Smart Contract:

Generador de Números Pseudo-Aleatorios

Gabriel Fernando Martín Fernández y Aarón Rial Gonzalez

1 *Problema y Objetivo del Proyecto*

El uso de números aleatorios y pseudo-aleatorios es cada vez más notorio, siendo utilizados desde en usos tan mundanos como juegos de azar, hasta ser factores importantes de criptografía, animación o para tests de errores. Su utilidad es innegable, pero la generación de estos números debe contar con ciertos factores: impredictibilidad e incontrolabilidad. Es decir, la creación de números aleatorios o pseudo-aleatorios debería ser afectada lo menos posible, mientras que sea lo más aleatorio posible. La maximización de éstos parámetros es lo que suele llevar a causar problemas cuando se diseña un método para generar números aleatorios, y es uno de los objetivos de éste proyecto.

La implementación de un generador de números aleatorios en la blockchain acarrea nuevos inconvenientes para la maximización de los parámetros anteriormente definidos. Debido a la naturaleza descentralizada de la blockchain, ésta debería ser ideal para éste tipo de procesos, donde la capacidad de demostrar la aleatoriedad del proceso sería muy importante, pero la blockchain también es colaborativa por naturaleza, por lo que los distintos nodos que participen en la generación del número aleatorio tendrán la capacidad de alterar y (dependiendo del generador de números) controlar el número aleatorio generado, causando que deje de ser completamente aleatorio.

El objetivo de éste proyecto será diseñar un smart contract que sirva para generar números pseudo-aleatorios en la blockchain, evitando o minimizando al mismo tiempo la capacidad de los nodos de modificar la generación del número.

2 *Concepto del Proyecto*

El proyecto ha sido diseñado inspirado por otro proyecto con objetivo similar: Randao (randao.org). Siguiendo su ejemplo, éste contrato funciona por etapas, en las cuales los distintos nodos participantes se someterán a un proceso de registro, compromiso y revelación mediante los cuales se generará el número aleatorio. En esta sección la lógica tras el proceso será explicada, sin utilizar el análisis del código.

1. En primera instancia, los nodos participantes y observadores se registran como tal.

2. La siguiente fase del proceso es la fase del compromiso (commit) en la que un hash es entregado por cada nodo, el cual es registrado como el commit de dicho nodo. Dicho hash se obtiene de un número secreto elegido previamente por cada nodo.

3. El siguiente paso es la fase de la revelación, en la cual cada nodo revela el número secreto mencionado en el anterior apartado. Una vez revelado se confirma que el hash aportado en la anterior fase pertenezca al número secreto.

4. Finalmente se genera un número aleatorio obteniendo el hash del conjunto de los números secretos.

Este proceso tiene una gran ventaja por el hecho de estar implementando en la blockchain: es un proceso completamente accesible y observable. Sin embargo, como ya se ha dicho antes, la blockchain también tiene inconvenientes, los cuales se intentarán circumnavegar en este proyecto. Para reducir el conocimiento previo que tiene el último nodo con respecto a los demás, cada fase tiene un tiempo asignado que actua como delay, evitando que los

resultados se revelen antes de que el último nodo participe (o decida no hacerlo). También se incluyó una pequeña fianza para los nodos participantes, la cual será devuelta si el nodo realiza el proceso correctamente (se podría añadir posteriormente un incentivo), pero que no será devuelta si el nodo decide comportarse de manera maliciosa para con el proceso.

3 *Diseño del Código*

Ahora que la lógica del proceso ha sido explicada, se puede comenzar a explicar la implementación en blockchain con mayor detalle.

3.1 *Inicio del Código*

Las primeras lineas del código están dedicadas a la versión el establecimiento del smart contract, como es habitual. De éstas lo único destacable es que el programa puede ser compilado con todas las versiones a partir de la 0.8.19.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;

contract SimpleRandao{
```

3.2 *Estructuras*

Una vez establecido el smart contract, el primer paso del código consta de definir tres estructuras diferentes: *Participant*, *Observer* y *Round*. De estas tres estructuras, la más compleja es *Round*, la cual almacena variables, arrays y mappings. Estas estructuras serán utilizadas repetidamente durante el código para almacenar la información de manera más ordenada y eficiente.

```
    //STRUCTS
    struct Participant {
        uint256    secret;
        bytes32    commitment;
        bool       rewardable;
    }

    struct Observer {
        bool rewardable;
    }

    struct Round {
        uint32     startTime;
        uint32     commitStartTime;
        uint32     revealStartTime;
        uint256    random;
        uint256    bounty;
        bool       finished;
        bool       generatedValidRNG;

        Participant [] participants;
        Observer [] observers;

        mapping (address => uint) addrToParticipantIndex;
        mapping (uint => address) participantIndexToAddress;
```

```
26          mapping (address => uint) addrToObserverIndex;
27          mapping (uint => address) observerIndexToAddress;
28      }
```

3.3 *Variables Globales*

El siguiente paso será establecer variables globales. Estas variables son, en su mayoría, fácilmente ajustables para poder modificar ciertos parámetros del programa.

```
1       //GLOBAL VARIABLES
2       address private owner;
3       uint256 private roundCount;
4       Round[] private rounds;
5       bool private payment;
6
7       uint32 private registerPhaseDuration = 10 seconds;
8       uint32 private commitPhaseDuration = 10 seconds;
9       uint32 private revealPhaseDuration = 10 seconds;
10      uint32 private cooldownTime = 30 seconds;
11      uint32 private minimumParticipants = 1;
12      uint32 private minimumCommits = 1;
13      uint32 private minimumRevealers = 1;
14
15      uint256 private observerFee = 0 ether;
16      uint256 private participantFee = 0 ether;
17      uint256 private newRoundFee = 0 ether; //must be greater or equal
            than participantFee
18
19      uint16 private currentPhase;
```

3.4 *Constructor*

Después se define una función constructora, la cual se ejecutará al inicio del programa y que establece al "propietario" o *owner* como la persona que inicializa el programa (*msg.sender*).

```
1       //CONSTRUCTOR
2       constructor() {
3           owner = payable(msg.sender);
4           }
```

3.5 *Eventos*

Se definen tres eventos a continuación, que por orden se emiten cuando hay un cambio de fase, se distribuye un pago o se genera un nuevo número aleatorio (respectivamente).

```
1       //EVENTS
2       event phaseChangedTo(string currentPhase);
3       event paymentDistributed(address receiver, uint value);
4       event newRandomValueGenerated();
```

3.6 *Modificadores*

En las siguientes lineas se definen cinco modificadores de funciones distintos. El propósito de cada uno es bastante claro:

1. El primero llama a la función *updateRoundStatus* automáticamente, evitando que el cliente tenga que actualizar el estado de la ronda manualmente.

2. El segundo confirma si la actual fase es una fase válida haciendo uso de un array con valores 1 (válido) y 0 (no válido).

3. El tercer modificador evalua el nivel del usuario que intenta llamar a una función, siendo el valor 0 para usuarios que no están interactuando con el código, 1 para los observadores, 2 para los nodos participantes y finalmente 3 para el propietario o *owner*.

4. El cuarto confirma que el usuario que llama a la función modificada no esté previamente registrado.

5. El quinto modificador está creado para poder crear funciones de pago privadas (ya que para incluir el modificador estándar de *payable* la función debe ser pública o externa).

```
1     //MODIFIERS
2     modifier update(){
3         updateRoundStatus();
4         require(true);
5         _;
6     }
7
8     modifier validPhase(uint8 [5] memory _validPhases){
9         bool result = false;
10        uint8 valid = _validPhases[currentPhase];
11        if (valid == 1){result = true;}
12        require (result);
13        _;
14    }
15
16    modifier validUser(uint8 _privilegeLevel){
17        uint16 clearance = 0;
18        if (owner == msg.sender) {clearance=3;}
19        if (roundCount>0 && clearance == 0) {
20            Round storage currentRound = rounds[roundCount-1];
21            uint pIndex = currentRound.addrToParticipantIndex[msg.sender
                  ];
22            uint oIndex = currentRound.addrToObserverIndex[msg.sender];
23            if (currentRound.participants[pIndex-1].rewardable && pIndex
                  >0) {clearance=2;}
24            else if (currentRound.observers[oIndex-1].rewardable &&
                  oIndex>0) {clearance=1;}
25        }
26        require (clearance >= _privilegeLevel);
27        _;
28    }
29
30    modifier notRegistered() {
```

```
31          require(roundCount>0);
32          Round storage currentRound = rounds[roundCount-1];
33          require (currentRound.addrToParticipantIndex[msg.sender] == 0 &&
                currentRound.addrToObserverIndex[msg.sender] == 0);
34          _;
35      }
36
37      modifier privatePaymentOnly(){
38          require(payment);
39          payment = false;
40          _;
41      }
```

3.7 *Funciones y Métodos Auxiliares*

La siguiente sección del código sirve para definir funciones auxiliares.
La primera función en ser definida sirve para confirmar en que fase se encuentra el proceso.

```
1       //AUXILIAR METHODS
2       function strCurrentPhase() private view returns (string memory ) {
3           if (currentPhase == 0) {return "standby";}
4           if (currentPhase == 1) {return "register";}
5           if (currentPhase == 2) {return "commit";}
6           if (currentPhase == 3) {return "reveal";}
7           if (currentPhase == 4) {return "finished";}
8           else {return "invalid";}
9       }
```

La siguiente función es la encargada de comparar el hash con el numero revelado (3^{er} paso de la lógica del programa).

```
1       function validateSecret(uint256 _secret, bytes32 _commit) private
            pure returns (bool) {
2           return (keccak256(abi.encode(_secret)) == _commit);
3       }
```

La función *generateRandom*() es la encargada de generar el número aleatorio al final del proceso. En ella se crea un vector *secrets* en el que se acumulan los números secretos de los participantes que no hayan fallado o cambiado el número y luego se genera el hash del vector. La variable *generatedValidRNG* del struct *currentRound* se iguala a *true* significando que se generó con éxito el valor aleatorio y se emite el evento de *newRandomValueGenerated*(), para luego ejecutar la función que distribuye las fianzas e incentivos.

```
1       function generateRandom() private {
2           Round storage currentRound = rounds[roundCount-1];
3           uint256 [] memory secrets = new uint256 [] (currentRound.
                participants.length) ;
4           for (uint i = 0; i < currentRound.participants.length; i++){
5               if (currentRound.participants[i].rewardable){
6                   secrets[i] = (currentRound.participants[i].secret);
7                   }
8           }
9           currentRound.random = uint256(keccak256(abi.encodePacked(secrets
                )));
10          currentRound.generatedValidRNG = true;
11          emit newRandomValueGenerated();
```

```
12          distributeBounty();
13      }
```

la próxima función definida es, como se ha dicho en el anterior párrafo, la encargada de distribuir las fianzas y los incentivos. En esta función la fianza o tarifa de los observadores se les será devuelta si el número aleatorio no es generado, mientras que si es generado con éxito, los participantes reciben su parte.

```
1       function distributeBounty() private{
2           Round storage currentRound = rounds[roundCount-1];
3           //uint256 balance = currentRound.bounty; //For more complex
                bounty distributions
4           if (currentRound.generatedValidRNG == false && observerFee > 0){
5               for (uint i = 0; i < currentRound.observers.length; i++){
6                   if (currentRound.observers[i].rewardable){
7                       address payable dest = payable(currentRound.
                            observerIndexToAddress[i+1]);
8                       payment = true;
9                       sendViaCall(dest,observerFee);
10                  }
11              }
12          }
13          if (participantFee == 0) {return;}
14          for (uint i = 0; i < currentRound.participants.length; i++){
15                  if (currentRound.participants[i].rewardable){
16                      address payable dest = payable(currentRound.
                            participantIndexToAddress[i+1]);
17                      payment = true;
18                      sendViaCall(dest,participantFee);
19                  }
20              }
21      }
```

La última función de esta sección del código es *sendViaCall*, la cual es utilizada para poder pagar directamente a una dirección de Ethereum usando los fondos almacenados en el contrato.

```
1       function sendViaCall(address payable _to, uint _value) public
            payable privatePaymentOnly {
2           // Call returns a boolean value indicating success or failure.
3           // This is the current recommended method to use.
4           (bool sent,) = _to.call{value: _value}("");
5           require(sent, "Failed to send Ether");
6           emit paymentDistributed(_to, _value);
7       }
```

3.8 *Funciones de Actualización de Protocolos*

En esta sección se definen las funciones encargadas de actualizar las distintas fases de proceso.
La primera de ellas es una función que tiene el propósito de detectar en que fase del proceso se encuentra el programa y de llamar la función de actualización correspondiente.

```
1       //PROTOCOL UPDATE FUNCTIONS
2       function updateRoundStatus() public {
3           if (currentPhase == 1) {updateRegisterPhase();}
4           if (currentPhase == 2) {updateCommitPhase();}
```

```
        if (currentPhase == 3) {updateRevealPhase();}
        if (currentPhase == 4) {updateFinishingPhase();}
    }
```

Las siguientes funciones de esta sección son las encargadas de actualizar cada una de las fases del proceso: *Registro*, *Compromiso*, *Revelacion* y *Final* (son las funciones llamadas en la función de selección anteriormente definida).

```
    function updateRegisterPhase() private {
        Round storage currentRound = rounds[roundCount-1];
        if (uint32(block.timestamp) - currentRound.startTime >=
            registerPhaseDuration){
            if (currentRound.participants.length < minimumParticipants){
                currentPhase = 0;
                currentRound.finished = true;
                distributeBounty();
                //NOT ENOUGH PARTICIPANTS
            }
            else {
                currentPhase++;
                emit phaseChangedTo(strCurrentPhase());
            }

        }
    }

    function updateCommitPhase() private {
        Round storage currentRound = rounds[roundCount-1];
        if (uint32(block.timestamp) - currentRound.commitStartTime >=
            commitPhaseDuration){
            uint commiters = 0;
            for (uint i = 0; i < currentRound.participants.length; i++){
                if (currentRound.participants[i].commitment != 0){
                    commiters++;}
                else {currentRound.participants[i].rewardable = false;}
            }
            if (commiters < minimumCommits){
                currentPhase = 0;
                currentRound.finished = true;
                distributeBounty();
                //NOT ENOUGH COMMITS
            }
            else {
                currentPhase++;
                emit phaseChangedTo(strCurrentPhase());
            }

        }
    }

    function updateRevealPhase() private {
        Round storage currentRound = rounds[roundCount-1];
        if (uint32(block.timestamp) - currentRound.revealStartTime >=
            revealPhaseDuration){
            uint revealers = 0;
            for (uint i = 0; i < currentRound.participants.length; i++){
                if (currentRound.participants[i].secret != 0){revealers
                    ++;}
```

```
                else {currentRound.participants[i].rewardable = false;}
            }
            if (revealers < minimumRevealers){
                currentPhase = 0;
                currentRound.finished = true;
                distributeBounty();
                //NOT ENOUGH REVEALERS
            }
            else {
                generateRandom();
                currentPhase++;
                emit phaseChangedTo(strCurrentPhase());

            }

        }
    }

    function updateFinishingPhase() private {
        Round storage currentRound = rounds[roundCount-1];
        if (uint32(block.timestamp) - currentRound.revealStartTime >=
            revealPhaseDuration + cooldownTime){
            currentPhase=0;
            currentRound.finished = true;
            emit phaseChangedTo(strCurrentPhase());
            }
    }
```

3.9 *Métodos Principales*

En esta sección se determinan las funciones principales del programa.
La primera función de la sección es la encargada de comenzar una ronda, estableciendo todos los valores iniciales.

```
    //MAIN METHODS
    function startRound() update() validPhase([1,0,0,0,0]) public
        payable {
        require(msg.value == newRoundFee);
        Round storage newRound = rounds.push();
        roundCount++;
        newRound.startTime = uint32(block.timestamp);
        newRound.commitStartTime = newRound.startTime +
            registerPhaseDuration;
        newRound.revealStartTime = newRound.commitStartTime +
            commitPhaseDuration;
        newRound.bounty = newRoundFee;
        newRound.participants.push(Participant(0,0,true));
        newRound.addrToParticipantIndex[msg.sender] = 1;
        newRound.participantIndexToAddress[1] = msg.sender;
        currentPhase = 1;
        emit phaseChangedTo(strCurrentPhase());
    }
```

Las siguientes dos funciones definidas en el código son ambas utilizadas para registrar nodos como observadores y participantes respectivamente. Ambas requieren sus respectivas tarifas o fianzas, y añaden a la estructura *currentRound* los datos de los nuevos

observadores/participantes.

```
    function registerObserver() update() validPhase([0,1,1,1,0])
        notRegistered public payable {
        require(msg.value == observerFee);
        Round storage currentRound = rounds[roundCount-1];
        currentRound.observers.push(Observer(true));
        currentRound.addrToObserverIndex[msg.sender] = currentRound.
            observers.length;
        currentRound.observerIndexToAddress[currentRound.observers.
            length] = msg.sender;
        currentRound.bounty = currentRound.bounty + observerFee;
    }

    function registerParticipant() update() validPhase([0,1,0,0,0])
        notRegistered public payable {
        require(msg.value == participantFee);
        Round storage currentRound = rounds[roundCount-1];
        currentRound.participants.push(Participant(0,0,true));
        currentRound.addrToParticipantIndex[msg.sender] = currentRound.
            participants.length;
        currentRound.participantIndexToAddress[currentRound.participants
            .length] = msg.sender;
        currentRound.bounty = currentRound.bounty + participantFee;
    }
```

La siguiente función es la función *commit*, que como su nombre indica será la encargada de crear el compromiso entre un participante y el hash que ha enviado. Este hash será luego comparado con el número secreto, como ya se explicó anteriormente.

```
    function commit(bytes32 _commit) update() validPhase([0,0,1,0,0])
        validUser(2) public {
        Round storage currentRound = rounds[roundCount-1];
        uint senderIndex = currentRound.addrToParticipantIndex[msg.
            sender] - 1;
        currentRound.participants[senderIndex].commitment = _commit;
    }
```

Finalmente, la última función es la encargada de la fase de revelación del número secreto. En esta función el nodo participante añade su número secreto y es guardado en la estructura *part* como *part.secret*. Después se ejecuta la función de *validateSecret* que fue explicada en la subsección 3.7 Funciones y Métodos Auxiliares.

```
    function reveal(uint256 _secret) update() validPhase([0,0,0,1,0])
        public validUser(2) {
        Round storage currentRound = rounds[roundCount-1];
        uint senderIndex = currentRound.addrToParticipantIndex[msg.
            sender] - 1;
        Participant storage part = currentRound.participants[senderIndex
            ];
        part.secret = _secret;
        if (part.rewardable==false){return;}
        if (validateSecret(part.secret,part.commitment) == false){
            part.rewardable = false;
        }
    }
```

3.10 *Visionados Externos*

En esta sección del código se definen varias funciones, todas ellas con el mismo propósito: Devolver valores en cualquier momento del proceso.
Estas funciones son muy útiles para observar el proceso y poder ver los cambios en cualquier punto de éste. Cada una de ellas devuelve el valor de un elemento del código distinto.

```
1     //EXTERNAL VIEWS
2     function get_observerFee() external view returns (uint256) {return
          observerFee;}
3
4     function get_participantFee() external view returns (uint256) {
          return participantFee;}
5
6     function get_newRoundFee() external view returns (uint256) {return
          newRoundFee;}
7
8     function get_currentPhase() external view returns (uint16) {return
          currentPhase;}
9
10    function get_registerPhaseDuration() external view returns (uint32)
          {return registerPhaseDuration;}
11
12    function get_commitPhaseDuration() external view returns (uint32) {
          return commitPhaseDuration;}
13
14    function get_revealPhaseDuration() external view returns (uint32) {
          return revealPhaseDuration;}
15
16    function get_cooldownTime() external view returns (uint32) {return
          cooldownTime;}
17
18    function getRandom() validPhase([0,0,0,0,1]) validUser(1) external
          view returns (uint) {
19        if (roundCount > 0) {
20            Round storage currentRound = rounds[roundCount-1];
21            if (currentRound.generatedValidRNG) {return currentRound.
                  random;}
22        }
23        return 0;
24    }
25
26    function getLastValidRandom() validPhase([1,1,1,1,0]) validUser(0)
          external view returns (uint) {
27        for (uint i = roundCount-1; i >= 0; i--){
28            if (rounds[i].generatedValidRNG) {return rounds[i].random;}
29        }
30        return 0;
31    }
```

3.11 *Establecer Parámetros de Protocolo*

Finalmente, la última sección del código se reserva a la definición de funciones útiles para el propietario (*owner*).
Las funciones aquí definidas sirven para establecer distintos parámetros del programa,

permitiendo maleabilidad del proceso según requiera el propietario.

```
    //OWNER ONLY, SET PROTOCOL PARAMETERS
    function set_registerPhaseDuration(uint32 _registerPhaseDuration)
       public validPhase([1,0,0,0,1]) validUser(3) {
       registerPhaseDuration = _registerPhaseDuration;}

    function set_commitPhaseDuration(uint32 _commitPhaseDuration) public
        validPhase([1,0,0,0,1]) validUser(3) {commitPhaseDuration =
       _commitPhaseDuration;}

    function set_revealPhaseDuration(uint32 _revealPhaseDuration) public
        validPhase([1,0,0,0,1]) validUser(3) {revealPhaseDuration =
       _revealPhaseDuration;}

    function set_cooldownTime(uint32 _cooldownTime) public validPhase
       ([1,0,0,0,1]) validUser(3) {cooldownTime = _cooldownTime;}

    function set_minimumParticipants(uint32 _minimumParticipants) public
        validPhase([1,0,0,0,1]) validUser(3) {minimumParticipants =
       _minimumParticipants;}

    function set_minimumCommits(uint32 _minimumCommits) public
       validPhase([1,0,0,0,1]) validUser(3) {minimumCommits =
       _minimumCommits;}

    function set_minimumRevealers(uint32 _minimumRevealers) public
       validPhase([1,0,0,0,1]) validUser(3) {minimumRevealers =
       _minimumRevealers;}

    function set_observerFee(uint32 _observerFee) public validPhase
       ([1,0,0,0,1]) validUser(3) {observerFee = _observerFee;}

    function set_participantFee(uint32 _participantFee) public
       validPhase([1,0,0,0,1]) validUser(3) {participantFee =
       _participantFee;}

    function set_newRoundFee(uint32 _newRoundFee) public validPhase
       ([1,0,0,0,1]) validUser(3) { newRoundFee = _newRoundFee;}

}
```

3.12 *Contrato de prueba*

El proyecto incluye un archivo de prueba, llamado *SimpleRandaoTester.sol*, que permite probar el smart contract simulando a otros contratos interactuando con el generador de números pseudo-aleatorios. El tester está diseñado para que genere un número secreto y su hash para la fase de compromiso, teniendo las funciones de honestCommit, en la que se da el hash para el protocolo de compromiso, honestReveal en la que se revela el número secreto adecuado, y el dishonestReveal, en el que se puede probar el error que sucede en caso de que un nodo actue de manera maliciosa y revele otro número. El resto de las funciones son similares a las del contrato original, teniendo el mismo nombre y función (se implementan con interfaces que señalan al contrato original).

4 *Conclusiones*

En resumen, este proyecto pretendía crear una implementación de un generador de numeros pseudo-aleatorios en la blockchain, algo que se ha logrado. El código aquí desarrollado y explicado presenta una opción mejor que la mayoría de posibilidades originales de Solidity, pero también tiene sus imperfecciones. El generador de números no puede ser considerado seguro y esta implementación presenta una serie de soluciones parciales a varios de los requisitos de un generador de números, pero no las solventan de manera definitiva. La generación de aleatoriedad en Blockchain es un tema muy complejo y hay varios tipos de implementaciones que tratan de lograr un equilibrio entre la eficiencia y seguridad del generador de números pseudo-aleatorios. El uso de firmas BLS, por ejemplo, haría nuestra implementación similar a la forma en la que Ethereum maneja la generación de valores aleatorios aunque también aumentaría en gran medida la complejidad del código.