Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

C-EVM: Implementación en C de instrucciones ensamblador EVM

Autor: Virginia Esteban Salguero

Tutor: Guillermo Román Díez

Madrid, Junio de 2020

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: C-EVM: Implementación en C de instrucciones ensamblador EVM
Madrid, Junio de 2020

Autor: Virginia Esteban Salguero

Tutor:
Guillermo Román Díez
Lenguajes y sistemas informáticos e ingeniería de software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Hoy en día la transmisión de información a través de Internet es una realidad, así como el uso de tecnologías como Blockchain [1] que permiten realizar estas transacciones de forma segura mediante una red distribuida formada por diferentes entidades, o Bitcoin [2] que permite hacer intercambios monetarios a través de esa red. Todo ello se consigue realizar de forma segura mediante programas informáticos, o contratos inteligentes, que se encargan de hacer cumplir las normas que haya en la red.

Por ello, cuanto más se pueda ayudar a esas tecnologías con herramientas que perfeccionan su uso, que lo hacen más fácil, y más seguro, mejor.

Una herramienta típica en el mundo de la programación actualmente son los verificadores de software, que realizan análisis estáticos de los programas, es decir, estudian el comportamiento de un programa sin llegar a ejecutarlo, evaluando todos los posibles estados por los que puede pasar la ejecución en función de los distintos valores que puedan tomar las variables que tenga. Por tanto, es una herramienta muy útil también para los contratos inteligentes, porque cuantos más seguros se puedan hacer estos programas mucho mejor funcionará la red, y menos vulnerabilidades tendrá.

Abstract

Nowadays the transmission of information through Internet is a reality, as well as the use of technologies such as Blockchain that allow these transactions to be carried out safely through a distributed network formed by different entities, or Bitcoin that allows to make monetary exchanges through that network. All this can be done safely using computer programs, or smart contracts, which are responsible for enforcing the rules that are on the network.

Therefore, the more these technologies can be helped with tools that improve their use, make it easier, and safer, the better.

A typical tool in the programming world today are software verifiers, which perform static analysis of programs, that is, study the behavior of the program without executing them, analyzing all the possible states through which its execution can pass in function of the different values that the variables it has can take. Therefore, it is a very useful tool also for smart contracts, because the more secure these programs can be made, the better the network will work, and the fewer vulnerabilities it will have.

Tabla de contenidos

1 Introducción **1**

2 Tecnologías **4**
2.1 Blockchain 4
2.2 Verificadores de código 7
2.3 Ethir 8
2.4 CPAChecker 9
2.5 SAFEVM 10

3 Desarrollo **15**
3.1 Listado de operaciones 15
3.2 Transformación de EVM a C 26
3.3 Ejemplos de uso 31

5 Resultados y conclusiones **34**

6 Bibliografía **35**

Tabla de imágenes

Imagen 1: Blockchain..... 4

Imagen 2: Logo Ethereum..... 5

Imagen 3: Esquema Ethir..... 8

Imagen 4: Logo CPAChecker..... 9

Imagen 5: Pasos en la transformación.....10

Imagen 6: Ejemplo programa Solidity 1.....11

Imagen 7: Ejemplo grafo de control de flujo.....12

Imagen 8: Ejemplo de RBR obtenida.....13

Imagen 9: Objetivos de la implementación.....14

Imagen 10: Tabla de operaciones 1.....15

Imagen 11: Implementación variable 256 bits 16

Imagen 12: Implementación asignaciones..... 16

Imagen 13: Ejemplo asignaciones en RBR.....17

Imagen 14: Ejemplo asignaciones en C.....17

Imagen 15: Tabla de operaciones 2.....17

Imagen 16: Implementación ADD.....18

Imagen 17: Implementación BYTE.....19

Imagen 18: Tabla de operaciones 3.....20

Imagen 19: Implementación SIGNEXTEND.....20

Imagen 20: Ejemplo contrato Solidity 2.....21

Imagen 21: Tabla de operaciones 4.....22

Imagen 22: Implementación OR.....22

Imagen 23: Tabla de operaciones 5.....23

Imagen 24: Implementación GT.....24

Imagen 25: Tabla de operaciones 6.....24

Imagen 26: Implementación SLT.....25

Imagen 27: Ejemplo programa Solidity 3.....25

Imagen 28: Ejemplo bloque RBR.....26

Imagen 29: Ejemplo bloque en C.....26

Imagen 30: Ejemplo programa Solidity 4.....28

Imagen 31: Bloque del CFG con invalid.....29

Imagen 32: Bloque de RBR con invalid.....29

Imagen 33: Ejemplo de C con invalid.....29

Imagen 34: Ejemplo programa Solidity 5.....31

Imagen 35: Grafo de control de flujo obtenido.....32

Imagen 36: Bloque RBR con invalid.....32

Imagen 37: Ejemplo programa Solidity.....33

1 Introducción

En este capítulo se va a describir el problema que dio origen a este trabajo, así como los objetivos planteados para realizarlo.

La transmisión de información a través del mundo gracias a Internet es, hoy en día, una realidad. Y entre todos esos tipos de movimientos existe la tecnología Bitcoin. En esta tecnología gracias al uso de contratos a través de la red, y del compromiso de todas las personas que lo utilizan, se ha conseguido crear un sistema descentralizado mediante Internet, a través del cual se puedan realizar transacciones de valor monetario.

Para conseguir este sistema descentralizado con dichos contratos, o normas para que todo funcione correctamente, se utiliza la tecnología Blockchain, gracias a la cual se consigue crear una red de nodos, que son las entidades participantes en ella, que se comprometen a cumplir los contratos. Gracias a ello se tiene una red por la que se transmiten bloques de información de forma segura y fiable. Esta red es como una máquina de estados basada en transacciones criptográficamente segura.

Existen diversas razones sobre las que mantener los objetivos de esta tecnología, pero posiblemente resalta el objetivo de facilitar la transmisión de transacciones entre las entidades, o personas, que de otra forma no podrían realizarse, por la zona geográfica donde se encuentra cada entidad, por problemas para conectar ambas entidades, por miedo a que haya corrupción… Y por eso, es tan beneficioso un sistema de estados, en el que todas las operaciones posibles siempre estarán bajo control y siguiendo unas reglas. Además de asegurar que los contratos o acuerdos que se crean en la red se van a cumplir, y de manera autónoma.

También es importante destacar otro objetivo clave de esta tecnología, y es la transparencia, y esto se debe a que cualquier nodo de la red podrá tener toda la información acerca de una transacción, de cómo se llegó a un estado, o de cualquier información relativa a alguna operación mediante un registro de operaciones, así como todas las reglas que hay en la red.

Como es natural en un mundo tecnológico relativamente nuevo, existen diversas vías de investigación dentro de él y diferentes herramientas que están en pleno desarrollo, para mejorar y facilitar su uso.

Una de las ramas con gran desarrollo en los últimos años es la referente a los contratos, o acuerdos que deben cumplir todas las partes, y es en esa parte en la que está orientada el trabajo realizado. En concreto, se ha querido conseguir que un contrato pueda ser fácilmente verificado, con herramientas que existen en la actualidad muy potentes.

La verificación en la programación es algo muy útil a muchos niveles, y aunque en posteriores capítulos se expondrá más información sobre ello, así como ejemplos de uso, es importante destacar desde el comienzo su importancia, ya que ha motivado el trabajado que se ha realizado durante este tiempo.

Las herramientas de verificación son un elemento muy útil para detectar fallos en el código, estudiar propiedades concretas… y, en definitiva, para poder asegurar que se tiene un programa suficientemente sólido y bueno, todo ello de forma estática, es decir, sin necesidad de ejecutar el programa con unos ciertos valores. Y este es una de las características más fuertes que tienen estas herramientas, pues permite estudiar todo el programa, con todos los posibles valores de entrada, con todas sus posibles ramas… Lo que es algo de gran ayuda para la persona que lo desarrolla, ya que le permite dar una visión global y total sobre su código.

Es por ello, que durante los últimos años han ido siendo más demandadas estas herramientas por los programadores, y se han ido creando con el paso del tiempo algunas muy potentes y eficaces para diferentes lenguajes.

Y, dentro del contexto del Blockchain pueden ser una herramienta muy útil también, para asegurar que los contratos, que no son más que programas informáticos, funcionan correctamente y cumplen unas ciertas propiedades deseadas. Pero al ser una tecnología que está en vías de desarrollo todavía no cuenta con verificadores tan potentes para los lenguajes típicos de los contratos, como el lenguaje Solidity.

Para proveer a Solidity [3] de herramientas de verificación eficaces ya existentes se planteó la conversión de un programa Solidity en un programa C, pudiendo así utilizar los verificadores de C, que se ha comprobado que obtienen resultados muy satisfactorios. Pero existe un problema en esta conversión, y es que Solidity trabaja con palabras de tamaño de 256 bits, mientras que C trabaja con palabras de 32 o 64 bits, lo cual dificulta la transformación.

En posteriores secciones del documento se va a tratar más en profundidad todos los pasos realizados para conseguir la transformación descrita anteriormente, y el código asociado, así como ejemplos de uso, pero a continuación se va a plantear cual fue la solución propuesta.

La idea inicial para superar el problema era crear una estructura a bajo nivel. Creando una palabra en C de tamaño de 256 bits, y todo el juego de operaciones típico en cualquier ensamblador asociado a ella. Y para logar ese tamaño de palabra en C se creó una estructura que concatenara 4 palabras de 64 bits cada una, obteniendo lo que se necesitaba, un tamaño de palabra de 256 bits, y con ella todas las instrucciones asociadas. Pero al realizar pruebas con el verificador de C se observó que no funcionaba correctamente ya que se generaba overflows, con todos los problemas de seguridad que esto conlleva para un programa, por lo que se cambió el objetivo, y se realizó la concatenación de palabras más pequeñas, 32 bits, para conseguir los 256 bits, y todo su juego de instrucciones asociado.

Por último, destacar que dado lo interesante que era esta herramienta se presentó un poster sobre la misma a Association for Computing Machinery, ACM. Fue aceptado y está a falta de presentación.

2 Tecnologías

En este capítulo se van a explicar brevemente las tecnologías y herramientas que han formado parte del trabajo con el objetivo de poder entender después el trabajo desarrollado.

2.1 Blockchain

Blockchain, o cadena de bloques, es una tecnología que permite crear una red formada por entidades, distribuida en la red, en la que va a haber intercambios de bloques de información de forma segura y fiable.

Cada entidad o nodo tiene una posición exacta en la red, estando vinculado mediante un algoritmo matemático al siguiente nodo.

Existe un único registro para toda la cadena que va a almacenar todos los movimientos y cambios por los que va pasando, registro transparente para todas las entidades. Cada operación o transacción será un bloque de la cadena, y ese bloque lo tendrán todas las entidades. En el momento en el que la transacción es aprobada por todos los nodos de la cadena y es realizada, se añade al registro común de transacciones

Además, es importante destacar que la información va a estar disponible en todo momento, teniendo cada entidad una copia exacta de la cadena de transacciones que se ha ido realizando. Dicha información siempre será verificada mediante firmas digitales y certificados, para poder asegurar la autenticidad de esta.

Con todo ello se consigue una cadena en la que realizar transacciones se convierte en una operación segura, resistente a la manipulación de la información, y donde cualquier movimiento queda registrado, y todo de forma descentralizada.

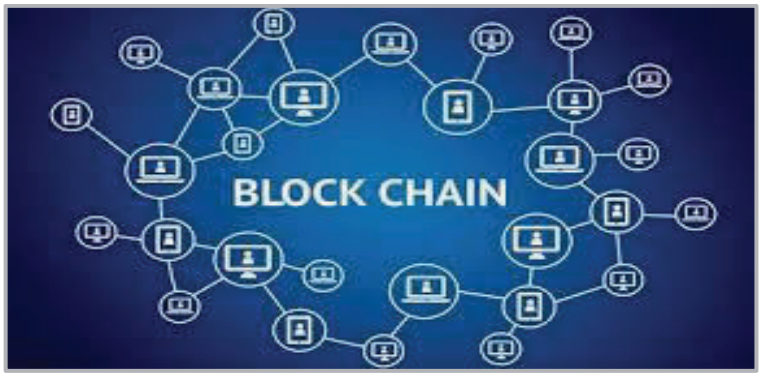


Imagen 1. Blockchain

Para ejemplificar como funciona se va a explicar un supuesto.

Varias personas quieren hacer intercambios monetarios sin tener que recurrir a intermediarios, crean la cadena de bloques y cada persona o conjunto de ellas va a forman un nodo en esta cadena. En ella ninguno de los nodos sabe la identidad del resto de nodos, únicamente van a saber las operaciones que quieran realizar para consensuar si es una operación permitida o no.

En dicha red uno de los nodos pretende enviar una cantidad de bitcoins a otro nodo, de forma que esta operación se comunica al resto de nodos creándose un bloque de información. Los demás nodos comprobarán si el nodo que quiere realizar la transacción tiene suficiente capacidad monetaria para hacerlo. Si comprueban que todo es correcto aprueban la operación, de forma que cada nodo se anota la transacción realizada y además se añade la transacción al registro único de transacciones.

Para que puedan cumplirse todas las propiedades que caracterizan una cadena de bloques, o Blockchain, existen los Smart Contracts, o contratos inteligentes.

Estos contratos son programas informáticos que no estan sujetos a ninguna de las entidades en concreto, sino a todas, representando todos los acuerdos a los que llegan las diferentes partes de la cadena, y se encargar de decidir qué operaciones se pueden hacer en un determinado momento, y que acciones desencadenará que alguna de las partes incumpla alguna condición.

Por tanto, los Smart Contracts es una parte fundamental en la cadena de bloques, siendo los encargados de asegurar que todas las normas se cumplen.

Existen diferentes lenguajes en los que se pueden desarrollar estos contratos, destacando Solidity, que está pensado para ejecutarse en la plataforma Ethereum [4].

Imagen 2. Logo Ethereum

Ethereum es una plataforma global, de código abierto, creada para aplicaciones descentralizadas, pudiéndose ejecutar desde cualquier parte del mundo. En ella los participantes pueden crear Smart Contracts y la plataforma comprueba si se cumplen o no los requisitos para ejecutar una determinada transacción.

Es una plataforma que permite eliminar los obstáculos burocráticos, permitiendo a las entidades interactuar entre ellos directamente. A su vez toda la información referente a la entidad es privada, ninguna entidad podrá acceder a información de otra. Ademas, las entidades pueden utilizar su propia moneda o si fueran empresas representar las acciones de la misma.

En esta plataforma ejecutar una operación o transacción conlleva un coste. Este coste se denomina *gas,* y su finalidad no es otra que asignar un coste a cada tarea, teniendo mayor coste crear una aplicación descentralizada, por ejemplo, que ejecutar un Smart Contract.

No hay que confundir el concepto de *gas* con el de moneda digital, no es algo que tenga valor monetario y se posea o se intercambie. Es un concepto que permite cuantificar el gasto computacional que supone realizar las operaciones en la plataforma, y cada transacción lleva asociado su *gas limit,* que representa la máxima cantidad de gas que puede llegar a "costar" realizar esa transacción.

Ethereum trabaja a bajo nivel con una máquina virtual, denominada Ethereum Virtual Machine (EVM) para ejecutar los programas. Esta máquina virtual trabaja con un código intermedio que es una mezcla de scripts de bitcoin y ensamblador. Utiliza unos tamaños de palabra de 256 bits, un juego de instrucciones, y está basado en pila.

2.2 Verificadores de código

Los verificadores de código son herramientas que analizan de forma estática un programa informático, cuyo objetivo es evaluar todas las posibles ramas que podría ejecutar dicho programa, siendo capaz de determinar si se cumple una propiedad determinada, estudiar el comportamiento del programa ante determinadas entradas, evaluar puntos de fallo... Es, es definitiva, una herramienta muy útil para un programador, y es por eso por lo que en los últimos años se han ido desarrollando algunas muy potentes para diferentes lenguajes.

Para utilizar un verificador simplemente hay que indicarle el programa a verificar y la propiedad a estudiar. Por ejemplo, dado un código evaluar si el resultado final es siempre positivo. Y la herramienta, como salida indicará si siempre se cumple la propiedad, si no se cumple nunca, si se cumple a veces, o si no es capaz de determinarlo.

Gracias a ello se puede conseguir reparar puntos de fallo en un momento muy temprano del ciclo de vida del software, permitiendo ahorrar en costes de reparaciones futuras.

Es, por tanto, una herramienta muy útil para analizar Smart Contracts sobre todo teniendo en cuenta que estos programas son públicos, por lo que están sometidos a errores de programación. Pero realmente no existe hoy en día una herramienta suficientemente potente para Solidity como pueden existir para otros lenguajes como C.

También es importante destacar que en Ethereum existe el concepto de *gas*. Gas es lo que los participantes van comprando para que sus programas puedan ejecutarse en la plataforma, en cuanto no hay gas se aborta la ejecución. Pero, además, existe una instrucción en el juego de instrucciones del ensamblador de EVM que es *invalid,* y cuando se da esta instrucción deja de ejecutarse el programa.

Esta instrucción puede surgir de cuatro formas. Por intentar acceder a una posición no perteneciente a un array, cuando se intenta hacer una división por cero, cuando un tipo enumerado se sale de su rango de representación, y por último cuando se escribe en el programa una condicional con un *assert*. Este tipo de condicionales funcionan como un if, pero saltando un *invalid* en caso de no cumplirse ésta.

Por todo ello, proveer a un programador de una herramienta capaz de analizar si su código alcanza un *invalid* sin tener que ejecutarlo, es una gran ayuda.

2.3 Ethir [5]

En este trabajo se ha utilizado el framework de Ethir como punto de partida.

Es una herramienta que permite analizar programas EVM para estudiar posibles vulnerabilidades en el código. Para ello realiza una serie de transformaciones llegando a obtener una Representación Basada en Reglas, o RBR, la cual muestra los cambios que se van produciendo en la pila con la que trabaja EVM, las instrucciones, los saltos… pero en forma de código y con una estructura de bloques.

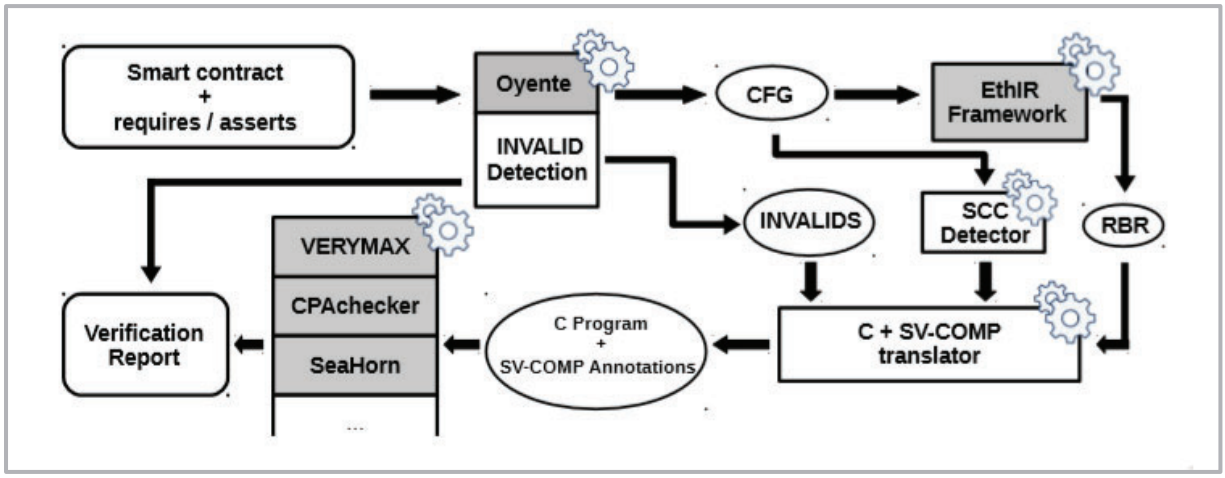


Imagen 3. Esquema de Ethir

Este esquema muestra los pasos que realiza esta herramienta.

Toma como punto de entrada un Smart Contract en el que puede haber *requires* o *asserts*, posteriormente utiliza el framework Oyente [6]. Este framework hace un análisis simbólico de cómo sería la ejecución del programa en funcion de diferentes entradas, todo de forma estática, es decir, es una herramienta capaz de razonar sobre las diferentes rutas o estados por los que puede pasar un programa simplemente analizándolo, sin tener que ejecutarlo. De esta forma es capaz de ir recreando los posibles valores que pueden tomar las diferentes variables que tenga el programa e ir estudiando sus posibles terminaciones.

Este análisis genera un Control Flow Graph, o CFG, que muestra todos los posibles estados a los que puede transitar un programa.

Una vez realizado este grafo Ethir se encarga de recrear la RBR mencionada anteriormente, cuyos bloques representan los estados del grafo, que contienen todas las instrucciones EVM que se ejecutan, y los saltos son las posibles transiciones entre ellos.

2.4 CPAChecker [7]

La finalidad que tiene este proyecto es poder verificar Smart Contracts transformados desde código Solidity a codigo C, para lo cual se toma como punto de partida el framework de Ethir para llegar a una representación de la pila que se genera en la máquina virtual de Ethereum al ejecutar un programa, basada en reglas y en formato de código.

Una vez se tiene esta representación se ha trabajado en recrear dicho código, pero en un programa en C, cuyo objetivo es poder usar los verificadores que existen para este lenguaje, ya que tienen unos resultados muy satisfactorios y se lleva trabajando con ellos bastante tiempo.

En concreto, con el verificador de C con el que se ha querido trabajar ha sido con CPAChecker.

Imagen 4. Logo CPAChecker

Es una herramienta de verificación que lleva desde 2007 en desarrollo, y que genera unos resultados de verificación muy satisfactorios, ademas ha conseguido ganar varios premios en competiciones de verificación de software.

Como su nombre indica es una herramienta que se basa en conceptos CPA y en verificación de programas informáticos.

Además, es una herramienta que se instala con mucha facilidad y su uso es también muy sencillo, y que permite ver en una página web un grafo con los pasos de la verificación del programa

Cuando se verifica una propiedad de un programa CPAChecker tiene dos posibles salidas. True, en cuyo caso no se incumple nunca la propiedad, o false en caso contrario.

2.5 SAFEVM [8]

SAFEVM es una herramienta que transforma un código Solidity en un código C.

En esta sección del documento se quiere mostrar el funcionamiento de dicha herramienta para posteriormente poder explicar en profundidad cuál ha sido el trabajado realizado, el cual parte de la transformación que realiza esta herramienta.

La secuencia de pasos necesaria para conseguir dicha transformación sería comenzar compilando el programa Solidity, por ejemplo, con el compilador solc, consiguiendo asi las instrucciones EVM (Ethereum Virtual Machine) del programa. Con dichas instrucciones la herramienta hace una transformación pasando de la pila inicial obtenida por la EVM a una representación de dicha pila basada en reglas, la RBR, en la cual estarán todas las variables, tanto de pila, de estado, y de memoria de forma explícita, así como todas las operaciones que realiza el ensamblador de EVM. Posteriormente, dicha RBR pasa a generar el programa en C.

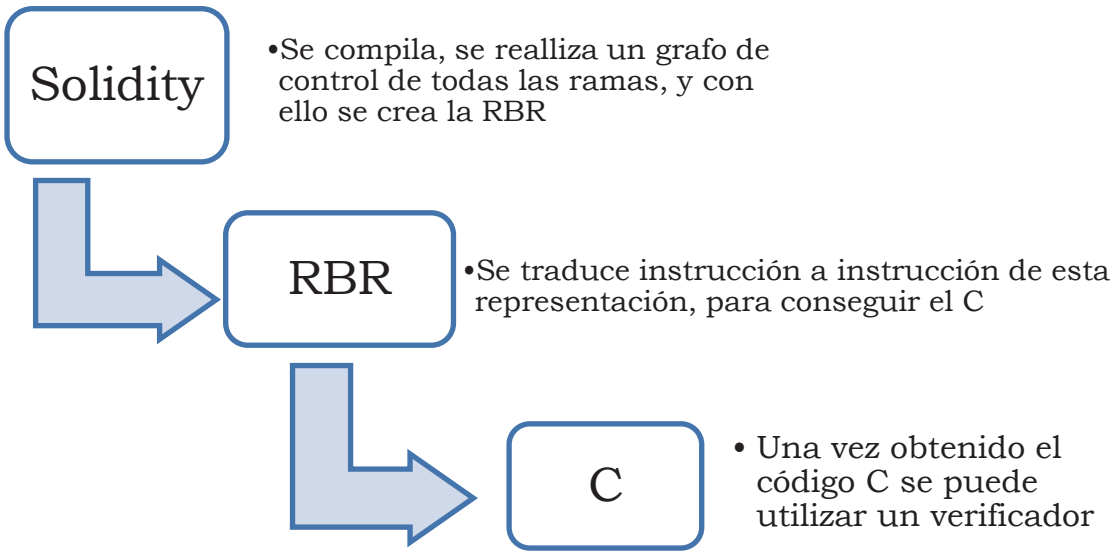


Imagen 5. Pasos en la transformación

Una vez se tiene el programa en C se pueden utilizar algunos de sus verificadores más potentes, como CPAChecker o SeaHorn [9].

A continuación, se va a detallar gráficamente como se realiza este proceso.

Primero se realiza un programa Solidity, como muestra la imagen 1. Simplemente se tiene un contrato en el que hay una función que recibe un array y un número n, posteriormente se comprueba que ese valor n nunca va a ser mayor que el tamaño del array. Luego se recorre el array en un bucle de n iteraciones dejando un 0 en cada posición.

La función que tiene la instrucción requiere es comprobar la propiedad de que no se intente acceder nunca a una posición del array inexistente, y es la propiedad que va a estudiar el verificador.

```
contract Example {

    function foo (uint [] a, uint n) public  {
        require(n < a.length);
        for (uint i = 0; i < n; i ++) {
            a[i] = 0;
        }
    }
}
```

Imagen 6. Ejemplo programa Solidity 1

El siguiente paso sería la compilación, obteniendo las instrucciones EVM, y posteriormente el uso del framework Oyente para construir el grafo de control de flujo. Este grafo lo que muestra son todas las posibles ramas o estados que puede tomar el programa Solidity de entrada.

En él se puede observar que cada nodo es un bloque, $Block_x$, y en función del valor que tomen las variables el programa ejecutará por una rama o por otra, es decir, se transforman los saltos propios de instrucciones de una pila, en saltos entre bloques. Los valores de esas variables no son conocidos, sino que el framework hace una representación simbólica del programa, es decir, lo analiza cada salto entre bloques para todos los posibles valores que tomen las variables del programa.

Cada bloque está formado por un conjunto de instrucciones del ensamblador EVM, las cuales luego pasaran a formar la representación basada en reglas.

Se puede observar también que existe un Block con un *invalid*, que como se comentó anteriormente esta instrucción es del ensamblador de EVM y se produce por cuatro razones. En este ejemplo se produce en caso de ejecutar algo que no cumple la propiedad especificada en el *requiere*, en este caso, querer acceder a una posición que no es del array.

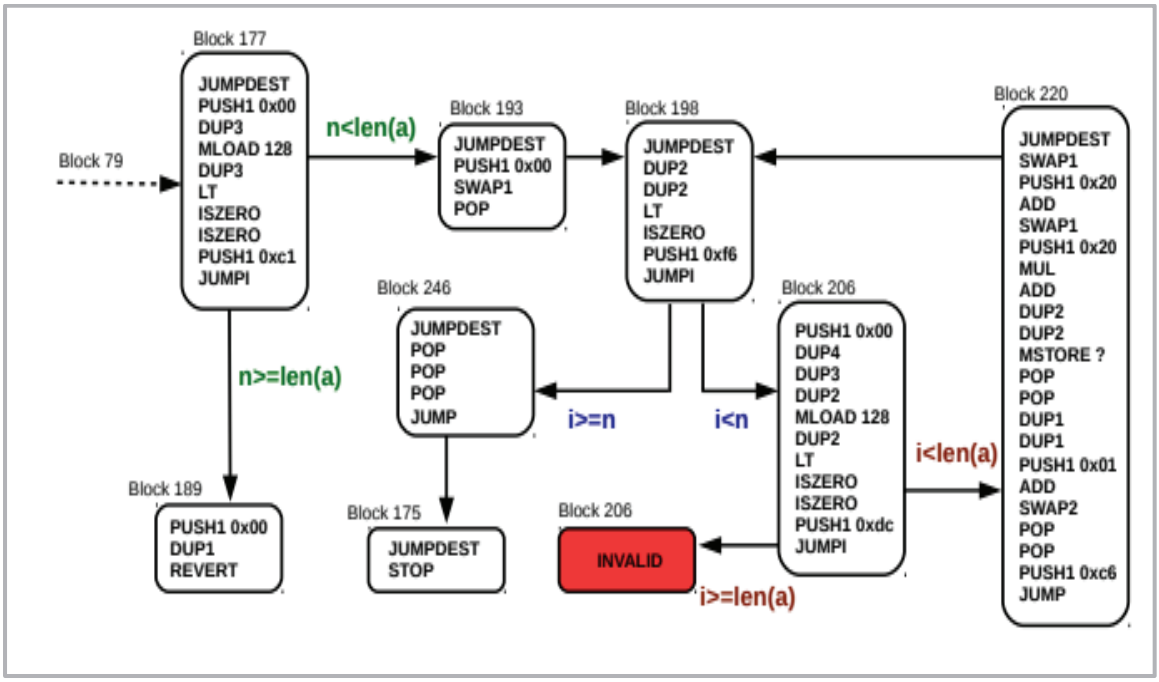


Imagen 7. Ejemplo de grafo de control de flujo

El siguiente paso en la herramienta es crear la Representación Basada en Reglas, o RBR. Esta representación, que se observa en la siguiente imagen, contendrá la misma información que el grafo anterior, pero en formato de código, y con todas las variables de estado, pila y memoria de forma explícita.

Gracias a esta representación se pueden ir traduciendo instrucción a instrucción para pasarlo a código C.

Se puede ver que cada bloque de código tiene una serie de instrucciones, tanto de asignación, lógicas, llamadas a otros bloques… Y hay una en concreto que es *nop (INVALID)*, la cual será llamada desde aquellos bloques en los que se pueda vulnerar la propiedad especificada en el requiere del código Solidity. De esta forma, el verificador al analizar las diferentes ramas por las que ejecuta un programa en concreto podrá evaluar si llega al bloque del INVALID, en cuyo caso la propiedad requerida no se ha cumplido, y salida de éste será negativa, mientras que, si al analizarlo no llega por ninguna rama a ese bloque, su salida será positiva.

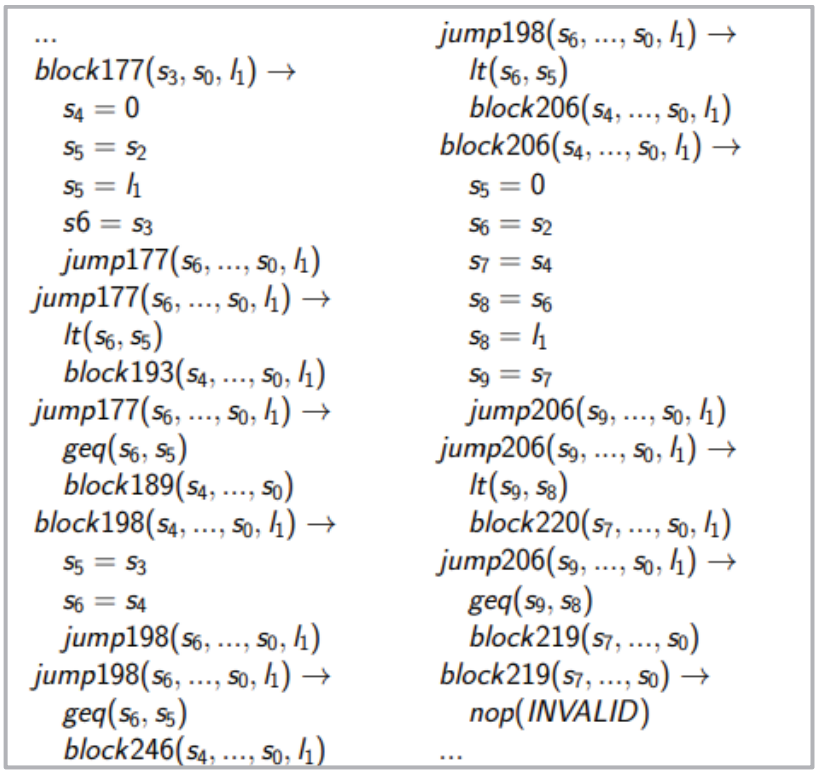


Imagen 8. Ejemplo RBR obtenida

Una vez obtenida la RBR el siguiente paso es conseguir el programa en C.

Pero como se ha comentado anteriormente, las instrucciones que se generan EVM, al compilar el programa Solidity, son con palabras de 256 bits, mientras que C trabaja con palabras de 32 o 64 bits, por lo que se produce un problema de tamaños en el último paso de la transformación descrita.

Es por ello por lo que se pensó en recrear a bajo nivel el comportamiento del ensamblador del lenguaje C, pero en vez de con palabras de 32 o 64 bits, con palabras de 256 bits.

Para crear dicha longitud de palabra se creó una estructura en C formada por 8 palabras de 32 bits.

Una vez obtenida el nuevo tamaño de palabra había que implementar todas las operaciones del juego de instrucciones de C, tanto aritméticas como lógicas, todo ello para las dos posibles representaciones que tiene una variable en C, con signo y sin él.

Después de conseguir las operaciones y estudiar que funcionan correctamente, otro objetivo es comprobar que el verificador funciona bien con ellas, ya que no son las operaciones típicas del lenguaje C podrían desarrollar más problemas. Para ello se hacen programas de prueba instrucción a instrucción y se observa el comportamiento del verificador, y luego se hacen otros en los que haya más de una instrucción presente para comprobar el funcionamiento del verificador.

Por tanto, la lista de objetivos prevista para el desarrollo completo de este trabajo sigue los pasos que muestra la figura a continuación.

Finalmente, cuando las operaciones funcionaban correctamente se añadieron a la herramienta SAFEVM para hacer transformaciones de Solidity a C completas con este ensamblador recreado.

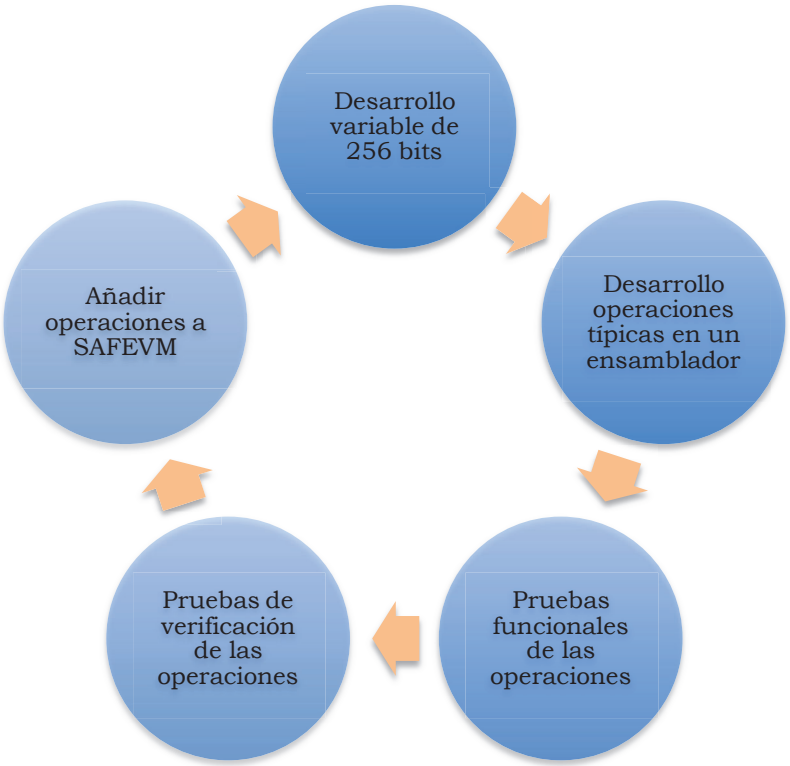


Imagen 9. Objetivos en la implementación

3. Desarrollo

Este capítulo se centra en el desarrollo de estos meses de trabajo, explicando paso a paso como se ha ido creando lo anteriormente expuesto.

Además, al final del capítulo se encuentran algunos ejemplos gráficos de la ayuda que puede llegar a suponer esta transformación.

3.1 Listado operaciones realizadas en C

Para facilitar las referencias a las funciones implementadas que se van a realizar en las secciones posteriores, se va a mostrar a continuación un listado con las operaciones realizadas, su nombre, tipo, parametros de entrada y valores de retorno, clasificadas por el tipo de operación que realizan.

Tipo	Operación	Parametros de entrada	Retorno
Creación y asignaciones	cons	8 variables de tipo unsigned int	1 variable de tipo ethint256
	COPY	2 variables de tipo ethint256	1 variable de tipo ethint256
	ASSIGN	8 variables de tipo int	1 variable de tipo ethint256

Imagen 10. Tabla operaciones 1

Primero se creó el nuevo tipo de tamaño de variable y su constructor asociado, como se ve en la siguiente imagen.

```
typedef struct  {
  unsigned int w0;
  unsigned int w1;
  unsigned int w2;
  unsigned int w3;
  unsigned int w4;
  unsigned int w5;
  unsigned int w6;
  unsigned int w7;
} ethint256 ;


 ethint256 cons(unsigned int w7, unsigned int w6, unsigned int w5,
    unsigned int w4 ,unsigned int w3, unsigned int w2, unsigned int w1, unsigned int w0) {
   ethint256 res;
  res.w7 = w7;
  res.w6 = w6;
  res.w5 = w5;
  res.w4 = w4;
  res.w3 = w3;
  res.w2 = w2;
  res.w1 = w1;
  res.w0 = w0;
  return res;
}
```

Imagen 11. Implementación variable 256 bits

Luego se realizaron las asignaciones, tanto de valores numéricos como de otras variables, como muestra la siguiente imagen.

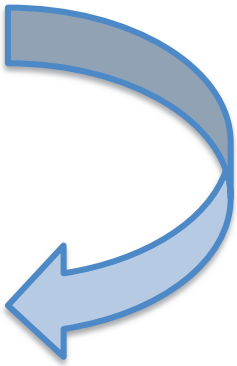
```
ethint256 COPY(ethint256 x){
  ethint256 res;
  res.w0 = x.w0;
  res.w1 = x.w1;
  res.w2 = x.w2;
  res.w3 = x.w3;
  res.w4 = x.w4;
  res.w5 = x.w5;
  res.w6 = x.w6;
  res.w7 = x.w7;
  return res;
}

ethint256 ASSIGN(int x0,int x1,int x2,int x3,int x4,int x5,int x6, int x7){
  ethint256 res;
  res.w0 = x7;
  res.w1 = x6;
  res.w2 = x5;
  res.w3 = x4;
  res.w4 = x3;
  res.w5 = x2;
  res.w6 = x1;
  res.w7 = x0;
  return res;
}
```

Imagen 12. Implementación asignaciones

Como pueden ser números de extensión muy grande debido al tamaño de 256 bits, desde el programa de Python que realiza las llamadas a funciones C se crean palabras de 32 bits, 8 como máximo, y se realiza la llamada a la función *assign* con las 8 palabras en valor hexadecimal, en las siguientes imágenes se muestra un breve ejemplo de transformación.

```
s(1) = 4294967295
nop(PUSH4)
s(0) = and(s(1), s(0))
nop(AND)
s(1) = s(0)
nop(DUP1)
s(2) = 3794413849
```

Imagen 13. Ejemplo asignaciones en la RBR



```
s1 = ASSIGN(0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xffffffff);
s0 = AND(s1,s0);
s1 = COPY(s0);
s2 = ASSIGN(0x0,0x0,0x0,0x0,0x0,0x0,0x0,0xe22a2919);
```

Imagen 14. Ejemplo de asignaciones en C

En la primera imagen se observa el código de la RBR, en la cual hay una variable con valor decimal 4294967295, y otra con valor decimal 3794413849. En el programa de Python lo que se hace es transformarlas a valor hexadecimal, en el primer caso 0xffffffff y en el segundo caso 0xe22a2919, y posteriormente formar con ese valor hexadecimal las 8 palabras que recibe la función *assign* de C para crear la variable con tamaño de palabra de 256 bits.

Una vez se comprobó que funcionaban adecuadamente ambas operaciones se pasó a la implementación de dos operaciones básicas, la suma y la resta.

Tipo	Operación	Parametros de entrada	Retorno
Aritméticas sin signo	ADD	2 variables de tipo ethint256	1 variable de tipo ethint256
	SUB	2 variables de tipo ethint256	1 variable de tipo ethint256
	BYTE	2 variables de tipo ethint256	1 variable de tipo ethint256

Imagen 15. Tabla operaciones 2

Un ejemplo de implementación se muestra continuación, en la cual se va sumando palabra a palabra de cada estructura y se va llevando el acarreo hacia la izquierda.

```
ethint256 ADD(ethint256 x, ethint256 y) {
  ethint256 resAdd;
  int carry;
  resAdd.w0 = x.w0 + y.w0;
  carry = (resAdd.w0 < x.w0);
  resAdd.w1 = x.w1 + y.w1 + carry;
  carry = (resAdd.w1 < x.w1);
  resAdd.w2 = x.w2 + y.w2 + carry;
  carry = (resAdd.w2 < x.w2);
  resAdd.w3 = x.w3 + y.w3 + carry;
  carry = (resAdd.w3 < x.w3);
  resAdd.w4 = x.w4 + y.w4 + carry;
  carry = (resAdd.w4 < x.w4);
  resAdd.w5 = x.w5 + y.w5 + carry;
  carry = (resAdd.w5 < x.w5);
  resAdd.w6 = x.w6 + y.w6 + carry;
  carry = (resAdd.w6 < x.w6);
  resAdd.w7 = x.w7 + y.w7 + carry;
  carry = (resAdd.w7 < x.w7);
  return resAdd;
}
```

Imagen 16. Implementación ADD

La última operación que muestra la tabla es la de BYTE, y es una operación muy importante en el ensamblador de EVM porque es muy habitual al realizar contratos Solidity declarar variables de un tamaño predeterminado, por ejemplo, se crea un int pero en vez de con 4 bytes con 8 bits, o 16.

```
ethint256 BYTE(ethint256 x, ethint256 y){
    ethint256 res;
    unsigned int w;
    res.w1 = res.w2 = res.w3 = res.w4 = res.w5 = res.w6 = res.w7 = 0;

    if (y.w0 < 8) w = x.w7;
    else if (y.w0 < 16 ) w = x.w6;
    else if (y.w0 < 24 ) w = x.w5;
    else if (y.w0 < 32 ) w = x.w4;
    else if (y.w0 < 40 ) w = x.w3;
    else if (y.w0 < 48 ) w = x.w2;
    else if (y.w0 < 56 ) w = x.w1;
    else if (y.w0 < 64 ) w = x.w0;

    int offset = y.w0 % 8;

    if (offset == 0) res.w0 = (w & 0xF0000000) >> 28;
    if (offset == 1) res.w0 = (w & 0x0F000000) >> 24;
    if (offset == 2) res.w0 = (w & 0x00F00000) >> 20;
    if (offset == 3) res.w0 = (w & 0x000F0000) >> 16;
    if (offset == 4) res.w0 = (w & 0x0000F000) >> 12;
    if (offset == 5) res.w0 = (w & 0x00000F00) >> 8;
    if (offset == 6) res.w0 = (w & 0x000000F0) >> 4;
    if (offset == 7) res.w0 = (w & 0x0000000F) >> 0;
    return res;
}
```

Imagen 17. Implementación BYTE

Retomando la implementación de la operación ADD o SUB, como puede observarse, no tiene en cuenta si son tipos con signo o no, simplemente suma palabra a palabra y si hay acarreo lo suma tambien, es decir, se puede crear un contrato en Solidity con tipos int (con signo) que van a realizar esa suma igual que si se tratase de tipos uint (sin signo).

De igual forma ocurre con la operación resta, va restando palabra a palabra y si hay acarreo lo decrementa también.

Esto se debe a la forma con la que se ha tratado el signo.

Para hacerlo de la forma más real posible la herramienta debería tener en cuenta una versión de operaciones con signo, y el mayor problema era extender el signo, porque el caso más facil sería que el bit número 255, comenzando a contar por la derecha, fuera un uno, pero no siempre se va a dar de esa forma tan sencilla porque no siempre se van a tener valores que ocupen los 256 bits, es posible tener un valor que solo ocupe 8 bits y que sea negativo.

Por ello se realizó una operación que se encargase de extender el signo, se muestra en la imagen siguiente una parte de la implementación.

Para entender cómo funciona también hay que tener en cuenta que las palabras que forman el struct son sin signo, por lo que siempre será la herramienta la que deba tener en cuenta la representación con signo.

Dicha representación será como una representación en complemento a 2, CA2.

Tipo	Operación	Parametros de entrada	Retorno
Aritmética con signo	SIGNEXTEND	2 variables de tipo ethint256	1 variable de tipo ethint256

Imagen 18. Tabla operaciones 3

A continuación, se muestra cómo se ha implementado esta operación en C, y un ejemplo partiendo de un código Solidity para ver cómo funciona.

```
ethint256 SIGNEXTEND(ethint256 v0,  ethint256 y){

  int v1 = y.w0;
  ethint256 byte;

  if(v1 == 0){
    byte = cons(0,0,0,0,0,0,0,v0.w0 & 0x000000ff);
    int x = v0.w0 & 0x000000ff;
    if(x <= 127){ return v0;}
    else{ return cons(0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffff00 | x);
  }
  if(v1 == 1){
   byte = cons(0,0,0,0,0,0,0,v0.w0 & 0x0000ffff);
    int x = v0.w0 & 0x0000ffff;
    if(x <= 127){ return v0;}
    else{ return cons(0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffff0000 | x);
  }
  if(v1 == 2){
   byte = cons(0,0,0,0,0,0,0,v0.w0 & 0x00ffffff);
    int x = v0.w0 & 0x00ffffff;
    if(x <= 127){ return v0;}
    else{ return cons(0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xff000000 | x);
  }
  if(v1 == 3){
   byte = cons(0,0,0,0,0,0,0,v0.w0 & 0xffffffff);
    int x = v0.w0 & 0xffffffff;
    if(x <= 127){ return v0;}
    else{ return cons(0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0xffffffff,0x00000000 | x);
  }
```

Imagen 19. Implementación extensión de signo

Simplemente se muestra un fragmento para ver el funcionamiento. La función recibe el numero al que extender el signo como primer parametro y la posición desde la que extenderlo, parametros que se le pasan desde el programa en Python que se encarga de la traducción de la RBR.

La función lo que hace es quedarse únicamente con las posiciones que interesan para luego comprobar si en ellas hay un numero negativo, en cuyo caso devuelve el mismo número con todo unos por delante para hacerlo negativo, mientras que si es positivo no realiza ninguna transformación.

Como se puede ver en la imagen el segundo parametro, el que indica la posición desde la que extender el signo, puede tomar valores como 0, 1… hasta 31. Esto se debe a que en programas Solidity es usual declarar variables de tipo int en vez de con 4 bytes con 8 bits, 16 bits, 24 bits… Y por esa razón la operación de extender el signo tiene tantas posibles ramas.

A continuación, se va a ejemplificar lo que se ha explicado anteriormente para poder comprender como funciona la transformación.

Se parte de un contrato Solidity en el que se crean dos variables de tipo int, es decir, con signo, y de tamaño de 8 bits. Teniendo 2^8 posibles palabras en total con 8 bits existen 2^{8-1} -1 posibles números positivos, y 2^{8-1} posibles números negativos, es decir, positivos hay desde el 0 hasta el 127 y negativos del -1 hasta el -128.

Teniendo en cuenta como son los posibles rangos con 8 bits, se realiza la suma que se muestra en el programa. Al sumarle un 1 al 127 el resultado es negativo, porque el 128 se encuentra en el rango de los negativos. Y es precisamente esta la propiedad que comprueba el assert que se encuentra al final del contrato.

De igual forma se realiza un contrato que realiza la misma operación, pero con variables de tipo uint, por lo que la solución debe ser un numero positivo., propiedad que se comprueba en el assert del segundo contrato.

```
pragma solidity ^0.4.11;

contract Sum {

    function suma1 () {
        int8 x = 127;
        int8 y = 1;
        int8 z = x+y;
        assert(z < 0);
    }
    function suma2 () {
        uint8 x = 127;
        uint8 y = 1;
        uint8 z = x+y;
        assert(z > 0);
    }
}
```

Imagen 20. Ejemplo contrato Solidity

Tras ejecutar la herramienta con este contrato se obtienen varios ficheros, entre ellos el fichero con la RBR y el fichero con el código C.

Se llama al verificador CPAChecker con el fichero del código C, y se obtiene una salida positiva, es decir, no se ha incumplido ninguna condición de los assert.

Por tanto, se ha conseguido que la implementación de la operación suma y de la operación de extensión de signo trabajen correctamente para ambos escenarios, con signo y sin él.

De igual forma se han hecho comprobaciones con la operación resta y con diferentes tamaños de variable, y el verificador ha obtenido resultados satisfactorios.

Por otro lado, se realizaron las operaciones lógicas.

Tipo	Operación	Parametros de entrada	Retorno
Lógicas	AND	2 variables de tipo ethint256	1 variable de tipo ethint256
	OR	2 variables de tipo ethint256	1 variable de tipo ethint256
	XOR	2 variables de tipo ethint256	1 variable de tipo ethint256
	NOT	2 variables de tipo ethint256	1 variable de tipo ethint256

Imagen 21. Tabla operaciones 4

Las operaciones AND, OR, XOR y NOT se han realizado igual que el ensamblador de EVM, a nivel de bit.

Para ver cómo es la implementación se adjunta la operación OR.

```
ethint256 OR( ethint256 x,  ethint256 y) {
 ethint256 resOr;
 resOr.w0 = x.w0 | y.w0;
 resOr.w1 = x.w1 | y.w1;
 resOr.w2 = x.w2 | y.w2;
 resOr.w3 = x.w3 | y.w3;
 resOr.w4 = x.w4 | y.w4;
 resOr.w5 = x.w5 | y.w5;
 resOr.w6 = x.w6 | y.w6;
 resOr.w7 = x.w7 | y.w7;
 return resOr;
}
```

Imagen 22. Implementación OR

Continuando con las operaciones lógicas se realizaron las de comparación.

Tipo	Operación	Parametros de entrada	Retorno
Comparaciones lógicas sin signo	LT (Less Than)	2 variables de tipo ethint256	1 variable de tipo ethint256
	LT	2 variables de tipo ethint256	1 variable de tipo int
	LEQ (Less Than or Equals)	2 variables de tipo ethint256	1 variable de tipo ethint256
	LEQ	2 variables de tipo ethint256	1 variable de tipo int
	GT (Greater Than)	2 variables de tipo ethint256	1 variable de tipo ethint256
	GT	2 variables de tipo ethint256	1 variable de tipo int
	GEQ (Greater Than or Equals)	2 variables de tipo ethint256	1 variable de tipo ethint256
	GEQ	2 variables de tipo ethint256	1 variable de tipo int
	EQ (Equals)	2 variables de tipo ethint256	1 variable de tipo ethint256
	EQ	2 variables de tipo ethint256	1 variable de tipo int

Imagen 23. Tabla operaciones 5

Como se puede ver en la tabla cada operación de comparación ha sido realizadas dos veces para devolver dos posibles resultados, esto se debe a que dependiendo de cómo sea le programa Solidity de entrada se genera una RBR diferente, y en función de esta RBR se necesita que devuelva la palabra de 256 bits o simplemente un int con un 1 si la comparación fue correcta o un 0 en caso contrario.

También es importante tener en cuenta que todas estas comparaciones no tienen en cuenta el signo, únicamente serán llamadas desde la transformación de la RBR cuando los tipos del contrato Solidity que se estén traduciendo sean sin signo.

Un ejemplo de implementación se muestra en la siguiente imagen.

```
int GT( ethint256 x,  ethint256 y) {

  if( x.w7 > y.w7 || x.w6 > y.w6 || x.w5 > y.w5 ||
  x.w4 > y.w4 || x.w3 > y.w3 || x.w2 > y.w2 || x.w1 > y.w1 ||
  x.w0 > y.w0 ){ return 1;}

  if( x.w7 < y.w7 || x.w6 < y.w6 || x.w5 < y.w5 ||
  x.w4 < y.w4 || x.w3 < y.w3 || x.w2 < y.w2 || x.w1 < y.w1 ||
  x.w0 < y.w0 ){ return 0;}

  return 0;
}
```

Imagen 24 Implementación GT

Ahora si se van a mostrar las implementaciones de las comparaciones con signo.

Tipo	Operación	Parametros de entrada	Retorno
Comparaciones lógicas con signo	SLT (Signed Less Than)	2 variables de tipo ethint256	1 variable de tipo ethint256
	SLT	2 variables de tipo ethint256	1 variable de tipo int
	SGT (Signed Greater Than)	2 variables de tipo ethint256	1 variable de tipo ethint256
	SGT	2 variables de tipo ethint256	1 variable de tipo int

Imagen 25. Tabla operaciones 6

Para este caso únicamente se va a mostrar un fragmento de la implementación de una de ellas porque es una implementación mucho más densa.

En esta función lo primero que se hace es crear dos posibles resultados, una variable de tipo ethint256 con un 0 como valor, y otra con un 1.

Si la operación comprueba que la variable x contiene un valor menor que la variable y la funcion devuelve la variable con valor 1, en caso contrario devuelve la variable con valor 0.

Esta función va recorriendo palabra a palabra los struct de ambas variables, comenzando por la izquierda, y va comprobando si ambas son positivas o negativas, en cuyo caso se puede llamar a la función que se encarga de las operaciones de comparación sin signo. Si no es ese el caso se comprueba si la palabra correspondiente de la primera variable es negativa y la segunda no, en cuyo caso la primera variable es menor que la segunda. Y si es la palabra de la

primera variable positiva y la palabra de la segunda variable negativa, es menor la segunda por lo que devuelve un 0.

```
ethint256 slt(ethint256 x, ethint256 y){
    ethint256 res0 = cons(0,0,0,0,0,0,0,0);
    ethint256 res1 = cons(0,0,0,0,0,0,0,1);
    ethint256 z1 = cons(0,0,0,0,0,0,0,0x10000000);
    ethint256 z2 = cons(0,0,0,0,0,0,0x10000000,0);
    ethint256 z3 = cons(0,0,0,0,0,0x10000000,0,0);
    ethint256 z4 = cons(0,0,0,0,0x10000000,0,0,0);
    ethint256 z5 = cons(0,0,0,0x10000000,0,0,0,0);
    ethint256 z6 = cons(0,0,0x10000000,0,0,0,0,0);
    ethint256 z7 = cons(0,0x10000000,0,0,0,0,0,0);
    ethint256 z8 = cons(0x10000000,0,0,0,0,0,0,0);

    if((x.w7 >= z8.w7 && y.w7 >= z8.w7) || (x.w7 < z8.w7 && y.w7 < z8.w7) ){return lt(x,y); }
    if((x.w7 >= z8.w7 && y.w7 < z8.w7)){return res1; }
    if((x.w7 < z8.w7 && y.w7 >= z8.w7)){ return res0; }

    if((x.w6 >= z7.w6 && y.w6 >= z7.w6) || (x.w6 < z7.w6 && y.w6 < z7.w6 ) ){return lt(x,y); }
    if((x.w6 >= z7.w6 && y.w6 < z7.w6)){ return res1; }
    if((x.w6 < z7.w6 && y.w6 >= z7.w6)){ return res0; }

    if((x.w5 >= z6.w5 && y.w5 >= z6.w5) || (x.w5 < z6.w5 && y.w5 < z6.w5 ) ){ return lt(x,y); }
    if((x.w5 >= z6.w5 && y.w5 < z6.w5)){ return res1; }
    if((x.w5 < z6.w5 && y.w5 >= z6.w5)){ return res0; }
```

Imagen 26. Implementación comparación con signo

Para mostrar brevemente el funcionamiento correcto de estas operaciones se va a crear un programa en Solidity con números positivos y negativos que haga comparaciones. Se muestra a continuación.

```
pragma solidity ^0.4.11;

contract Sum {

function comparacion ()  {
        int x = 10;
        int y = -5;
        int z = y - x;
        assert( z <= -15 );
 }
}
```

Imagen 27. Ejemplo programa Solidity

Después de analizarlo con la herramienta y verificar el fichero C con CPAChecker se observa que la salida de éste es positiva.

4.2 Transformación de EVM a C

Para hacer la transformación se realiza en lenguaje Python un programa que va leyendo instrucción a instrucción de la RBR y se van cambiando por llamadas a funciones de C, almacenando todas en una pila para luego escribirlas en un fichero, el cual contendrá el código Solidity transformado a código C.

A continuación, se muestra un ejemplo de transformación. En la primera imagen se tiene un fragmento de la RBR, en concreto del $Block_0$, en el cual se hace una asignación a s_0 de 128 y se guarda en pila, lo mismo con s_1 de 64, luego los 128 se almacenan en memoria, otra asignación a s_0 de 4, y por último se hace introduce en pila la instrucción LT, y se hace un salto a $jump_0$.

$Jump_0$ recoge de la pila s_1, s_0 y la instrucción LT, y pasa a hacer la comprobación.

```
block0(l(l0), calldataload, calldatasize, callvalue)=>
        s(0) = 128
        nop(PUSH1)
        s(1) = 64
        nop(PUSH1)
        l(l0) = s(0)
        nop(MSTORE)
        s(0) = 4
        nop(PUSH1)
        s(1) = calldatasize
        nop(CALLDATASIZE)
        call(jump0(s(1),s(0), l(l0), calldataload, calldatasize, callvalue))
        nop(LT)
        nop(PUSH1)
        nop(JUMPI)
```

Imagen 28 Ejemplo bloque de la RBR

```
void block0(){
        s0 = ASSIGN(0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x80);
        s1 = ASSIGN(0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x40);
        l0 = COPY(s0);
        s0 = ASSIGN(0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x4);
        s1 = COPY(calldatasize);
        jump0();
}

void jump0(){
        if(LT(s1,s0)){
                block63(); }
        else {
                block12(); }
}
```

Imagen 29. Ejemplo transformación en C

En la segunda imagen se observa la misma secuencia, pero en código C. Como se puede ver no son el juego de instrucciones ni la sintaxis típica del lenguaje C, sino que son llamadas a la implementación que se ha desarrollado en este trabajo para poder trabajar con tamaños de palabra de 256 bits.

Para hacer dicho desarrollo se ha creado en C una estructura de datos con 8 palabras de 32 bits, de forma que se pueden recrear los 256 bits de las palabras de EVM. En la siguiente imagen se puede ver su implementación.

Inicialmente se pensó una versión mas sencilla, con una estructura de 4 palabras de 64 bits en C, lo cual también facilitaba la implementación de las operaciones. Pero, al comprobar si el verificador trabajaba bien con ellas se observó que no, y es que verificadores tan potentes como los que hay para C son muy precisos y muy sensibles a cualquier variación.

Es por lo que finalmente se realizó de nuevo el trabajo pero con palabras mas pequeñas dentro de la estructura.

Al ver las implementaciones del apartado anterio es normal pensar que se realizan muchos pasos por no utilizar ningún tipo de bucle, cuya implementación hubiese sido mucho más sencilla. La realidad es que se hicieron pruebas usando bucles y aunque funcionalmente las operaciones se comportaban correctamente, cuando se verificaba el programa en C las salidas que daba este no eran correctas. Por lo que se prefirió hacer implementaciones más largas y menos visuales a cambio de poder utilizar verificadores con los programas C que genera la herramienta.

De hecho, la mayor parte de dificultad de todas las implementaciones ha sido conseguir que el verificador funcionase bien con ellas, para lo cual ha habido que evitar uso de bucles, llamadas a otras funciones, recursividad, cortar la ejecución de la función en cuanto se encontrase la solución, evitar en algunos casos la concatenación de muchas condiciones…

Por otra parte, otro factor a tener en cuenta cuando se usa un verificador para analizar un programa es el tiempo.

Al ser una herramienta que analiza todas las posibles transiciones entre todos los posibles estados en los que se puede encontrar la ejecución de un programa con cualquier valor de entrada, es una herramienta que conlleva un tiempo, y a ese tiempo hay que sumarle el que “gasta” en la propia función de C, por lo que uno de los objetivos era evitar incrementar innecesariamente ese tiempo.

De esta forma se han ido realizando los distintos tipos de operaciones, para posteriormente comprobar su funcionamiento, y por último comprobar que el verificador trabaja bien con ellas. Pero de esto se mostrarán ejemplos gráficos en siguientes secciones. A continuación, se va a explicar cómo se ha tratado el bloque del *invalid* que se veía en el grafo.

Cuando el ensamblador de Ethereum, EVM, detecta que un programa intenta hacer una división por cero, acceder a una posición de un array que no existe, o que un tipo enumerado se sale de su rango de representación genera una instrucción *invalid* que frena inmediatamente la ejecución del programa, con las consecuencias de "perdidas" de gas que esto conlleva. Esta instrucción como se veía en el grafo de control de flujo pertenece a un bloque. Este bloque lo traduce la herramienta Ethir en la instrucción de la RBR *nop(ASSERTFAIL),* y posteriormente se ha traducido en la instrucción a la llamada *VERIFIER_ERROR* del programa C. Esta llamada es la que produce que el verificador cuando está analizando el código saque un resultado negativo, es decir, si hay alguna posibilidad de que EVM genere un *invalid* se refleja en el código C para que el verificador pueda detectarlo. En este caso la salida será negativa, mientras que si en ningún caso se podría llegar a generar un invalid el verificador obtendría una salida positiva.

Además, para permitir que el verificador no solo sea capaz de analizar esas tres propiedades, sino que sea capaz de analizar aquellas que el programador considere necesario, se puede incluir la instrucción *assert* o *require* en el codigo Solidity, y funcionarán de la misma forma que se ha explicado anteriormente.

Para ver mas claramente como es la traducción paso a paso de un invalid desde que lo genera la máquina virtual de Ethereum hasta que lo detecta un verificador de C se va a mostrar gráficamente a continuación.

Se comienza incluyendo la instrucción *require* en un programa en Solidity. Esta instrucción va a comprobar que el valor de entrada n siempre es menor que el tamaño del array a de entrada. Esta condición se puede poner antes de un bucle que recorra el array a en el rango de n valores para comprobar que en ningún caso se accede a una posición inexistente.

```
function foo (uint [] a, uint n) public {
    require(n < a.length);
    for (uint i = 0; i < n; i ++) {
        a[i] = 0;
    }
}
```

Imagen 30. Ejemplo programa Solidity con requiere

El framework Oyente al analizar el conjunto de instrucciones EVM que se producen con ese código de entrada genera el grafo de control de flujo, en el

cual hay un estado con la instrucción *invalid*, a la cual se llega por las ramas en las que n sea mayor que el tamaño del array.

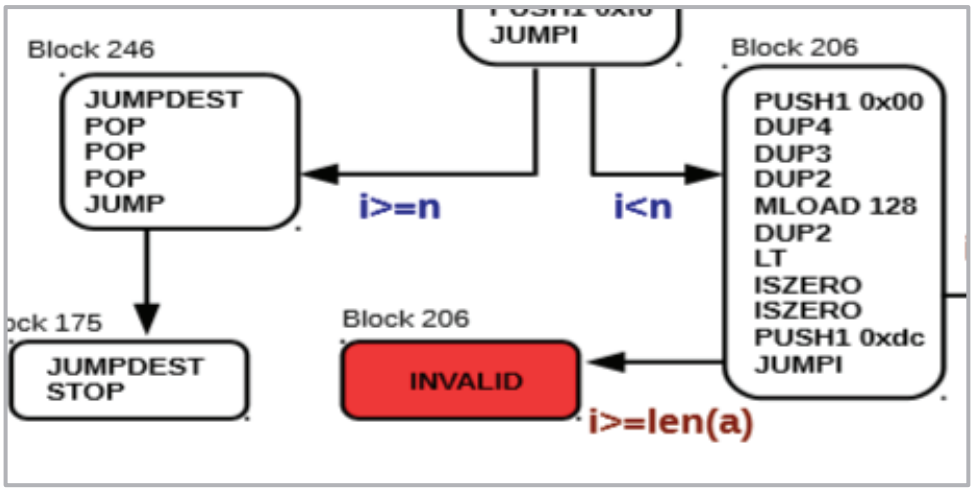


Imagen 31. Ejemplo bloque en CFG con invalid

Ethir al transformar esas instrucciones del grafo en instrucciones de la representación basada en reglas crea un bloque con la instrucción *invalid*.

$$block219(s_7, \ldots, s_0) \rightarrow nop(INVALID)$$

Imagen 32. INVALID en la RBR

Posteriormente en el trabajo realizado se convierte ese bloque de la RBR en uno del código C con una llamada a *VERIFIER_error*, que es el que produce que el verificador determine una salida negativa tras el análisis.

```
void block219(){
  __VERIFIER_error();
  //INVALID
}
```

Imagen 33. INVALID en el código C

Además, el verificador indica la linea donde encontró en el programa C la llamada a *VERIFIER_error(),* la cual puede indicar en que bloque de la RBR se produjo el error. Por lo que se puede ir trazando la ejecución hacia atrás para saber que linea del programa Solidity llevó a que el verificador diese una salida negativa, y así poder arreglarlo, todo ello sin haber "gastado" gas en la ejecución del programa, puesto que es un análisis estático.

4.3 Ejemplos de uso

Para poder estudiar en conjunto la ayuda que supone para alguien que desarrolla contratos inteligentes en Solidity poder verificarlos con verificadores tan potentes como CPAChecker, se van a mostrar algunos ejemplos de uso.

Se parte de la suposición de un programador que necesita una funcion que reciba un array y dos variables. A la variable x le aplica una serie de procesos como operaciones aritméticas, llamadas a otras funciones… y posteriormente necesita acceder a la posición x del array, como muestra la imagen.

```
pragma solidity ^0.4.11;

contract Sum {

function suma (uint[] nums, uint x, uint t) returns (uint sol) {
        sol = 0;
        x = x + t;
        nums[x] = sol;
 }

}
```

Imagen 34. Ejemplo programa Solidity

Obviamente cuando se crean contratos inteligentes no son tan pocas líneas de codigo ni programas tan sencillos, pero no es el objetivo de este documento explicar el desarrollo de un programa completo en Solidity. Aunque en este fragmento de código es muy intuitivo ver cuál es el problema, en un programa completo no es una tarea tan sencilla.

Suponiendo que el programador ejecuta dicho fragmento de código en la plataforma Ethereum, la ejecución del mismo será abortada, lo cual conlleva una “pérdida” del gas del que dispusiera. Esto se debe a que no se controla si el valor de x esta dentro del rango del array, pudiéndose acceder a una posición no existente.

Si antes de ejecutarlo se hiciera un análisis estático del programa se podría ver el error y solucionarlo antes de que produzca mayores problemas.

Para ello, se utilizaría SAFEVM, la cual utilizaría el framework Oyente para crear el grafo de control de flujo, posteriormente crearía la representación basada en reglas, y por último crearía el programa en código C con el tipo de variable creado en este trabajo y todas sus operaciones.

El grafo que se obtiene es el siguiente, como se puede ver tiene dos estados de terminación posibles, los bloques 190 y 245. Buscando esos bloques en la RBR se observa que el 190 se da cuando no ha habido ningún *invalid,* y al 245 se salta en caso de haberse producido un *invalid* durante la ejecución.

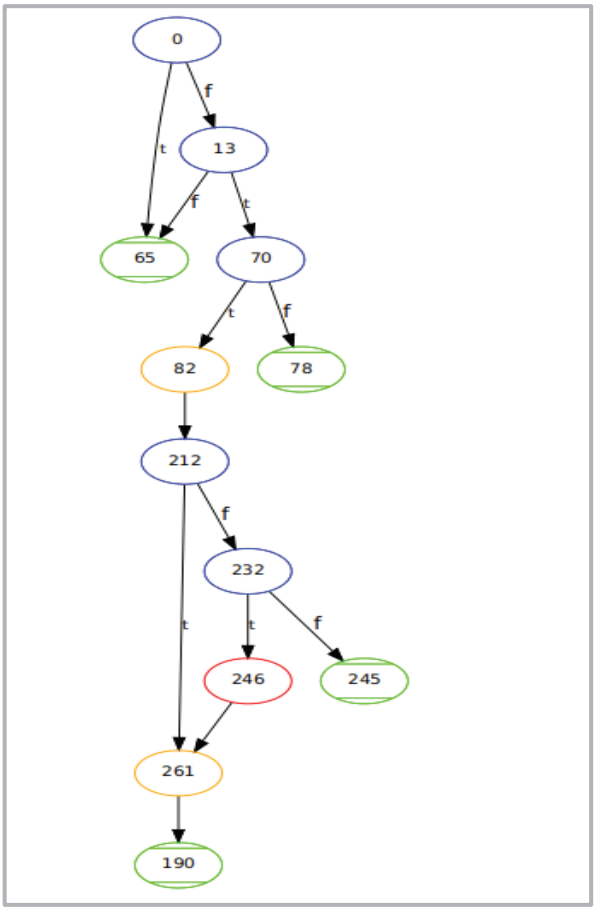


Imagen 35. CFG obtenido

```
block245(s(8), s(7), s(6), s(5), s(4), s(3), s(2), s(1), s(0))=>
        nop(ASSERTFAIL)
```

Imagen 36. Bloque de la RBR con invalid

Con este análisis estático se respetaría el tamaño de palabra máximo, 256 bits, por lo que no se pierde información, y se obtiene un fichero C. Este fichero C se puede analizar con CPAChecker, u otro verificador de C, y se obtendrá de salida *FALSE* junto con la línea del fichero C que la provocó. Con esa línea se puede encontrar el bloque de la RBR que lo produjo, pudiendo hallar el origen del problema para solucionarlo.

Una vez encontrado el problema la persona que lo ha programado añade una instrucción a su código para comprobar que siempre se accede a una posición el array existente.

```
pragma solidity ^0.4.11;

contract Sum {

function suma (uint[] nums, uint x, uint t) returns (uint sol) {
        sol = 0;
        x = x + t;
        if( x < nums.length )
           nums[x] = sol;
 }

}
```

Imagen 37 Ejemplo de programa Solidity

De nuevo se vuelve a hacer un análisis estático del programa.

El grafo de control de flujo y la RBR son muy similares a los anteriormente mostrados, la diferencia es que en este caso no llega a entrar nunca por la rama del bloque del *invalid*.

Finalmente se crea el fichero C y se verifica dicho fichero con CPAChecker. En este caso la salida es *TRUE*.

Ahora se puede ejecutar en la plataforma Ethereum sin temor a que haya una interrupción en la ejecución porque se produzca una instrucción *invalid*.

3 Resultados y conclusiones

Inicialmente cuando se propuso la idea y se comenzó el trabajo no se confiaba plenamente en que fuese a resultar del todo satisfactorio, no tanto por la transformación de la representación basada en reglas a un código C, o por la propia implementación de la recreación del ensamblador de C, sino porque los verificadores son herramientas realmente muy sensibles a cualquier cambio, y más los utilizados en este trabajo, que son de los verificadores más potentes ahora mismo en el mundo de la verificación.

Sin embargo, se iban haciendo las operaciones en C y se comprobaba que funcionaban a nivel funcional, con una serie de pruebas en codigo C, correctamente, y sobretodo que el verificador trabajaba bien con ellas y sin unos tiempos de análisis demasiado altos. Por lo que después de todas las implementaciones se integró esta especie de ensamblador en la herramienta Ethir para hacer la transformación de la RBR a un programa en C. Posteriormente se empezó a hacer pruebas de pequeños contratos Solidity con todo el proceso de la herramienta SAFEVM y posteriormente con el análisis del verificador CPAChecker, y aunque fallaban algunas cosas porque el verificador no las consideraba correctas y por lo que hubo que cambiar ciertos fragmentos de las operaciones, finalmente se consiguió que se analizasen bien contratos pequeños.

Luego, se comprobó que trabaja bien con contratos Solidity mas reales, no con operaciones tan simples, sino con un programa Solidity con varios contratos y con todo tipo de operaciones. Por lo que el trabajo ha obtenido muy buenos resultados, y por todo lo que se ha detallado en este documento puede llegar a suponer una herramienta muy eficaz para personas que ejecutan sus Smart Contracts en plataformas como Ethereum.

4 Bibliografía

[1] https://www.blockchain.com/es/

[2] https://bitcoin.org/es/

[3] https://solidity.readthedocs.io/en/v0.6.8/

[4] https://ethereum.org/

[5] https://www.doc.ic.ac.uk/~livshits/papers/pdf/atva18.pdf

[6] https://www.comp.nus.edu.sg/~prateeks/papers/Oyente.pdf

[7] https://cpachecker.sosy-lab.org/doc.php

[8] https://arxiv.org/pdf/1906.04984.pdf

[9] https://seahorn.github.io/

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Sat Jun 06 21:34:17 CEST 2020
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)