Análisis de la Privacidad y de la Seguridad en Contratos Inteligentes

Antonio López Vivar, Ana Lucila Sandoval Orozco, and Luis Javier García Villalba, *Member, IEEE*

***Resumen*—Los contratos inteligentes han ganado mucha popularidad en los últimos tiempos ya que son una herramienta muy potente para el desarrollo de aplicaciones seguras descentralizadas y automáticas en multitud de campos sin necesidad de intermediarios o terceros de confianza. No obstante, debido a la naturaleza descentralizada de la cadena de bloques en la que se basan, se han puesto de manifiesto una serie de retos relacionados con la privacidad de la información que manejan dichos contratos, así como vulnerabilidades en su programación que, dadas sus particularidades, podrían tener (y ya han tenido) un impacto económico muy elevado. Este artículo proporciona una visión holística de los retos en materia de privacidad y seguridad asociados con los contratos inteligentes, así como del estado del arte de las herramientas y soluciones disponibles.**

***Palabras claves*—Blockchain, privacy, security, smart contracts**

I. Introducción

Los contratos inteligentes está ganando popularidad en los últimos tiempos, aunque el concepto original es relativamente antiguo. La idea de los contratos inteligentes aparece en [1] y ha evolucionado hasta nuestros días, especialmente después de la introducción en 2009 de Bitcoin [2] y su cadena de bloques descentralizada. En pocas palabras, un contrato inteligente es un programa de ordenador que se ejecuta de forma descentralizada, modificando, si la ejecución es correcta, el estado global del sistema, que se almacena en una cadena de bloques. Aunque suele estar asociada a la plataforma Ethereum [3], hoy en día existen muchas plataformas que hacen uso de ellas, como por ejemplo Hyperledger o Corda entre muchas otras (en [4] puede consultarse una lista actualizada de todas las plataformas de contratos inteligentes existentes). En este artículo, nos centraremos en Ethereum, aunque mucho de los conceptos de seguridad son extrapolables al resto de plataformas.

En cuanto a las aplicaciones y casos de uso de los contratos inteligentes son muy variados pero su naturaleza descentralizada, velocidad, automatización, ausencia de intermediarios y transparencia los hacen particularmente adecuados en diferentes sectores, tales como: gestión de identidades [5] [6], voto electrónico [7] [8], servicios bancarios y financieros [9], gestión de herencias [10], cadena de suministro [11], IoT [12], juego en línea [3] e información médica [13] entre otros.

En el ejemplo del voto electrónico, sería posible mediante un contrato inteligente llevar la votación de unas elecciones y el recuento de forma automatizada, transparente y sin requerir de una autoridad de confianza.

Este trabajo está estructurado en 5 secciones, siendo la primera la presente introducción. En la Sección II se introducen las de cadenas de bloques y los contratos inteligentes. La Sección III se centra en los problemas de privacidad de los contratos inteligentes y las herramientas. La sección IV se centra en las vulnerabilidades en la programación de los contratos. Finalmente, en la Sección V se presentan las conclusiones del presente trabajo.

II. Contexto

Se presenta en esta sección una breve introducción de la cadena de bloques y los contratos inteligentes.

II-A. Tecnologías de Registro Distribuido

Una cadena de bloques es básicamente un registro replicado en múltiples nodos, llamados "mineros" que mantienen una copia (a veces no completa). Las operaciones, llamadas transacciones, se registran como en un libro de contabilidad, agrupándolas en bloques, que son añadidos a la cadena mediante un algoritmo de consenso basado habitualmente en una prueba de trabajo [14] Para realizar dicha prueba, cada uno de los nodos prepara un bloque candidato con un conjunto de transacciones nuevas pendientes de ser añadidas a la cadena de bloques al que le añade un contador y un puntero al último bloque de la cadena. Después cada nodo ejecuta sobre su bloque candidato un algoritmo de hash criptográfico (en el caso de Ethereum es KECCAK-256 [15]). Si el hash resultante es menor que un valor determinado, el bloque se considera minado y se propaga junto al hash calculado para ser verificado por el resto de nodos y que actualicen su copia de la cadena de bloques. En caso, contrario, si el hash resultante no es válido (lo más probable), se incrementa el contador del bloque y se vuelve a calcular el hash.

II-B. Contratos inteligentes

Como se ha mencionado, en la cadena de bloques se almacenan transacciones. En Ethereum, existen dos tipos de transacciones, las "normales", donde un usuario A envía una cantidad de "Ether" (una moneda virtual) a la dirección de un usuario B. Las direcciones de usuario constan básicamente de un par de claves pública/privada y cada transacción que se genera va firmada digitalmente por el usuario que envía los fondos. Además de las direcciones de usuario, existen las direcciones de contrato, que como su nombre indican apuntan a un contrato inteligente. Para añadir un nuevo contrato inteligente a la cadena de bloques, un usuario tiene

A. López Vivar, A. L. Sandoval Orozco and L. J. García Villalba. Grupo de Análisis, Seguridad y Sistemas (GASS), Departamento de Ingeniería del Software e Inteligencia Artificial, Facultad de Informática, Despacho 431, Universidad Complutense de Madrid (UCM), Calle Profesor José García Santesmases, 9, Ciudad Universitaria, 28040 Madrid, España. e-mail: {alopezvivar, asandoval, javiergv}@fdi.ucm.es.

https://doi.org/10.12804/si9789587844337.12

que generar un tipo de transacción especial con una serie de datos, entre los que se incluye el código fuente del contrato ya compilado. Para ejecutar un contrato inteligente, un usuario tendrá que crear una transacción desde su dirección de usuario enviando Ether (y los parámetros que necesite el contrato) hacia la dirección del contrato. Si la ejecución del contrato es correcta, se modificará el estado global de Ethereum. En caso contrario, si la ejecución falla, se le cobrará al usuario el coste computacional usado del Ether que envió en la transacción, pero no habrá cambios en el estado global del sistema.

Los contratos inteligentes pueden escribirse en diferentes lenguajes de programación alto nivel, siendo el lenguaje más utilizado Solidity [16], con una sintaxis muy parecida a JavaScript. Este lenguaje es compilado a un bytecode que será ejecutado por la máquina virtual de Ethereum (EVM).

II-C. Un contrato inteligente de ejemplo

En la Figura 1 se presenta un ejemplo de un contrato inteligente muy básico que va a servir de ejemplo.

simple sc.sol

```
1  contract SimpleStorage {
2      uint storedData;
3
4      function set(uint x) public {
5          storedData = x;
6      }
7
8      function get() public view returns (uint) {
9          return storedData;
10     }
11 }
```

Figura 1. Contrato inteligente de ejemplo

Puede verse que se declara una variable de tipo entero sin signo que en Ethereum tiene 256 bits. Después se declara una función pública para asignar el valor de la variable anterior y otra función que devuelve el valor de la variable. Cada vez que algún usuario invoca este contrato y asigna un nuevo valor, éste se actualiza. Aunque en este ejemplo no se usan, hay mecanismos en los contratos inteligentes para restringir el acceso a determinadas funciones en función de quién ejecute el contrato inteligente.

Un dato importante a recordar es que antes de añadir un nuevo contrato a la cadena de bloques, los contratos se ejecutan de forma local en cada nodo minero y el resultado de la ejecución debe ser determinista para garantizar la coherencia de todo el sistema. Esta propiedad puede ser un problema a la hora de generar números aleatorios dentro de los contratos inteligentes. Por último, faltaría hablar del concepto de *gas*. Solidity es un lenguaje Turing-completo, por lo que existe la posibilidad de provocar bucles infinitos en la ejecución. Una manera de evitar esto, lo cual podría ser utilizado como forma de ataque de denegación de servicio contra la red de Ethereum, existe un mecanismo mediante el cual, cada una de las operaciones de bytecode posee un coste prefijado llamado *gas* Cuando un usuario invoca un contrato, tiene que pasarle el precio unitario del gas y el límite de gas que está dispuesto a asumir. Si durante la ejecución del contrato, el *gas* alcanza el máximo que fijó el usuario, la ejecución se detiene lanzando una excepción y el usuario que lanzó la ejecución pierde el gas multiplicado por el precio unitario. En caso de que la ejecución termine, después de pagado el precio del gas usado, el Ether sobrante será devuelto al usuario. Los mineros por su parte reciben unos honorarios para recompensarles por mantener la red. Sus honorarios están definidos por el *gas* y el precio del *gas* Si el atacante intenta lanzar un ataque de denegación de servicio y elige un precio de *gas* acorde al mercado, los mineros ejecutarán el ataque pero el precio del ataque será muy alto. Por otro lado, si el atacante eligiera un precio del *gas* muy bajo, los mineros no incluirían su transacción en ningún bloque y por tanto no se ejecutaría el ataque.

III. Privacidad en contratos inteligentes

Los contratos inteligentes que se ejecutan en cadenas de bloques públicas, como es el caso de Ethereum, sufren de problemas de privacidad tanto en las transacciones, que quedan registradas en la cadena de bloques, como en el propio código fuente de los contratos. Si bien es cierto que las claves utilizadas para las transacciones de Ethereum no están asociadas a personas físicas concretas, hay varios estudios que analizando la cadena de bloques han podido correlacionar dichas transacciones y agruparlas [17], [18], [19]. Además, el propio código fuente de los contratos inteligentes, puede contener datos o claves de carácter privado y muy sensible, como es por ejemplo el caso de los contratos inteligentes asociados a dispositivos médicos. Dicho código fuente, por la propia arquitectura de la cadena de bloques se encuentra replicado para su ejecución en múltiples nodos repartidos por la red, por lo que se encuentran expuestos a fugas de información. En las secciones que vienen a continuación se hará una descripción de algunas de las herramientas más relevantes que existen actualmente para mitigar estos problemas de privacidad de los contratos inteligentes.

III-A. Hawk

Hawk es un framework para preservar la privacidad en los contratos inteligentes que funciona como una capa de abstracción para que el desarrollador escriba el código del contrato inteligente y después el compilador de Hawk creará tres programas a saber: el contrato inteligente que se ejecutará en la cadena de bloques, un programa que se ejecutará por lo usuarios y otro programa que será ejecutado por un manager (ver Figura 2).

Como puede verse en la Figura 2, un contrato de Hawk consta de dos partes diferenciadas: una parte privada donde se encuentran todos los datos sensibles que no queremos que sean visibles en la cadena de bloques y una parte pública donde se manejan datos de carácter menos sensible o públicos.

Para conseguir privacidad en la cadena de bloques, Hawk utiliza principalmente criptografía y pruebas de conocimiento zero para garantizar la integridad y el correcto funcionamiento de los contratos. Por otro lado, la privacidad a nivel de contrato no sólo se centra en usar criptografía para proporcionar confidencialidad si no que añade la figura del manager, una entidad que vigila comportamientos maliciosos de otros agentes, pero sin tener capacidad para modificar la correcta ejecución de los contratos inteligentes.

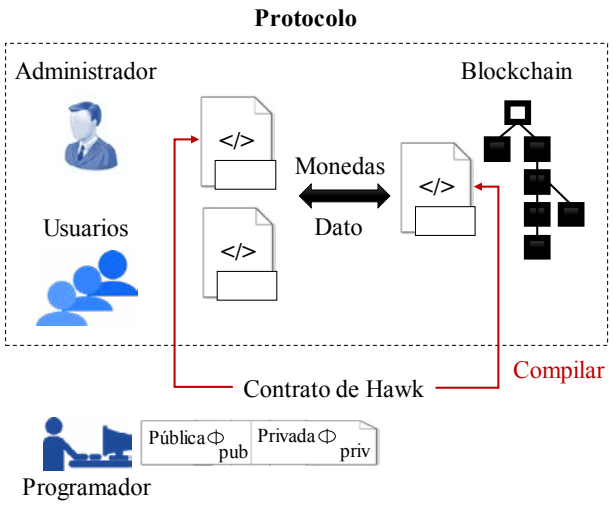

Figura 2. Esquema de funcionamiento de Hawk [20]

III-B. Ekiden

Ekiden [21], de forma parecida a Hawk, se trata de un entorno de ejecución segura de contratos inteligentes, pero a diferencia de Hawk que basa su área privada de ejecución en criptografía y pruebas de conocimiento cero, Ekiden apuesta por el uso de Entornos de Ejecución de Confianza [22] para ejecutar las partes más sensibles de los contratos inteligentes.

Como puede verse en la Figura 3, existen tres tipos de entidades en Ekiden: por un lado estarían los clientes, que serían los usuarios, los cuales pueden crear y ejecutar contratos. En segundo lugar se encontrarían los nodos de ejecución segura, que se encargarían de ejecutar los contratos y que deben disponer de una plataforma de ejecución en entorno de confianza. Por último, estarían los nodos de consenso, encargados de validar los resultados de ejecución de los nodos de computación y de añadir la información y cambios de estados del sistema a la cadena de bloques.

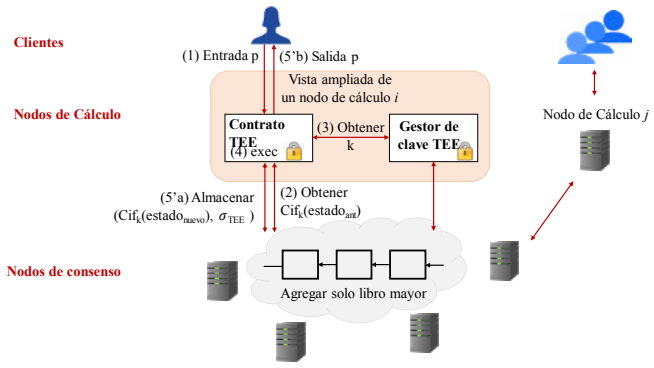

Figura 3. Arquitectura de Ekiden [21]

III-C. Enigma

Enigma [23] por su parte, utiliza computación multi-agente [24] para proveer seguridad en la ejecución de los contratos inteligentes. Los datos son segmentados y repartidos en varios nodos aleatoriamente entre varios nodos, de forma que ningún nodo tiene nunca la totalidad de la información. Enigma separa la ejecución de ciertas partes intensivas en una cadena de bloques diferente de la principal, donde se guardan los resultados y las pruebas de verificación.

IV. Seguridad en contratos inteligentes

Aunque la cadena de bloques está diseñada de forma segura y dicha seguridad está sustentada en algoritmos criptográficos ampliamente estudiados y probados, los contratos inteligentes como programas informáticos son susceptibles de contener vulnerabilidades de seguridad en su código, que dada su naturaleza inmutable y unido al hecho de que pueden funcionar en ámbitos financieros o de salud, supone una amenaza de seguridad grave, como ya ha sucedido en el pasado.

Un ejemplo famoso de ataque en Ethereum fue el *DAO attack* que se produjo en 2016 mediante el cual se robaron 3,6 millones de Ethers (el equivalente a unos 70 millones de dólares) aprovechando una vulnerabilidad en la programación de un contrato inteligente que controlaba el balance de una fundación. El contrato no tenía en cuenta la posibilidad de hacer llamadas recursivas que extraían dinero y no actualizaban el total, por lo que el hacker pudo realizar muchas de estas llamadas hasta vaciar los fondos. En [25] se hace un análisis detallado de este ataque.

En [26] se hace una clasificación de las vulnerabilidades existentes en los contratos inteligentes:

- Llamadas a lo desconocido: ocurre que algunas primitivas del lenguaje Solidity usadas para llamar a otras funciones o enviar Ether pueden sufrir de un efecto lateral de llamar a una función definida por defecto que tienen todos los contratos (y cuyo código podría ser desconocido para el llamador), en caso de no encontrarse la función llamada. Las primitivas afectadas por este efecto son:
  - CALL: es una primitiva utilizada para llamar a las funciones de un contrato inteligente (del mismo o de otro). Si la función que se le pasa como parámetro a la primitiva, no existe en el contrato, se ejecutará una función por defecto o *fallback*.
  - SEND: esta primitiva permite enviar Ether desde el contrato en ejecución a otro destinatario. Una vez que se ha enviado la cantidad de Ether, se ejecuta la función por defecto en el contrato de destino.
  - DELEGATECALL: esta primitiva es muy parecida a CALL a diferencia de que en ésta, se utiliza el contexto de variables del contrato llamador.
- Envío sin gas: al enviar Ether usando la primitiva SEND podría producirse una excepción por agotamiento de *gas* si el destinatario es un contrato tiene una función de *fallback* con mucho código.
- Desorden en las excepciones: en Solidity existen varias situaciones que pueden provocar que se dispare una excepción durante la ejecución, a saber: si la ejecución del contrato se queda sin *gas*, si se agota la pila de llamadas o en caso de que se lance la excepción de forma explícita llamando al comando *throw*. No obstante, Solidity no trata de la misma forma las excepciones si éstas se producen durante una llamada a una función de forma directa o usando la primitiva CALL. En el primer caso, la ejecución se detiene y cualquier efecto lateral se

revierte, incluyendo las transferencias de Ether. Pero si la excepción se produjo en el contexto de una llamada usando CALL, la excepción se propagará hacia arriba revirtiendo los efectos en los contratos llamados hasta alcanzar la llamada CALL, devolviendo *false* y continuará la ejecución a partir de ahí y consumiendo *gas*. Esta inconsistencia a la hora de manejar las excepciones puede dar lugar a vulnerabilidades.

- Conversión de tipos: aunque el compilador de Solidity puede detectar errores con los tipos, por ejemplo si una función espera un entero y es llamada pasándole una cadena, en el caso de definiciones de contratos o funciones con una determinada estructura, en caso de llamar a una función en un contrato, si el programador se equivoca y llama por error a otro contrato pero que contiene una función con la misma estructura esperada por el compilador se ejecutará la función y en caso de que no exista la función, se llamará a la función de *fallback* En cualquier caso, no se lanzará ninguna excepción.
- Reentrada: esta es una vulnerabilidad muy conocida por su impacto. El programador puede pensar que una función no recursiva no puede ser re-llamada mientras se está ejecutando, pero esto no es siempre así, ya que podría darse el caso de que dentro de la función se llame a un contrato malicioso vacío que sólo contenga una función de *fallback* que vuelva a llamar a la función de la que viene. Por ejemplo, supongamos que tenemos un contrato así:

```
contract Bob {
    bool sent = false;
    function foo(address c) {
        if (!sent) {
            c.call.value(1)();
            sent = true;
        }
    }
}
```

La función *foo* recibe como parámetro la dirección de un contrato y si el testigo no está activado, envía 1 *wei* (la mínima unidad de Ether) al contrato c.
El problema viene si el contrato llamado es algo así:

```
contract Alice {
    function() {
        Bob(msg.sender).foo(this)
    }
}
```

En este caso, después de recibir el Ether, Alice llama su función de *fallback*, lo que a su vez vuelve a llamar a la función *foo* de Bob y como el testigo no está puesto a *true* volverá a transferir a Alice Ether y esto se repetirá hasta que se agote el *gas* o se alcance el límite de la pila de llamadas. Esta vulnerabilidad se utilizó el Ataque DAO comentado más arriba y en [25] se explica en detalle.

- Secretos: Solidity permite definir la visibilidad de los campos en los contratos como públicos o privados. Esto puede ser útil si se necesita ocultar determinada información entre llamadas de contratos. Desgraciadamente este sistema no es efectivo ya que los cambios en campos privados tienen que enviarse a nodos mineros para que estos los metan en la cadena de bloques, la cual es pública. Aquellos contratos que necesiten ocultar información sensible tendrán que usar técnicas como las mencionadas en la Sección III
- Estado impredecible: todos los contratos inteligentes tienen un estado determinado por el valor de sus campos y su balance de Ether. Pero no se puede garantizar que el estado que tenía un contrato cuando hicimos una transacción hacia él será el mismo que cuando dicha transacción sea minada e incluida en la cadena de bloques. Es decir, podría ocurrir que antes de procesar nuestra transacción, otras transacciones hayan cambiado el estado del contrato de destino y además ser rápidos no nos garantiza nada porque los mineros pueden minar las transacciones en el orden que quieran. Se da otro problema añadido por la naturaleza de la cadena de bloques, y es que podría producirse un *fork* de la cadena si dos mineros coniguen minar un bloque válido al mismo tiempo. Esto provocaría que algunos mineros intentarían añadir su bloque en una de las dos cadenas y los otros en la otra. Llegado un momento la cadena más corta se desecharía, perdiendo las transacciones contenidas en esta y cambiando el estado de los contratos a un estado indeterminado. Otro caso serían contratos que utilizan librerías dinámicas (un tipo especial de contratos que no pueden tener campos mutables). Ese tipo de contratos podrían cambiar de forma maliciosa para engañar a la víctima que los llamaría sin saber que han cambiado.
- Números aleatorios: la ejecución del código de la máquina virtual de Ethereum es determinista. Eso quiere decir, que el código ejecutado con las mismas entradas debe de producir las mismas salida en todos los nodos que lo ejecuten. Esto presenta un problema a la hora de generar número aleatorios. Para simular aleatoriedad, muchos contratos utilizan un generador de números aleatorios inicializado con la misma semilla para todos los mineros. Una opción muy utilizada por los programadores es utilizar como semilla el hash de un bloque determinado en el futuro. Al tratarse de un valor impredecible a priori, es una buena manera de inicializar el generador de números aleatorios. Sin embargo, como los mineros pueden elegir qué transacciones meter en los nuevos bloques, podrían conspirar para intentar conseguir alterar el funcionamiento del generador de números aleatorios.
- Restricciones de tiempo: muchas aplicaciones tienen restricciones de tiempo para operar. Habitualmente esas restricciones utilizan *timestamps*. En el caso de los contratos inteligentes, el programador puede obtener el *timestamp* de cuándo se minó el bloque, que es compartido por todas las transacciones del bloque. El problema es que los mineros en las primeras versiones del protocolo podían elegir el *timestamp* del bloque que iban a minar de forma arbitraria, lo que podía usarse para llevar a cabo ataques.
- Bugs inmutables: esto no es una vulnerabilidad en sí misma, sino la consecuencia de una propiedad de la ca-

dena de bloques. Todo el código fuente de los contratos inteligentes, incluidos aquellos que contienen *bugs* son inmutables una vez son minados y añadidos a la cadena de bloques, aunque si pueden ser bloqueados mediante la llamada a una función destructor.

- Pérdida de Ether: si el programador se equivoca al introducir la dirección para enviar Ether y esa dirección existe pero es una dirección huérfana que no pertenece a nadie ese Ether se perderá para siempre.
- Tamaño de pila: cada vez que un contrato llama a otro contrato la pila de llamada asociada aumenta en uno. El límite de la pila es 1024 y cuando se llega al límite se lanza una excepción. Hasta el 18 de octubre de 2016 era posible aprovecharse de esto para lanzar un ataque donde un usuario malicioso incrementaba el contador de la pila hasta casi agotarlo y entonces llamaba a la función de la víctima lo que lanzaba una excepción al agotar el límite de la pila. Si la víctima no tuvo en cuenta esto y no maneja correctamente la excepción, podría tener éxito el ataque. El impacto de esta vulnerabilidad hizo que se rediseñara Ethereum.

A fecha de creación de este documento existen multitud de herramientas para este propósito. A continuación se hará una pequeña descripción de los métodos o técnicas utilizados por las herramientas de análisis y reflejados en la tabla II y después de se hará una breve descripción de las herramientas.

IV-A. Métodos de análisis

- Ejecución simbólica: en vez de usar valores concretos para las variables se utilizan símbolos. Las operaciones sobre estos símbolos conducen a términos algebraicos, y las declaraciones condicionales dan lugar a fórmulas propositivas que caracterizan a las ramas. Una parte particular del código es alcanzable si la conjunción de fórmulas en el camino a esta parte es satisfactoria, lo que puede ser comprobado por los solucionadores de SMT.
- Interpretación asbtracta: mediante el uso de Árboles de Sintaxis Abstractos que se obtienen en una fase intermedia en el proceso de compilación de Solidity en bytecode se analiza en el código en busca de vulnerabilidades.
- Reglas Horn: la lógica de Horn [27] está compuesta por una versión restringida de la lógica de primer orden donde todas las reglas son de la forma si-entonces que aunque limitada es computacionalmente universal y puede llevar a cabo los mismos cálculos que cualquier ordenador.
- Resolución de restricciones: mediante esta técnica las herramientas intentan resolver una serie de condiciones en el código para determinar si la ejecución podría tomar determinados caminos que conduzcan a potenciales vulnerabilidades.
- Verificación de modelo: este método se basa en una verificación automática de las propiedades de un sistema de estados finitos utilizando un modelo del sistema que se cruza con un conjunto de especificaciones.

IV-B. Herramientas

Oyente [28] es una de las primeras herramientas de seguridad que aparecieron. Ejecuta el bytecode EVM de los contratos inteligentes de forma simbólica y se centra en comprobar bloques de código susceptibles de ser vulnerables por su estructura, llamadas a contratos externos o una mala gestión de las excepciones de ejecución, por ejemplo la Figura 4. Disponible en Github [29] desde enero de 2016 (licencia GPL-3.0).

Remix-IDE es una extensión para el navegador web que permite escribir contratos inteligentes en el lenguaje Solidity. Muestra de forma gráfica distintos avisos de potenciales vulnerabilidades o fallos en el código a la vez que hace un análisis estático ligero. Disponible en Github [30] desde abril de 2016 (licencia MIT).

Solgraph [31] es una herramienta que permite visualizar el flujo ejecución de un contrato inteligente escrito en Solidity y ver de forma más fácil potenciales vulnerabilidades. Está disponible en Github [32] desde julio de 2016.*Porosity [33]* es una herramienta que desensambla el bytecode EVM y genera un diagrama de flujo. También permite descompilar el bytecode y convertirlo a código fuente. Disponible en Github [34] desde febrero de 2017.

Tabla I
VULNERABILIDADES DETECTADAS POR LAS HERRAMIENTAS

		Oyente	Remix-IDE	Solgraph	Porosity	Manticore	SmartCheck	FSolidM	Mythril	ContractLarva	E-EVM	SolMet	Vandal	EthIR	MAIAN	Erays	Rattle	Osiris	Securify	Ethertrust
Solidity	Llamadas a lo desconocido	✗	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
	Envío sin gas	✗	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
	Desorden en excepciones	✗	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
	Conversión tipos	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Reentrada	✓	✓	✗	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✗
	Conversión tipos	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Revelación de secretos	✗	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Blockchain	Estado impredecible	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	Números aleatorios	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Restriciones de tiempo	✓	✓	✗	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
EVM	Bugs inmutables	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Pérdida de Ether	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗
	Tamaño de pila	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Manticore [35] es una herramienta de ejecución simbólica que en el caso de la EVM, analiza la ejecución del código del contrato inteligente en busca de las vulnerabilidades mas típicas apoyándose del solucionador SMT Z3 [36] La herramienta está disponible en Github [37] desde febrero de 2017 (licencia AGPL-3.0).

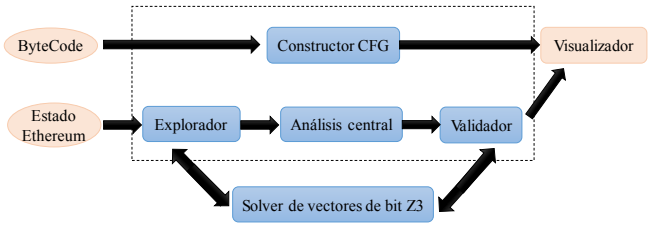

Figura 4. Esquema de funcionamiento de Oyente [28]

SmartCheck [38] busca patrones de vulnerabilidades conocidas en el código fuente. Para ello utiliza un archivo XML con el código fuente en forma de árbol y explora caminos que pueden llevar a vulnerabilidades durante la ejecución. Disponible en Github [39] desde mayo de 2017 (licencia GPL-3.0).

FsolidM [40] utiliza una máquina de estados finita para permitir al usuario definir el comportamiento del contrato inteligente y a partir de ésta generar de forma automática el código en Solidity del contrato inteligente. Está disponible en Github [41] desde septiembre de 2017 (licencia MIT).

Mythril [42], es una herramienta muy parecida a Manticore, que permita la ejecución simbólica del bytecode EVM y que genera un grafo de control de flujo. Todo esto permite detectar una serie de tipos de vulnerabilidades. Desarrollado por la empresa ConsenSys, se encuentra disponible en Github [43] desde septiembre de 2017.

contractLarva [44] Es una herramienta que verifica los contratos inteligentes en tiempo de ejecución. Para ello, al compilar el contrato, añade al código fuente original, una serie de instrucciones encargadas de velar durante la ejecución de que no se producen vulnerabilidades y/o evitarlas dentro de lo posible sin alterar el flujo de ejecución normal del contrato. Está disponible en Github [45] desde diciembre de 2017 (licencia Apache-2.0).

E-EVM [46] es una heramienta que permite la ejecución del bytecode EVM de forma visual, creando un grafo de flujo y mostrando información sobre la pila de ejecución, a modo de depurador. Está disponible en Github [47] desde enero de 2018.

SolMet [48] se trata de una herramienta para calcular la complejidad del código fuente de un contrato escrito en Solidity que utiliza un parser para generar un árbol de sintaxis abstracta donde evalúa una serie de métricas para medir la complejidad. Disponible en Github [49] desde febrero de 2018.

Vandal [50] es un desensamblador y descompilador de Solidity que genera un grafo de control de flujo que puede visualizarse como una página web HTML. Además permite especificar los análisis de seguridad de forma lógica usando el lenguaje Soufflé [51]. Disponible en Github [52] desde febrero de 2018.

EthIR [44] es un framework que transforma el código fuente de los contratos inteligentes a un lenguaje intermedio. Está basado en la herramienta Oyente pero utiliza una representación basada en reglas . Se encuentra disponible en Github [53] desde marzo de 2018 (licencia GPL-3.0).

MAIAN [54] es una herramienta muy parecida a Oyente, pero que la amplia teniendo en cuenta los ataques que requieren de varias transacciones. Se apoya en el uso de una cadena de bloques privada para testeo y así mitigar los falsos positivos. Al igual que otras muchas herramientas utiliza el solucionador SMT Z3 para buscar caminos en la ejecución que lleven a potenciales vulnerabilidades. Está disponible en Github [55] desde marzo de 2018 (licencia MIT).

Erays [56] es un desensamblador de bytecode EVM que genera un archivo PDF con pseudocódigo de las rutinas presentes en los contratos inteligentes. Se encuentra disponible en Github [57] desde agosto de 2018 (licencia MIT).

Rattle [58] es un framework que hace análisis estático binario del bytecode EVM realizando un desensamblado que elimina instrucciones no necesarias para entender el funcionamiento del código fuente del contrato, aunque no detecta vulnerabilidades por sí mismo. Disponible en Github [59] desde agosto de 2018 (licencia GPL-3.0).

Tabla II
RESUMEN DE LAS HERRAMIENTAS DE ANÁLISIS

		Oyente	Remix-IDE	Solgraph	Porosity	Manticore	SmartCheck	FSolidM	Mythril	ContractLarva	E-EVM	SolMet	Vandal	EthIR	MAIAN	Erays	Rattle	Osiris	Securify	Ethertrust
Nivel	Bytecode	✓	✗	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓
	Solidity	✗	✓	✓	✗	✗	✓	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Tipo	Análisis dinámico	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
	Análisis estático	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Algoritmo	Ejecución simbólica	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓	✗	✗	✓	✗	✗
	Interpretación abstracta	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
	Instrumentación de código	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	Reglas Horn	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✓	✓
	Resolución de restricciones	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗	✓	✓	✓	✗	✗	✓	✗	✓
	Verificación de modelo	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Osiris [60] es una herramienta especializada en detectar vulnerabilidades relacionadas con números enteros dentro de los contratos inteligentes de Ethereum. Funciona ampliando la funcionalidad de Oyente analizando el flujo de ejecución para distinguir desbordamientos considerados benignos de aquellos malignos. Disponible en Github [61] desde septiembre de 2018.

Securify [62] realiza primero un desensamblado del bytecode EVM. Después, el código desensamblado lo descompila en un lenguaje intermedio con el que crea un conjunto de reglas DataLog como las mostradas en la figura 6 que usará para buscar la violación de patrones y así encontrar vulnerabilidades. Disponible en Github [63] desde septiembre de 2018 (licencia Apache-2.0).

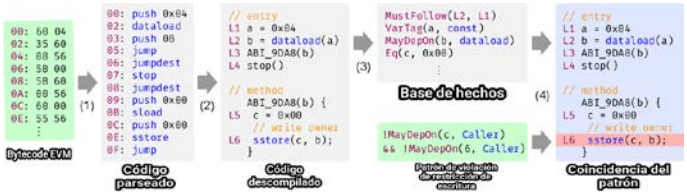

Figura 5. Análisis de código Securify [62]

EtherTrust [64] es un framework que basa su funcionamiento en traducir el bytecode EVM los contratos inteligentes en cláusulas de Horn que junto al solucionador SMT Z3 verifica que el código del contrato no presente potenciales vulnerabilidades aunque no las detecta. Está disponible en Github [65] desde agosto de 2019 (licencia GPL-3.0).

En la tabla I basada en la taxonomía de vulnerabilidades de [26] puede verse un resumen de todas las vulnerabilidades que las herramientas de seguridad analizadas detectan. Algunas de ellas, como por ejemplo FSolidM o ContractLarva no detectan ninguna porque no es su función.

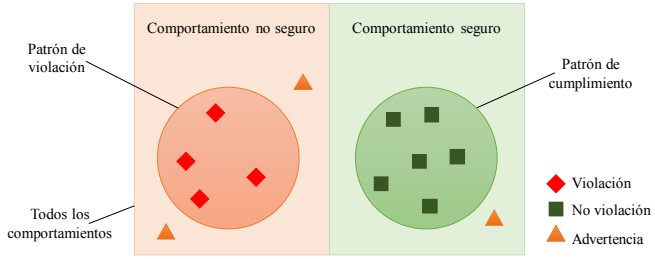

Figura 6. Sistema de patrones basados en reglas que utiliza Securify [62]

V. Conclusiones y trabajos futuros

La seguridad y privacidad de contratos inteligentes es un campo de investigación relativamente nuevo, pero con este documento pretendemos dejar patente que ya existen multitud de herramientas y soluciones, algunas de ellas bastante maduras. Es cierto, que con tanta variedad de soluciones, el desarrollador de contratos inteligentes puede a priori sentirse confundido y no saber por dónde empezar, aunque trabajos como este podrían ser un buen punto de partida.

En el caso de las herramientas de análisis de vulnerabilidades, aunque pueda parecer en principio que existe mucha variedad, muchas de ellas comparten algoritmos y técnicas de análisis y otras son evoluciones de herramientas más antiguas. Además, a la hora de desarrollar nuevas herramientas, existen ya librerías que aceleran el proceso como el desensamblador de código de Ethereum de la Fundación Ethereum.

Por último, como trabajo futuro, quedaría pendiente el estudio de todas aquellas herramientas no disponibles públicamente (y/o cuyo código fuente no es público) y que no se han incluido en este trabajo.

Agradecimientos

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 700326. Website: http://ramses2020.eu. This paper has also received funding from THEIA (Techniques for Integrity and authentication of multimedia files of mobile devices) UCM project (FEI-EU-19-04).

Referencias

[1] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.
[2] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[3] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, p. 37, 2014.
[4] Coinlore, "Smart contract platforms."
[5] M. Al-Bassam, "Scpki: a smart contract-based pki and identity system," in *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. ACM, 2017, pp. 35–40.
[6] P. Dunphy and F. A. Petitcolas, "A first look at identity management schemes on the blockchain," *IEEE Security & Privacy*, vol. 16, no. 4, pp. 20–29, 2018.
[7] P. McCorry, S. F. Shahandashti, and F. Hao, "A smart contract for boardroom voting with maximum voter privacy," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 357–375.
[8] N. Kshetri and J. Voas, "Blockchain-enabled e-voting," *IEEE Software*, vol. 35, no. 4, pp. 95–99, 2018.
[9] G. W. Peters and E. Panayi, "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money," in *Banking beyond banks and money*. Springer, 2016, pp. 239–278.
[10] P. Sreehari, M. Nandakishore, G. Krishna, J. Jacob, and V. Shibu, "Smart will converting the legal testament into a smart contract," in *2017 International Conference on Networks & Advances in Computational Technologies (NetACT)*. IEEE, 2017, pp. 203–207.
[11] T. Bocek, B. B. Rodrigues, T. Strasser, and B. Stiller, "Blockchains everywhere-a use-case of blockchains in the pharma supply-chain," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 772–777.
[12] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *Ieee Access*, vol. 4, pp. 2292–2303, 2016.
[13] N. Rifi, E. Rachkidi, N. Agoulmine, and N. C. Taher, "Towards using blockchain technology for ehealth data access management," in *2017 Fourth International Conference on Advances in Biomedical Engineering (ICABME)*. IEEE, 2017, pp. 1–4.
[14] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 3–16.
[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak specifications," *Submission to nist (round 2)*, pp. 320–337, 2009.
[16] C. Dannen, *Introducing Ethereum and Solidity*. Springer, 2017.
[17] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, 2017.
[18] F. Reid and M. Harrigan, "An analysis of anonymity in the bitcoin system," in *Security and privacy in social networks*. Springer, 2013, pp. 197–223.

[19] M. Möser, K. Soska, E. Heilman, K. Lee, H. Heffan, S. Srivastava, K. Hogan, J. Hennessey, A. Miller, A. Narayanan *et al.*, "An empirical analysis of traceability in the monero blockchain," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 3, pp. 143–163, 2018.
[20] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.
[21] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 185–200.
[22] G. Arfaoui, S. Gharout, and J. Traoré, "Trusted execution environments: A look under the hood," in *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2014, pp. 259–266.
[23] G. Zyskind, O. Nathan, and A. Pentland, "Enigma: Decentralized computation platform with guaranteed privacy," *arXiv preprint arXiv:1506.03471*, 2015.
[24] Y. Lindell, "Secure multiparty computation for privacy preserving data mining," in *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005, pp. 1005–1009.
[25] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.
[26] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.
[27] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 343–361.
[28] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 254–269.
[29] Melon Project, "Oyente," https://github.com/melonproject/oyente, 2016.
[30] Ethereum Foundation, "Remix-ide," https://github.com/ethereum/remix-ide, 2018.
[31] R. Revere, "Solgraph," https://github.com/raineorshine/solgraph, 2018.
[32] R. Revere, "Solgraph," https://github.com/raineorshine/solgraph, 2016.
[33] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, p. 11, 2017.
[34] Comae Technologies, "Porosity," https://github.com/comaeio/porosity, 2017.
[35] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," *arXiv preprint arXiv:1907.03890*, 2019.
[36] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
[37] T. of Bits, "Manticore," https://github.com/trailofbits/manticore.
[38] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2018, pp. 9–16.
[39] SmartDec, "Smartcheck," https://github.com/smartdec/smartcheck, 2017.
[40] A. Mavridou and A. Laszka, "Tool demonstration: Fsolidm for designing secure ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 270–277.
[41] A. Mavridou and A.Laszka, "Fsolidm," https://github.com/anmavrid/smart-contracts, 2017.
[42] B. Mueller, "Smashing smart contracts," in *9th HITB Security Conference*, 2018.
[43] ConsenSys, "Mythril," https://github.com/ConsenSys/mythril-classic, 2017.
[44] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2018, pp. 513–520.
[45] G. Pace, "contractlarva," https://github.com/gordonpace/contractLarva, 2017.
[46] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, "Visual emulation for ethereum's virtual machine," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–4.
[47] pisocrob, "E-evm," https://github.com/pisocrob/E-EVM, 2018.
[48] P. Hegedus, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, p. 6, 2019.
[49] P. Hegedus, "Solmet," https://github.com/chicxurug/SolMet-Solidity-parser, 2018.
[50] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.
[51] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 422–430.
[52] Smart Contract Research (USYD), "Vandal," https://github.com/usyd-blockchain/vandal, 2018.
[53] P. Gordillo, "Ethir," https://github.com/costa-group/EthIR, 2018.
[54] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 653–663.
[55] MAIAN-tool, "Maian," https://github.com/MAIAN-tool/MAIAN, 2018.
[56] H. Gans, "Poole and plans: Erays on urban solutions and problems," 1968.
[57] teamnsrg, https://github.com/teamnsrg/erays, 2018.
[58] R. Stortz, "Rattle - an Ethereum EVM binary analysis framework," https://github.com/crytic/rattle, 2018.
[59] Trail of Bits, "Rattle," https://github.com/trailofbits/rattle, 2018.
[60] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 664–676.
[61] C. Ferreira, "Osiris," https://github.com/christoftorres/Osiris, 2018.
[62] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 67–82.
[63] SRI Lab, "Securify," https://github.com/eth-sri/securify, 2018.
[64] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.
[65] SecPriv, "Ethertrust," https://github.com/SecPriv/EtherTrust, 2019.

Antonio López Vivar received his Computer Engineering degree at Universidad Carlos III of Madrid (2011) and Master's Degree in Security of Information and Communication Technologies at Universidad Europea of Madrid (2015). Currently he is a Ph.D. student in the Department of Software Engineering and Artificial Intelligence of the Faculty of Computer Science and Engineering at the Universidad Complutense de Madrid (UCM) and Member of the Complutense Research Group GASS (Group of Analysis, Security and Systems, http://gass.ucm.es). working as Research Support Staff in the Group of Analysis, Security and Systems (GASS) of Universidad Complutense of Madrid. His research interests are: blockchain, cryptocurrencies, computer forensics and cybersecurity.

Ana Lucila Sandoval Orozco was born in Chivolo, Magdalena, Colombia in 1976. She received a Computer Science Engineering degree from the Universidad Autónoma del Caribe (Colombia) in 2001. She holds a Specialization Course in Computer Networks (2006) from the Universidad del Norte (Colombia), and holds a M.Sc. in Research in Computer Science (2009) and a Ph.D. in Computer Science (2014), both from the Universidad Complutense de Madrid (Spain). She is currently a postdoctoral researcher and member of the Research Group GASS (Group of Analysis, Security and Systems, http://gass.ucm.es) at Universidad Complutense de Madrid (Spain). Her main research interests are coding theory, information security and its applications.

Luis Javier García Villalba received a Telecommunication Engineering degree from the Universidad de Málaga (Spain) in 1993 and holds a Ph.D. in Computer Science (1999) from the Universidad Politécnica de Madrid (Spain). Visiting Scholar at COSIC (Computer Security and Industrial Cryptography, Department of Electrical Engineering, Faculty of Engineering, Katholieke Universiteit Leuven, Belgium) in 2000 and Visiting Scientist at IBM Research Division (IBM Almaden Research Center, San Jose, CA, USA) in 2001 and 2002, he is currently Associate Professor of the Department of Software Engineering and Artificial Intelligence at the Universidad Complutense de Madrid (UCM) and Head of Complutense Research Group GASS (Group of Analysis, Security and Systems) which is located in the Faculty of Computer Science and Engineering at the UCM Campus. His professional experience includes the management of both national and international research projects and both public (Spanish Ministry of R&D, Spanish Ministry of Defence, Horizon 2020 - European Commission, . . .) and private financing (Hitachi, IBM, Nokia, Safelayer Secure Communications, TB Solutions Security, . . .). Author or co-author of numerous international publications is editor or guest editor of numerous journals such as Entropy MPDI, Future Generation Computer Systems (FGCS), Future Internet MDPI, IEEE Latin America Transactions, IET Communications (IET-COM), IET Networks (IET-NET), IET Wireless Sensor Systems (IET-WSS), International Journal of Ad Hoc and Ubiquitous Computing (IJAHUC), International Journal of Multimedia and Ubiquitous Engineering (IJMUE), Journal of Supercomputing, Sensors MDPI, etc.