Proof memory proof = Verifier.Proof(
        Pairing.G1Point(a[0], a[1]),
        Pairing.G2Point([b[0][1], b[0][0]], [b[1][1], b[1][0]]),
        Pairing.G1Point(c[0], c[1])
    );

    // Verify the proof using verifyTx (which expects 3 inputs)
    bool proofValid = verifier.verifyTx(proof, input);

    if (proofValid) {
        // Successful login
        lastSuccessfulLogin[msg.sender] = block.timestamp;
        failedAttempts[msg.sender] = 0;
        emit LoginSuccessful(msg.sender);
        return true;
    } else {
        // Failed login
        failedAttempts[msg.sender]++;
        emit LoginFailed(msg.sender);
        return false;
    }
}

// Alternative login function that accepts the proof as a struct
function loginWithProofStruct(
    Verifier.Proof memory proof,
    uint[3] memory input
) public returns (bool) {
   // require(userPasswordHashes[msg.sender] != 0, "User not registered");
   // require(input[0] == userPasswordHashes[msg.sender], "Invalid stored hash");

    bool proofValid = verifier.verifyTx(proof, input);

    if (proofValid) {
        lastSuccessfulLogin[msg.sender] = block.timestamp;
        failedAttempts[msg.sender] = 0;
        emit LoginSuccessful(msg.sender);
        return true;
    } else {
        failedAttempts[msg.sender]++;
        emit LoginFailed(msg.sender);
        return false;
    }
}
```

```
// Helper function to convert proof components to struct
function createProof(
    uint[2] memory a,
    uint[2][2] memory b,
    uint[2] memory c
) public pure returns (Verifier.Proof memory) {
    return Verifier.Proof(
        Pairing.G1Point(a[0], a[1]),
        Pairing.G2Point([b[0][1], b[0][0]], [b[1][1], b[1][0]]),
        Pairing.G1Point(c[0], c[1])
    );
}

// Secure function that requires recent authentication
function secureAction() public view returns (string memory) {
    require(userPasswordHashes[msg.sender] != 0, "Not registered");
    require(lastSuccessfulLogin[msg.sender] > 0, "Never logged in");
    require(block.timestamp - lastSuccessfulLogin[msg.sender] < 1 hours, "Session expired");

    return "Secure action performed successfully!";
}

// Get user status
function getUserStatus(address user) public view returns (
    bool registered,
    uint256 lastLogin,
    uint256 failedAttemptsCount
) {
    return (
        userPasswordHashes[user] != 0,
        lastSuccessfulLogin[user],
        failedAttempts[user]
    );
}

// Update password hash (requires re-registration)
function updatePasswordHash(uint256 newPasswordHash) public {
    require(userPasswordHashes[msg.sender] != 0, "User not registered");
    userPasswordHashes[msg.sender] = newPasswordHash;
    emit UserRegistered(msg.sender, newPasswordHash);
}

// Check if user is currently authenticated
function isAuthenticated(address user) public view returns (bool) {
    return userPasswordHashes[user] != 0 &&
           lastSuccessfulLogin[user] > 0 &&
           block.timestamp - lastSuccessfulLogin[user] < 1 hours;
}
```

```
    // Reset failed attempts (admin function)
    function resetFailedAttempts(address user) public {
        // In production, add access control
        failedAttempts[user] = 0;
    }

    // Get the verification key (for frontend use)
    function getVerifierAddress() public view returns (address) {
        return address(verifier);
    }
}
```

You need to deploy verifier.sol contract and then use that address for passwordAuthentication.sol contract as constructor argument. Then User 1 and User 2 able to use those proof to login on **loginWithProofStruct** function.