Lab 6 – Bytecode EVM, consumo de gas y programación con Yul

1. Bytecode EVM, consumo de gas

Dado el siguiente contrato:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.4.0;
contract lab6 {
  uint[] arr;
  uint sum;
  function generate(uint n) external {
    for (uint i = 0; i < n; i++) {
        arr.push(i*i);
    }
  }

  function computeSum() external {
    sum = 0;
    for (uint i = 0; i < arr.length; i++) {
      sum = sum + arr[i];
    }
  }
```

1. Genera el código ensamblador del contrato y guárdalo en un fichero de texto.

2. Considera solamente el código de ejecución (runtime code). **Dibuja <u>a mano</u>** el grafo **CFG** de `computeSum()`. Numera los nodos del grafo con las etiquetas generadas por el compilador (`tag 1`, `tag 2`, etc.). Si el compilador no genera etiqueta para ese bloque, utiliza la etiqueta del nodo inmediatamente anterior con un sufijo 'b', 'c', etc. Marca aquellos nodos que terminan con una instrucción `REVERT`, `RETURN`, `STOP` o `INVALID`.

3. Localiza las instrucciones que acceden a *storage*. Indica cuál es el objetivo de cada una de estas instrucciones en el programa. Indica en el CFG en qué nodos aparecen estas instrucciones.

   NOTA: Los comentarios que añade el compilador a la derecha del código en ensamblador son orientativos pero ten en cuenta que pueden ser imprecisos o incluso inducir a errores de interpretación.

4. Ejecuta `computeSum()` con un array de 100 elementos para obtener su **coste de ejecución** en gas (**execution cost**).

   ¿Se podría modificar el código Solidity de `computeSum()` para reducir el número de accesos a *storage*? Utiliza el CFG para proponer una versión mejorada de `computeSum()` **sin utilizar bloques assembly en Yul** con un consumo mínimo de gas. Determina su coste de ejecución en gas para un array de 100 elementos, compáralo con el coste de la versión original y explica el motivo por el que el coste varía. Solo se puede modificar el código de `computeSum()`, y se debe suponer que el array puede contener cualquier contenido (podría contener datos diferentes a los producidos por `generate()`).

   IMPORTANTE: El coste de ejecución puede variar si se aumenta el número de funciones del contrato y si se ejecutan varias transacciones en el mismo contrato desplegado. Para obtener el coste de ejecución de forma precisa, crea otro contrato solo con las funciones `generate(n)` y `computeSum()` y despliégalo de nuevo cada vez que vayas a evaluar el coste de una función, y ejecuta en cada test solamente `generate(n)` y la función a evaluar.

2. Programación con Yul y el *assembly block*

5. Estudia la función `completeMaxAsm` del fichero `09fors.sol` y a partir de ella programa una función Solidity:

```
function maxMinMemory(uint[] memory arr) public pure
returns (uint maxmin) {
  assembly {
    // ...
  }
}
```

   que calcule la distancia entre el máximo y el mínimo del array **utilizando exclusivamente código Yul** dentro de `maxMinMemory`. Para ello, **debes definir una función Yul en un bloque `assembly`:**

```
function fmaxmin (array_pointer) -> maxVal, minVal
```

   que devuelva el máximo y el mínimo del array proporcionado como argumento para después invocarla y calcular la diferencia entre los dos resultados en el mismo bloque Yul.

6. Dado el siguiente contrato:

```
contract lab6ex6 {
  uint[] public arr;

  function generate(uint n) external {
    // Populates the array with some weird small numbers.
    bytes32 b = keccak256("seed");
    for (uint i = 0; i < n; i++) {
      uint8 number = uint8(b[i % 32]);
```

```
        arr.push(number);
      }
    }


    function maxMinStorage() public view returns (uint maxmin){
      //...
    }
  }
```

Escribe el cuerpo de la función `maxMinStorage` que realice el mismo cálculo que el apartado anterior para un array `arr` en *storage*, utilizando exclusivamente código Yul. Para ello, define una función Yul

```
function fmaxmin (slot) -> maxVal, minVal
```

que, dado un slot de storage que corresponde a un array, calcule el máximo y mínimo del array. Para obtener en Yul el slot de una variable de *storage* `arr`, se debe utilizar `arr.slot`. Intenta reducir al máximo el consumo de gas.

7. Crea otro contrato con una implementación simple de `maxMinStorage()` escrita en Solidity y compara el coste de ejecución respecto de la versión en Yul utilizando un array de 100 elementos. Analiza los resultados obtenidos y explica por qué tu implementación en Yul reduce el consumo de gas.