ブロックチェーン公開講座 第18回

ゼロ知識証明その2（実装）

芝野恭平

東京大学大学院工学系研究科技術経営戦略学専攻
ブロックチェーンイノベーション寄付講座
特任研究員
shibano@tmi.t.u-tokyo.ac.jp

Agenda

- Circomとは
- 回路の生成から証明生成，検証，コントラクト用Solidityコードの出力（snarkjs）
- 回路を書いてみよう
  - 掛け算
  - ３乗の掛け算
  - If-else
  - For文
  - 値の範囲チェック
  - ハッシュ関数
- コントラクトで検証してみよう
  - パスワード認証のコントラクトウォレットサンプル
- 演習

※ 本講義でのサンプルコードは講義の演習用に作ったものです．安全性などが担保されていませんので，
実製品に埋め込むことは避けてください．

Circomとは

- https://docs.circom.io/

- 独自のプログラミング言語で処理内容を記述できる.
- snarkjsというJSのライブラリを用いて，コンパイル後のファイルを取り扱うことができ，それを用いてZKのプルーフ生成や検証が可能.
- 検証用のコントラクトのSolidityコードも自動生成してくれる.

- zkSNARKsのプロトコルとして，PlonKにくわえGroth16，FFlonkが使用可能.

```
pragma circom 2.0.0;

include
"../node_modules/circomlib/circuits/eddsamimc.circom";

template Hash ()   {

   signal input in;
   signal output out;

   component hasher;
   hasher = MiMC7(91);
   hasher.x_in <== in;
   hasher.k <== 0;

   out <== hasher.out;

}

component main = Hash();
```

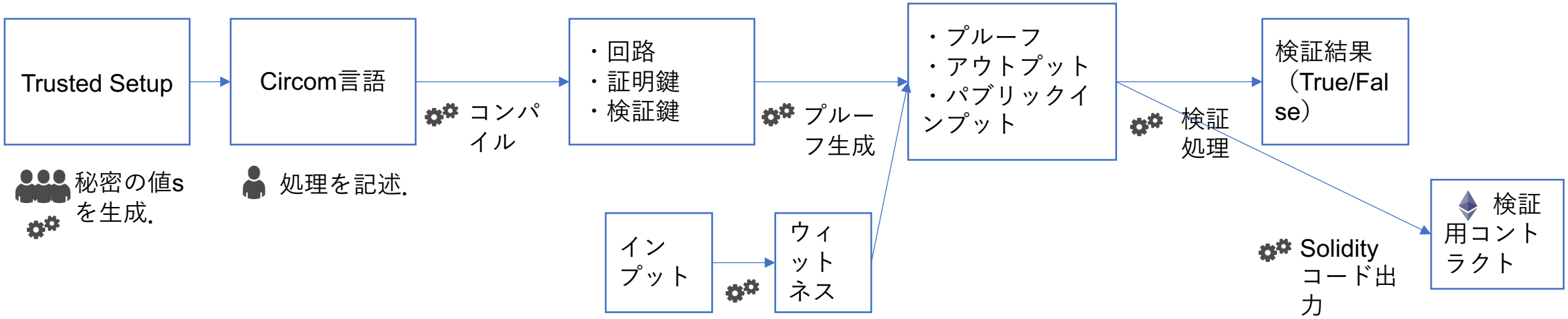
Circomでハッシュ値を計算する例

講義中のコード

- 以下のリポジトリで公開してあります.
- https://github.com/blockchaininnovation/circompractice

- Circomのコードサンプルと，簡易版パスワード認証コントラクトウォレットのサンプルコードが含まれています.
- appディレクトリ
  - Circomのプルーフを生成し，コントラクトにアクセスするためのコード.
  - JavaScriptで記述．Nodejsを使用.
- circomディレクトリ
  - Circomの回路を記述.
  - Circom言語で記述．Nodejsを使用.
- contractディレクトリ
  - 簡易版パスワード認証コントラクトウォレット.
  - Solidityで記述．Foundryのプロジェクト.

Circom/snarkjsを使った回路と証明の流れ



開発用スクリプト

zkkey.sh

zkbuild.sh

zkprove.sh

zkverify.sh

zkcreatesmacon.sh

zkall.sh

※ snarkjsを使っています．一つ一つの手順詳細は以下を参考にしてください．
https://github.com/iden3/snarkjs

Gitからリポジトリを取得し，Circomの回路生成から検証まで

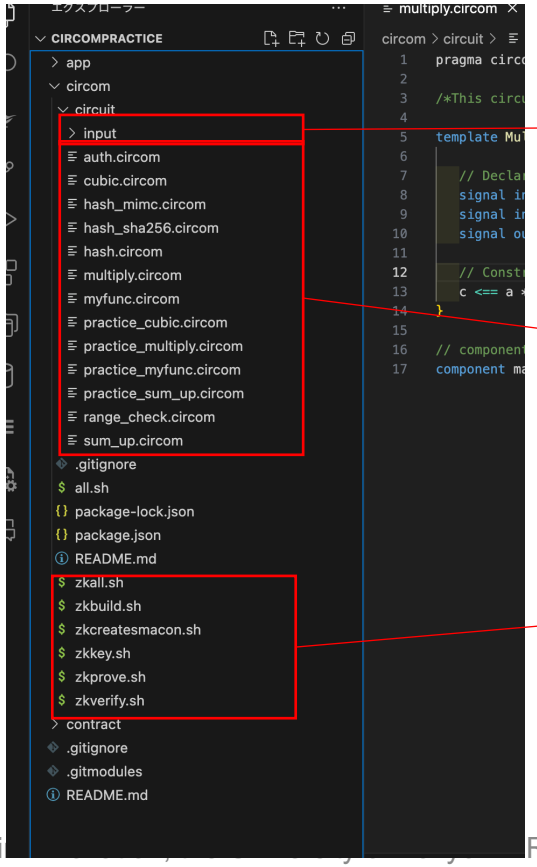
Git clone

```
~$ cd git
~/git$ git clone https://github.com/blockchaininnovation/circompractice.git
```

Vscodeで取得したcircompracticeを開く

```
~/git$ code circompractice/
```

circomディレクトリを開きましょう



回路への入力が入っているデータ．

回路．Circomファイル．
今回の講義で解説する回路のソースコードになります．

各種開発用スクリプト．
回路生成したり，プルーフ生成をしたりするためのスクリプトです．
※ トラステッドセットアップなど簡略化されています．実運用の利用には注意ください．

Gitからリポジトリを取得し，Circomの回路生成から検証まで

Vscode内のターミナルを起動.
Circomディレクトリに移動し，必要なライブラリのインストール.

```
~/git/circompractice$ cd circom/
~/git/circompractice/circom$ npm install
```

multiply.circomに対して，トラステッドセットアップ，回路の生成からプルーフ生成，検証，コントラクト出力のすべてをzkall.shを使用して実行．以下のコマンドを実行：数分かかります.

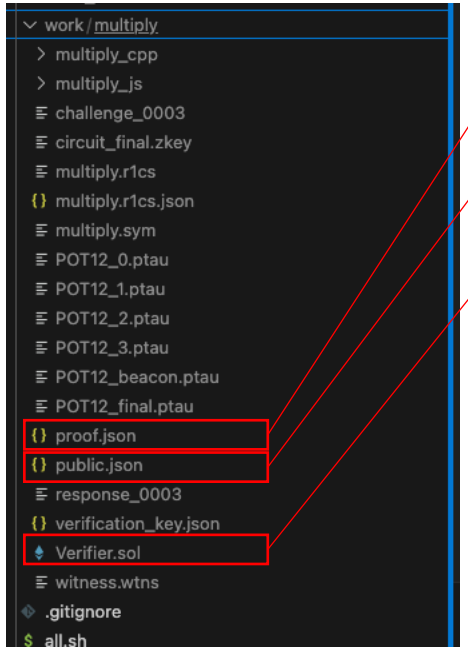
```
~/git/circompractice/circom$ ./zkall.sh multiply 12
1. Start a new powers of tau ceremony
[DEBUG] snarkJS: Calculating First Challenge Hash
[DEBUG] snarkJS: Calculate Initial Hash: tauG1
[DEBUG] snarkJS: Calculate Initial Hash: tauG2
・・・
```

※ ここでは，回路ファイルとしてcircom/circuit/multiply.circomを，プルーフ生成時の入力値としては，circom/circuit/input/multiply.jsonを使用しています.

zkall.shは以下のすべてを実行．2つ目の引数はトラステッドセットアップ時の乱数生成のときの最大制約数2^nのnを指定.
→ 大きな回路ではこの値を大きくする必要がある.
zkkey.sh ・・・ トラステッドセットアップ．コミットメント生成に必要な乱数値を生成.
zkbuild.sh ・・・ 回路の出力．証明鍵と検証鍵も同時に生成する.
zkprove.sh ・・・ インプット値を入力，証明鍵を用いてプルーフを生成する.
zkverify.sh ・・・ 検証鍵でプルーフを検証.
zkcreatesmacon.sh ・・・ Solidityの検証用コントラクトのソースを出力.

Gitからリポジトリの取得し，Circomの回路生成から検証まで

様々な結果ファイルはworkディレクトリに生成されます．各種ファイルを見てみる．

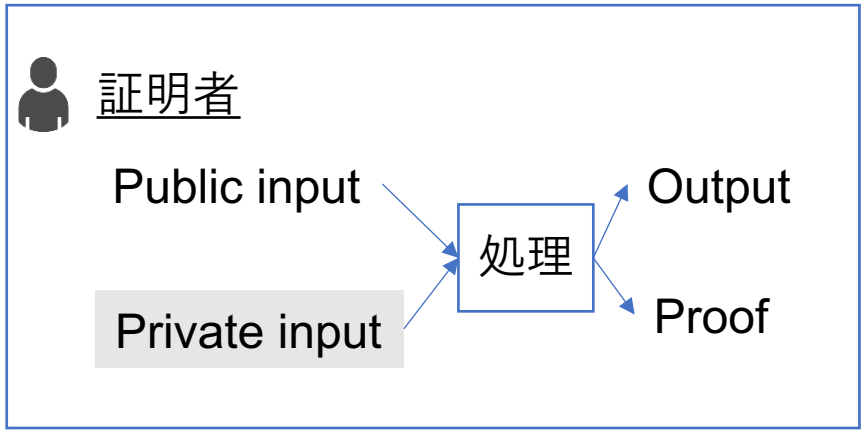


プルーフ

パブリック値．パブリックインプットとアウトプット．

検証用のコントラクトのソース．

各種ファイルと処理のまとめ



- 回路（処理）：
  circom/circuit 内のcircomファイル
- インプット（Public input, Private input)：
  circom/circuit/input内のjsonファイル
- 出力(Output, Public input)：
  circom/work/回路名/public.json
- プルーフ（Proof）：
  circom/work/回路名/proof.json

Circom回路コードの例：掛け算

- circom/circuit/multiply.circom　　入力値a, bをかけた結果を出力する回路

```
pragma circom 2.0.0;

template Multiplier2 () {
   signal input a;
   signal input b;
   signal output c;

   c <== a * b;
}

component main {public [a]} = Multiplier2();
```

```
$ ./zkall.sh multiply 12
```

インプットの値は「input」で宣言.
制約に関する入出力値は「signal」で宣言する.
一方で制約に関係しない変数は「var」で宣言.
※ signalは一度値を入れたらその後で書き換え不可.

アウトプットは「output」で宣言.

最終的にこのシグナル（変数）に
値を代入して出力値を決める.

mainがついているここが実行される.
上で作ったMultiplier2()を実行している.
インプットのうち，パブリックインプットにするものは実行時にここで指定する.
上記の例では変数a, bの2つある入力のうちのaがパブリックインプット，bがプライベートインプット.

Circom回路コードの例：３乗の掛け算

- circom/circuit/cubic.circom　　$x^3 + x + 5$ を計算する回路.

```
pragma circom 2.0.0;

template Cubic() {
    signal input in;
    signal output out;

    // x^3 + x + 5を出力
    signal t;
    t <== in * in;
    out <== t * in + in + 5;
}

component main = Cubic();
```

掛け算は1回までしかできません.
2回以上掛け算を行う場合には違うシグナルを使って計算する必要があります.

※ ためしにsignal tなしにして，outに一気に3乗した結果を入れてみましょう.
エラーメッセージが出力されます.

※ コードを修正してビルドする際は，zkbuild.shの実行だけするのが便利です.

```
$ ./zkall.sh cubic 12
```

Circom回路コードの例：If-else

- circom/circuit/myfunc.circom

```
pragma circom 2.0.0;
(インクルード省略)

template MyFunc() {
    signal input w;
    signal input a;
    signal input b;
    signal output out;

    component eq = IsEqual();
    ise.in[0] <== w;
    ise.in[1] <== 1;

    signal t1;
    t1 <== a * (b + 3);

    signal t2;
    t2 <== eq.out * t1;
    out <== t2 + (1- eq.out) * (a + b);
}

component main = MyFunc();
```

右のmyfunc()を回路で実装したもの

```
function myfunc(w, a, b){
    if (w == 1)
        return a * (b + 3);
     else
        return a + b;
}
```

```
$ ./zkall.sh myfunc 12
```

シグナルに対する比較など判定操作はこのようにコンポーネントを使用する．

2次制約にひっかからないように，途中でシグナルを追加して使用．2乗はもちろん，1つの式中に掛け算が2回以上でてくるとだめ．

動的な結果出力ができない（if else文を使ってoutにいれる値を変えるのはだめ）ので，eqの値と結果をかけ合わせた値の和でif-elseを表現している．

$$eq(a * (b + 3)) + (1 - eq)(a + b)$$

Circom回路コードの例：For文

- circom/circuit/sum_up.circom

```
pragma circom 2.0.0;
(インクルード省略)

template SumUp(n) {
    signal input in[n];
    signal output out;

    signal sums[n+1];
    sums[0] <== 0;
    for(var i = 0; i < n; i++){
      sums[i+1] <== sums[i] + in[i];
    }
    out <== sums[n];
}

component main = SumUp(4);
```

要素数nである配列を入れ，その値の和を取る．

```
$ ./zkall.sh sum_up 12
```

Signalであるsumsとしてn+1個確保している．
本ケースでは1つのvarでも代替可能．

For文の内部で何かしら処理をする場合は，各値はsignalとして処理を行う必要があるため，ここではループの回数分のsignalを宣言して使っている．
後の演習で活用．

Circom回路コードの例：値の範囲チェック

- circom/circuit/range_check.circom

```
pragma circom 2.0.0;
(インクルード省略)

template RangeCheck() {
    signal input in;
    signal input lowerBound;
    signal input upperBound;
    signal output out;

    component le1 = LessEqThan(252);
    component le2 = LessEqThan(252);

    le1.in[0] <== lowerBound;
    le1.in[1] <== in;

    le2.in[0] <== in;
    le2.in[1] <== upperBound;

    out <== le1.out * le2.out;
}

component main = RangeCheck();
```

```
$ ./zkall.sh range_check 12
```

lowerBound <= in
と
In <= upperBound
を同時に満たしているかをチェック

Circom回路コードの例：ハッシュ関数: sha256

- circom/circuit/hash_sha256.circom

```
pragma circom 2.0.0;
(インクルード省略)

template HashSha256(size) {

   signal input in[size];
   signal output out[256];

   component sha = Sha256(size);
   sha.in <== in;

   out <== sha.out;
}

component main = HashSha256(16);
```

```
$ ./zkall.sh hash_sha256 18
```

制約数が多い

入力も出力もbitの配列.
Bit単位での演算が必要になっているため.

入出力がバイト列等の場合，適宜変換する必要がある.

処理は結構時間がかかる.

Circom回路コードの例：ハッシュ関数: mimc

- circom/circuit/hash_mimc.circom

```
pragma circom 2.0.0;
(インクルード省略)

template Hash () {

   signal input in;
   signal output out;

   component hasher;
   hasher = MiMC7(91);
   hasher.x_in <== in;
   hasher.k <== 0;

   out <== hasher.out;
}


component main = Hash();
```

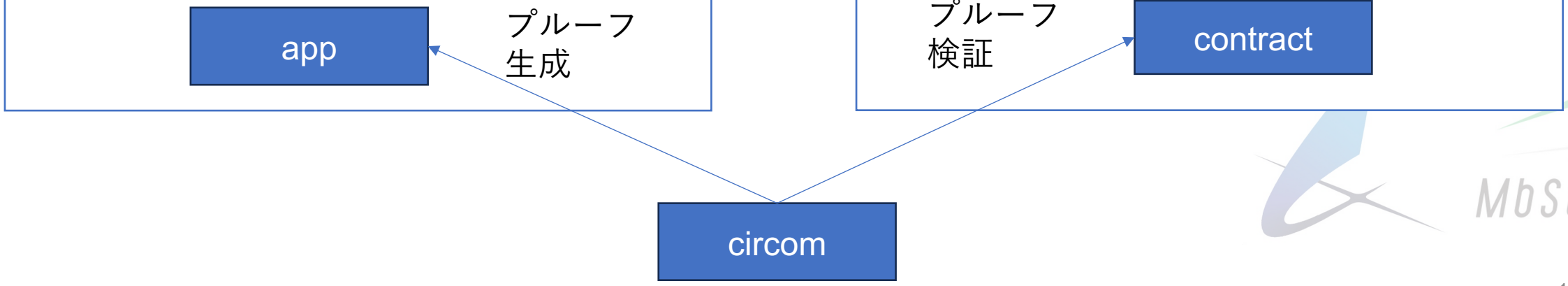
```
$ ./zkall.sh hash_mimc 12
```

ZKフレンドリーなハッシュ関数の一つであるMiMC.
　他にもPoseidonハッシュが有名.
処理時間はSHA256に比べて早い.
入出力は有限体上の数，即ち正の整数.

簡易版コントラクトウォレットを構築・動作させてみよう

アプリケーション
・ウォレット入りクライアントアプリケーション
・EthereumのTxを生成
  ・コントラクトの関数実行
  ・送金指示（どのアドレスに何Eth）
・ZKのプルーフ生成
  ・アドレス（Public input）
  ・パスワード（Private input）
  ・ハッシュ値（Output）

コントラクト
・ZK検証ロジック
・アドレスごとの残高管理
・パスワードのハッシュ値で認証



簡易版コントラクトウォレットを構築・動作させてみよう

- Circom回路

```
pragma circom 2.0.0;

include "./hash.circom";

template UserAuthentication () {

   signal input useraddress;
   signal input password;
   signal output passwordhash;

   component hash;
   hash = Hash();
   hash.in <== password;
   passwordhash <== hash.out;
}


component main {public [useraddress]} = UserAuthentication();
```

自分のアドレスを入れて，それに紐づくパスワードのハッシュを出力する．

アドレスはパブリックインプットで，パスワードはプライベートインプット．

アドレスは，処理中では特に何も使用されていないがパスワードハッシュとの紐づけのため入力させている．

ハッシュはMiMCを利用．
すなわち，ここでのパスワードで入力，ハッシュ値として出力される値は数値．

簡易版コントラクトウォレットを構築・動作させてみよう

- コントラクト

```
pragma solidity 0.8.19;

. . .
contract ContractWallet {
. . .
    function registerPasswdHash(uint passwd_hash) public {}

    function deposit() public payable {}

    function transfer(
        address payable send_to,
        uint amount,
        uint256[24] calldata _proof,
        uint256[2] calldata _pubSignals
    ) public {}

}
```

自分のアドレスの認証用のパスワードハッシュ値を登録

自分のアドレスのコントラクトウォレットへのデポジット.
デポジットの総量が，自分がこのコントラクトウォレットから送金できる総金額になる.

コントラクトウォレットからの送金.
自分のアドレスと，パスワードハッシュが出力されているZKPのプルーフで認証.

簡易版コントラクトウォレットを構築・動作させてみよう

- 前までのサンプルと同様にzkall.shを使用する.

```
$ ./zkall.sh auth 12
```

- Verifier.sol
- がworkディレクトリ中に出力されていることを確認.

簡易版コントラクトウォレットを構築・動作させてみよう

TextDAOのときと同様にfoundryを使用．anvil（ローカルのブロックチェーン）にデプロイする．
anvil起動．

```
$ anvil
```

.envを準備．
Contractディレクトリ内の.env.sampleをコピーして同じディレクトリ内に.envファイルを作る．
PRIVATE_KEY=
のところに，anvilのプロンプトで表示されているアカウントの一つの秘密鍵を入力．

Circomで生成されたVerifier.solをcontract/src内にコピーする．

```
~/git/circompractice$ cd contract/
~/git/circompractice/contract$ cp ../circom/work/auth/Verifier.sol src/
```

簡易版コントラクトウォレットを構築・動作させてみよう

Verifier.solのコード中の以下をコメントアウト．（Hardhat用のコードで，今回はFoundryを使っているためエラーになります．

```
// import "hardhat/console.sol";
```

ビルド

```
~/git/circompractice/contract$ forge build
```

anvilにデプロイ

```
~/git/circompractice/contract$ forge script script/Deployment.s.sol --rpc-url http://127.0.0.1:8545 --broadcast
```

デプロイに成功すると.envファイルにデプロイしたコントラクトアドレスが追記されている．

```
CONTRACT_WALLET_ADDR=0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
```

これをアプリ側に指定します（次ページ）．

簡易版コントラクトウォレットを構築・動作させてみよう

appディレクトリに移動後，必要なモジュールのインストール．（別ターミナルでの作業が便利です

```
~/git/circompractice$ cd app/
~/git/circompractice/app$ npm install
```

ブロックチェーンやコントラクトアドレス，秘密鍵を指定する設定ファイルconfig.jsonを編集：

config.sample.jsonを同じディレクトリ内にコピーしてconfig.jsonを作成．
Anvilのコンソールで出力されている秘密鍵，先ほどデプロイ時に.envに出力されたコントラクトアドレスを入力．

そして，src内のcontractwallet.jsを実行していく．

簡易版コントラクトウォレットを構築・動作させてみよう

ファイル下部のコメントアウト部を適宜編集.
（いくつかの関数が実行されるようになっていて，必要なものだけコメントアウトを解除して実行します.

まずは，コントラクトウォレットに100ETHデポジットする．以下をコメントアウト解除.

```
depositEther("100");
```

contractwallet.jsを実行する.

```
~/git/circompractice/app$ node src/contractwallet.js
Balance of 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 9699.989809269910370455 ETH
Balance of 0x70997970C51812dc3A010C7d01b50e0d17dc79C8: 10030.0 ETH
Deposit transaction sent: 0x3a6755982316b7f17affaf1e292b76d969ec06a4c0b261bf82937fed4ef9e627
Deposit confirmed in block: 24
```

簡易版コントラクトウォレットを構築・動作させてみよう

コントラクトウォレット内の残高が増えているかチェック．以下を同じようにコメントアウト解除し，実行．
逆に前のページで実行した関数の行はコメントアウトする．

```
getTotalBalance();
```

contractwallet.jsを実行する．

```
~/git/circompractice/app$ node src/contractwallet.js
Balance of 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 9599.989763548306814615 ETH
Balance of 0x70997970C51812dc3A010C7d01b50e0d17dc79C8: 10030.0 ETH
getTotalBalance() result: 10000000000000000000
```

残高が10000000000000000000weiになっていることが確認できる．
また，送金元アドレスの残高が100ETH減っていることも確認できる．

簡易版コントラクトウォレットを構築・動作させてみよう

認証用のパスワードハッシュを登録.

```
registerPasswdHash("962799119891586450548332532812346681384086725570082285861245066955930212388 6");
```

入力するハッシュ値は，public.jsonの1個目の値.

contractwallet.jsを実行する.

```
~/git/circompractice/app$ node src/contractwallet.js
Balance of 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 9599.989763548306814615 ETH
Balance of 0x70997970C51812dc3A010C7d01b50e0d17dc79C8: 10030.0 ETH
RegisterPasswdHash transaction sent: 0x41c8180354e2bc7f30cb38d69def0108aeee15f3c354402a05a65891352eeecc
RegisterPasswdHash confirmed in block: 25
```

簡易版コントラクトウォレットを構築・動作させてみよう

snarkjsから生成されたプルーフを用いて送金処理.

```
executeTransferByFile("0x70997970C51812dc3A010C7d01b50e0d17dc79C8", "10", "../circom/work/auth/proof.json",
"../circom/work/auth/public.json");
```

contractwallet.jsを実行する.

```
~/git/circompractice/app$ node src/contractwallet.js
Balance of 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 9599.98971759880503149 ETH
Balance of 0x70997970C51812dc3A010C7d01b50e0d17dc79C8: 10030.0 ETH
s: 15493ff9bce27b8b13a8afed532ab2282229e118ab2ec4ef4ba4c9fcf698f16e
s: 000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266
. . .
publicSignalsForContract: 2
962799119891586450548332532812346681384086725570082285861245066955930212388 6,13908492957860717682763809502386750
83608645509734
Transfer transaction sent: 0x8109d31eb6252e4f9370b960c773be59d7989e424b6568b8c93c368ba11a9f1f
Transfer confirmed in block: 26
```

簡易版コントラクトウォレットを構築・動作させてみよう

コントラクトウォレット内の残高と2番目のEOAアカウントの残高に変化があるかチェック.

```
getTotalBalance();
```

contractwallet.jsを実行する.

```
~/git/circompractice/app$ node src/contractwallet.js
Balance of 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266: 9599.989343201261659418 ETH
Balance of 0x70997970C51812dc3A010C7d01b50e0d17dc79C8: 10040.0 ETH
getTotalBalance() result: 9000000000000000000
```

残高が9000000000000000000weiになって（100ETHから10eth減っている）
2番目のアドレスの残高が10ETH増えている.

簡易版コントラクトウォレットを構築・動作させてみよう

同じプルーフを持って再送金できるか確認する

```
executeTransferByFile("0x70997970C51812dc3A010C7d01b50e0d17dc79C8", "10", "../circom/work/auth/proof.json",
"../circom/work/auth/public.json");
```

contractwallet.jsを実行する.

```
$ node src/contractwallet.js
省略
 shortMessage: 'transaction execution reverted'
```

エラーが生じていて送金できないことが確認できる.

簡易版コントラクトウォレットを構築・動作させてみよう

プルーフを改めて生成しなおして，送金できるか確認

```
~/git/circompractice/circom$ ./zkprove.sh auth auth
```

```
executeTransferByFile("0x70997970C51812dc3A010C7d01b50e0d17dc79C8", "10", "../circom/work/auth/proof.json",
"../circom/work/auth/public.json");
```

contractwallet.jsを実行する.

```
$ node src/contractwallet.js
省略
Transfer transaction sent: 0xff60618d16bc5f085ab85fe9221d21d93e898b51fe78d24008a66e1ab54ce78e
Transfer confirmed in block: 28
```

送金できることが確認できる.

演習

- 掛け算
  - 入力値a, bに対して$a * b + 2$ した値を出力する回路を作ってください.
  - practice_multiply.circomを編集して， practice_input.jsonを入力値とし解答してください.
- ３乗の掛け算
  - 入力値xに対して$x^4 + 12x + 1$した値を出力する回路を作ってください.
  - practice_cube.circomを編集して， practice_cube.jsonを入力値として解答してください.
- If-else
  - 右のmyfunc(w, a, b)で表現される処理の回路を作ってください.
  - practice_myfunc.circomを編集してpractice_myfunc.jsonを入力値として解答してください.
  - ヒント：不等号はcircomlibのcomparators.circom内から適当なものを使用してください.
- For文
  - 右のsumUpTo()で表現される処理の回路を作ってください.
  - practice_sum_up.circomを編集してpractice_sum_up.jsonを入力値として解答してください.

```
function myfunc(w, a, b) {
    if (w > 10)
        return a + b;
    else
        return a - b;
}
```

```
N = 4;
function sumUpTo(arr) {
    let count_less_than_3 = 0;
    for (let i = 0; i < N; i++) {
        if (arr[i] < 3){
            count_less_than_3 += 1;
        }
    }
    return count_less_than_3;
}
```

演習の解答例

- answerブランチに解答例を掲載してあります.
- https://github.com/blockchaininnovation/circompractice/tree/answer

まとめ

- Circomの基本的な使い方について学んだ
  - 回路への変換から証明生成，検証までをsnarkjsを使用してのやり方.
  - 通常のプログラミング言語との異なる特徴.

- 簡易的なコントラクトウォレットアプリケーションを実行した
  - 適切なプルーフとともにコントラクトが実行された場合にのみ送金が可能であることがわかった.

- 本スライドの著作権は，東京大学ブロックチェーンイノベーション寄付講座に帰属しています．自己の学習用途以外の使用，無断転載・改変等は禁止します．