Meta-Herramienta para el Análisis de Vulnerabilidades en Contratos Inteligentes

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
CURSO 2019–2020

Alberto Turégano Castedo

Directores
Luis Javier García Villalba
Antonio López Vivar

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Madrid, Junio de 2020

Agradecimientos

Quiero agradecer el incondicional apoyo mostrado por todos los integrantes del Grupo GASS en la realización del presente Trabajo Fin de Grado y, en particular, a mis directores, Javier y Antonio, por la labor realizada, y, especialmente a éste último, por haber sido además un compañero y un gran apoyo durante el desarrollo del trabajo, siempre estando disponible para cualquier duda y compartiendo su conocimiento para que el resultado fuera el mejor posible.

Índice General

Índice de Figuras xi

Índice de Tablas xiii

Lista de Códigos Fuente xiv

Lista de Acrónimos xvii

Abstract xix

Resumen xxi

1. Introducción 1

1.1. Motivación 1

1.2. Contexto 2

1.3. Objeto de la Investigación 2

1.4. Plan de Trabajo 2

1.5. Estructura del Trabajo 3

2. La Cadena de Bloques 5

2.1. Conceptos básicos 5

2.1.1. Criptomonedas 5

2.1.1.1. Bitcoin 5

2.1.2. Cartera Digital 6

2.1.3. Transacción 6

2.1.4. Salida de Transacción no Gastada 6

2.1.5. Bloque 7
2.1.6. Minado y Prueba de Trabajo 8
2.1.7. Proceso de Verificación 9
2.2. Clases de Cadenas de Bloques 10
2.3. Ethereum 11
2.3.1. Características de Ethereum 11
2.3.2. Contratos Inteligentes 12
2.3.3. Máquina Virtual de Ethereum 13

3. Seguridad **15**
3.1. Seguridad en la Cadena de Bloques 15
3.1.1. Gobiernos y Empresas 15
3.1.2. Procesos Basados en Arquitecturas de Cadena de Bloques 15
3.1.3. Tecnologías 16
3.2. Seguridad en Ethereum y en Contratos Inteligentes 16
3.2.1. Principios de Seguridad en Contratos Inteligentes 17
3.2.2. Incidentes de Seguridad en Ethereum 18
3.2.3. Vulnerabilidades relacionadas con Solidity 19
3.2.4. Vulnerabilidades Inherentes a la Cadena de Bloques 21
3.2.5. Vulnerabilidades que Afectan a la *Ethereum Virtual Machine* (EVM) 22
3.3. Herramientas de Análisis de Vulnerabilidades en Ethereum 23
3.3.1. Métodos de Análisis 23
3.3.2. Slither 24
3.3.3. Oyente 25
3.3.4. Remix - IDE 26
3.3.5. Solgraph 26
3.3.6. Manticore 27
3.3.7. Smartcheck 27
3.3.8. Mythril 27
3.3.9. SolMet 28

3.3.10. Vandal . . . 30
3.3.11. EthIr . . . 31
3.3.12. Maian . . . 31
3.3.13. Erays . . . 32
3.3.14. Rattle . . . 33
3.3.15. Osiris . . . 35
3.3.16. Securify . . . 36
3.3.17. MadMax . . . 38
3.3.18. ContractLarva . . . 39
3.3.19. Conclusiones Sobre las Herramientas . . . 41

4. Meta-Herramienta de Análisis de Vulnerabilidades en Contratos Inteligentes 43

4.1. Justificación y Alcance . . . 43
4.2. Entorno y Tecnologías . . . 43
4.3. Funcionamiento . . . 44
4.3.1. Contenedores . . . 45
4.3.1.1. Dockerfile . . . 45
4.3.1.2. Imagen Genérica . . . 46
4.3.1.3. Imágenes Personalizadas y Ejecución . . . 47
4.3.1.3.1. Subida y Descarga del Repositorio . . . 47
4.3.2. Implementación . . . 48
4.3.2.1. Diagrama de Flujo y Propiedades Previas al Análisis . . . 48
4.3.2.2. Diagramas de Flujo y Propiedades Durante el Análisis . . . 50
4.3.3. Interfaz Gráfica . . . 56
4.3.4. Funciones y Flujo de Uso . . . 56
4.3.5. Código fuente . . . 58
4.3.6. Almacenamiento en Base de Datos . . . 61
4.3.6.1. Mydb . . . 61
4.3.6.2. Results db . . . 62
4.4. Descarga e Instalación . . . 64

4.5. Dificultades y Limitaciones 64
4.6. Pruebas de la meta-herramienta 67

5. Conclusiones y Trabajo Futuro **73**
5.1. Conclusiones 73
5.2. Trabajo Futuro 74

6. Introduction **75**
6.1. Motivation 75
6.2. Context 76
6.3. Object of the Investigation 76
6.4. Work Plan 76
6.5. Structure of the Work 77

7. Conclusions and Future Work **79**
7.1. Future Work 80

A. Funcionamiento de las Herramientas **81**
A.1. Slither 81
A.2. Oyente 81
A.3. Remix - IDE 83
A.4. Solgraph 84
A.5. Manticore 84
A.6. Smartcheck 84
A.7. Mythril 86
A.8. SolMet 87
A.9. Vandal 87
A.10.EthIr 87
A.11.Erays 87
A.12.Rattle 88
A.13.Osiris 88
A.14.Securify 88

A.15.MadMax 89
A.16.ContractLarva 91

Bibliografía **93**

Índice de Figuras

2.1. Ejemplo de una transacción en la cadena de bloques de Bitcoin 7

2.2. Extrato del contenido del bloque 600391 de la red Bitcoin. 8

2.3. Proceso de verificación del protocolo Bitcoin. 10

3.1. Proceso de generación de las claves de una cuenta 17

3.2. Parte de los resultados de un análisis con Mythril. 28

3.3. Ejemplo de análisis con SolMet 29

3.4. Resultados visualizados en excel 29

3.5. Ejecución de desasemblado para el archivo daohack.hex. 30

3.6. Listado de diferentes opciones para ethir 32

3.7. Resultados de la ejecución de Erays. 33

3.8. Resultados de salida tras el análisis con Rattle 34

3.9. Ampliación de un de los archivos de salida 34

3.10. Resultados de un análisis con Osiris. 35

3.11. Resultados de un análisis con Securify. 37

3.12. Resultados de un análisis con Securify con el flag json. 38

3.13. Diagrama de ejecución generado por Mad Max 39

3.14. Contrato antes de usar ContractLarva. 40

3.15. Contrato después de usar ContractLarva. 40

4.1. Diagrama de flujo: lanzamiento de app.py hasta el comienzo de análisis . . 51

4.2. Diagrama de flujo: vista general del proceso de análisis 52

4.3. Diagrama de secuencia: flujo de trabajo de la aplicación 55

4.4. Maqueta inicial 56

4.5. Selección de la opción de análisis local 57
4.6. Selección de la opción de análisis por dirección 58
4.7. Dockerfile sin modificar. 66
4.8. Primera parte del análisis. 71
4.9. Segunda parte del análisis. 71
4.10. Tercera parte del análisis. 72

A.1. Ejemplo de un análisis con Slither 82
A.2. Uso del flag *print humman-summary* 82
A.3. Ejemplo de análisis con Oyente 83
A.4. Vista principal del entorno. 83
A.5. Parámetros de configuración. 84
A.6. Contrato para ser analizado. 85
A.7. Resultado de la ejecución de Solgraph 85
A.8. Librería de Smartcheck. 86
A.9. Resultados con Smartcheck. 86
A.10. Opciones de desasemblado. 88
A.11. Opciones de decompilado. 89
A.12. Comandos disponibles en Osiris. 90
A.13. Contenido de un fichero DAE 91

Índice de Tablas

3.1. Los ocho detectores más importantes de Slither 25

3.2. Resumen de las herramientas analizadas 42

4.1. Scripts y descripción usados por la herramienta 48

4.2. Parámetros de entrada de la meta-herramienta 49

4.3. Parámetros de entrada de la meta-herramienta 50

Lista de Códigos Fuente

2.1. Ejemplo de contrato inteligente 13
3.1. Contrato inteligente de ejemplo 18
3.2. Ejemplo del ataque reentrancy 1/2 19
3.3. Ejemplo del ataque reentrancy 2/2 20
3.4. Ejemplo de código Solidity seguro 20
3.5. Ejemplo de código Solidity inseguro 21
4.1. Docker file 46
4.2. Descarga de imágenes 47
4.3. Manejo de la base de datos 49
4.4. Ejemplo del contenido para la herramienta solmet 50
4.5. Pseudocódigo exec_start 53
4.6. Pseudocódigo intermediate_exec 54
4.7. Pequeño fragmento del archivo gui.py 59
4.8. Funciones del archivo gui_controller.py 60
4.9. Función de descarga dentro de contract_downloader.py 61
4.10. Ejemplo en formato json de un bloque 62
4.11. Ejemplo simplificado en formato json de un análisis 63
4.12. Resultado del análisis para el contrato 100 68

Lista de Acrónimos

BTC	*Bitcoin*
CA	*Contract Account*
CSV	*Comma Separated Values*
DAO	
	Decentralized Autonomous Organization
	Data Access Object
DLT	*Distributed Ledger Technology*
ELF	*Executable and Linkable Format*
EOA	*Externally Owned Account*
ETH	*Ether, la criptomoneda de Ethereum*
EVM	*Ethereum Virtual Machine*
P2P	*Peer to Peer*
POC	*Proof of Concept*
POO	*Programación Orientada a Objetos*
PoW	*Proof of Work*

RBR *Rule Based Representation*

SC *Smart Contract*

SDK *Software Development Kit*

SSA *Single Static Assignment*

TX *Transaction*

UTXO *Unspent Transaction Output*

Abstract

Bitcoin was born in 2008 creating a new concept around cryptocurrencies world. Bitcoin was the first cryptocurrency which is supported by Blockchain. It is a decentralized, autonomous and secure system giving the possibility to make transactions between two individuals in any part of the world without an intermediate government. Bitcoin was the genesis and after that new cryptocurrencies with some new features appeared. One of them was Ethereum, created by Vitalik Buterin, which added a new functionality never saw in a public Blockchain at that time called smarts contracts. Smart contracts gives a new address types which works as Blockchain users with a specific function. The facts to add new programmable variables to Blockchain generates new potential errors because of his nature: bugs, stolen accounts (and corresponding money loss), bad contract uses, wrong workflows, etc. Summarizing, this generates new issues that have to be resolved effectively. Developers around the world and Ethereum foundation started to create security tools to make new Solidity smart contracts secures. Different Solidity developers who want to use this kind of tools has to install each one separately, learn how to use them and execute each tool with each contract. This consumes a lot of time, resources, and effectiveness, so the main thing here was to create a unique tool which put together every security tool mentioned before into one single point. This tool could work in both cases such as command line or a visual interface. Command line is though to analyse existing contract already deployed in Ethereum Blockchain and visual interface to analyse local contracts.

Keywords: Analysis, Blockchain, Ether, Ethereum, Ethereum Compiler, Security, Smart Contract, Solidity, Wei.

Resumen

Con el nacimiento en el 2008 de Bitcoin surgió un nuevo paradigma tecnológico alrededor del mundo de las criptomonedas (o criptodivisas). Bitcoin fue la primera criptomoneda que se apoyaba en la cadena de bloques, un sistema descentralizado, autónomo y seguro que permite realizar transacciones entre dos individuos en cualquier parte del mundo sin necesidad de intermediarios con total seguridad. Bitcoin fue el pionero, apareciendo más tarde otro tipos de criptomoneda con características diferentes como es el caso de Ethereum. Esta nueva cadena de bloques desarrollada por su creador Vitalik Buterin añadía una nueva funcionalidad nunca vista en una cadena de bloques, los contratos inteligentes. Los contratos inteligentes otorgan a una red de cadena de bloques la inclusión de nuevos tipos de direcciones que pueden funcionar como usuarios de la red con una funcionalidad específica. El hecho de añadir más variables a la cadena de bloques y que estas sean programables generan de forma natural una serie de potenciales peligros: errores de programación, robos de cuentas (con su correspondiente pérdida de dinero), usos malintencionados de contratos, flujos de trabajo erróneos, etc. En definitiva, se abría paso un problema que debe solucionarse de forma efectiva mediante la búsqueda de aquellos errores y fallos (vulnerabilidades) en los contratos inteligentes. Desarrolladores del mundo y la propia fundación de Ethereum se pusieron a crear diferentes tipos de herramientas de análisis de seguridad informática centradas en contratos inteligentes escritos en Solidity, el lenguaje que maneja Ethereum. Los diferentes desarrolladores que quieran hacer uso de cada herramienta deben instalarlas por separado, aprender a usarlas y ejecutar análisis de forma individual, es decir, hacer un proceso de aprendizaje por cada herramienta nueva, lo cual, en tiempo y recursos es realmente alto. Aquí es donde surge el uso de la herramienta desarrollada como parte del trabajo de fin de grado, proporcionando un estudio completo de la mayor parte de las herramientas actualmente existentes, creando una herramienta tanto por línea de comandos para el análisis masivo de contratos ya desplegados en la cadena de bloques de Ethereum como el análisis de contratos locales. La herramienta provee una forma sencilla de realizar estudios para conocer cual es el estado actual de la red como para analizar un contrato que quiera ser desplegado teniendo todos los recursos unificados.

Palabras clave: Análisis, Cadena de Bloques, Compilador Ethereum, Contratos Inteligentes, Ether, Ethereum, Gas, Seguridad, Solidity, Wei.

Capítulo 1

Introducción

1.1. Motivación

El corto periodo de tiempo desde la creación de la cadena de bloques, 12 años, unido a la reciente expansión del paradigma tecnológico ha provocado una corriente de nuevos sistemas basados en su arquitectura. En 2020 se posiciona dentro de las tecnologías más prometedoras junto con la inteligencia artificial, aprendizaje automático, los macro-datos o el 5G entre otras.

Las cualidades que lo hacen especial (descentralización, permutabilidad y seguridad) crean nuevos negocios, puestos de trabajo y soluciones que hacen que sea un gran nicho de mercado emergente para las empresas tecnológicas. Las diferentes soluciones que se han ido adoptando para su implementación son muy variadas puesto que su aplicabilidad también lo es dando lugar a nuevos protocolos diferentes a Bitcoin, en concreto Ethereum que proporciona la capacidad de programar e introducir dicho código a la red para que aplicaciones externas puedan interactuar con la cadena de bloques, automatizar procesos e intercambiar información con agentes externos.

Ethereum al igual que la cadena de bloques es una tecnología muy moderna, tras la primera prueba de concepto se fueron sucediendo otras pruebas con el fin de hacer la red más eficiente con la creación por ejemplo del lenguaje que hoy en día maneja la red, Solidity. El proyecto, de código libre, ha tenido que hacer frente a una adopción cada vez mayor. Ahora mismo se encuentra inmerso en un proceso evolutivo complejo llamado Ethereum 2.0. Esta nueva versión de la red trae cambios muy significativos, dejará atrás su actual protocolo de consenso, permitirá el uso de POS y mejorará la tasa de transacciones por segundo que es uno de los principales problemas actuales en comparación con otras redes.

Este crecimiento y actualización de la red va acompañado también de la adopción de Solidity. La versión actual del lenguaje es la 0.6, siendo esta todavía inferior a la 1. Solidity está sujeto todavía a cambios que puedan provocar vulnerabilidades de forma natural por el uso de flujos de programación erróneos, uso de directivas en puntos donde no se deben usar, asignaciones erróneas o uso excesivo del gas. Por ello, la importancia de contar con herramientas que ayuden a los programadores para poder hacer frente a sus desarrollos es fundamental. A lo largo del tiempo han surgido diferentes herramientas, todas ellas sin tener un hilo común o un punto central donde llevar a cabo un único análisis que garantice pasar unas pruebas para todas las vulnerabilidades encontradas. Además, no se

ha realizado ningún estudio que haya obtenido resultados positivos de la situación actual de los contratos desplegados en la red, número de fallos por contrato, tasas de fallos, errores más comunes, porcentaje de la reducción de los fallos en función de la versión de Solidity, etc.

1.2. Contexto

Este Trabajo Fin de Grado ha sido realizado dentro del Grupo de Análisis, Seguridad y Sistemas (Grupo GASS, https://gass.ucm.es/, Grupo 910623 del catálogo de grupos reconocidos por la UCM) como parte de las actividades del proyecto de investigación THEIA (Techniques for Integrity and Authentication of Multimedia Files of Mobile Devices) con referencia FEI-EU-19-04.

1.3. Objeto de la Investigación

Con el rápido crecimiento de Solidity dentro de la red de Ethereum se precisa de herramientas que proporcionen todo el potencial de análisis de forma sencilla para un programador que desee analizar su contrato. Ahora mismo la información que hay es dispersa, y para un programador novel puede resultar muy complejo iniciarse en el mundo de la seguridad de contratos inteligentes. Es por ello que se precisa de un único sitio donde sintetizar todos los estudios que se han ido creando por separado, dando respuestas rápidas a cuáles son los fallos que se puede encontrar y cuales son las herramientas que pueden detectarlos. Cuando se habla de herramientas se habla tanto de un documento técnico como una herramienta informática que guíe al usuario en el proceso de análisis.

1.4. Plan de Trabajo

El desarrollo de este trabajo de fin de grado se ha realizado en tres fases:

1. **Aprendizaje**: La motivación principal para la selección de la cadena de bloques como trabajo de fin de grado surgió como lectura de un libro donde se hablaba del nacimiento de Bitcoin, Ethereum y Ethereum 2.0 y todas las bondades que proporciona la tecnología. Desde ese momento con la primera reunión se buscó durante aproximadamente 4 meses aprender todo lo posible acerca de Ethereum. Para ello se realizó un curso online en la plataforma Coursera llamado *Blockchain Basics* por la Universidad de Búfalo, así como la parte dos del programa llamada *Smarts Contracts*. Además, como formación complementaria se fueron buscando de forma más concreta los diferentes aspectos que componen el proyecto, elementos básicos de seguridad, diferentes sistemas de consenso, uso de hash... Finalmente, se decidió unir el aspecto de la seguridad informática junto con la cadena de bloques, al considerar podría salir un trabajo muy solido y completo.

2. **Estudio y experimentación de herramientas**: Tras la elección comenzó el proceso de estudio de cual era la situación exacta en el aspecto de la seguridad. Para ello se leyeron diferentes estudios ya publicados que han servido de pilar para la realización de este proyecto. Esos estudios están basados en las vulnerabilidades encontradas,

graves fallos históricos en Ethereum como son el *Ataque DAO (del inglés DAO)* o la vulnerabilidad de *reentrada* y las herramientas que hay actualmente disponibles. En cuanto a las herramientas se centran más en el estudio que hay por debajo de cada una, es decir, cómo funcionan, pero no proporcionan una clarificación de si su uso merece la pena o no. Una vez investigado lo suficiente para tener conocimiento de cual era el contexto actual, se realizó un estudio completo de la instalación, comandos, vulnerabilidades admitidas y una serie de puntos de cada herramienta. Gracias a este proceso se publicó un artículo del que soy coautor [LVCSOGV20].

3. **Desarrollo**: Tras la publicación el último paso ha sido la creación de una herramienta que simplificara al desarrollador el análisis individualizado. Para ello se planteó una solución basada en contenedores y su modularización. El sistema de gestión de contenedores ha sido *Docker* y para poder interactuar con cada contenedor se ha usado la librería de *Python* y el *Software Development Kit* (SDK) de Docker para Python. Durante este periodo han surgido muchas obstaculizaciones técnicas derivadas de las dependencias y de los desarrollos de cada uno de los creadores. La solución final se ha visto siempre condicionada a las limitaciones que las herramientas ofrecían (versiones del compilador, gestión de extensiones de archivos, formatos, versiones del código de Solidity). Finalmente, para el uso más simplificado se decidió añadir una pequeña interfaz para poder analizar contratos locales y contratos ya desplegados en Ethereum, además de los correspondientes contratos ya insertados en la base de datos.

1.5. Estructura del Trabajo

El resto del trabajo está organizado en 5 capítulos y un anexo con la estructura que se comenta a continuación:

El Capítulo 2 introduce algunos conceptos que son elementales para comprender el funcionamiento de la cadena de bloques y los contratos inteligentes. Así, se describen las criptomonedas, el Bitcoin, las carteras digitales, las transacciones, las salidas de las transacciones, los bloques, el proceso de minado, la verificación de los bloques, los tipos de cadenas de bloques y se presenta Ethereum, con sus características, los contratos inteligentes y la máquina virtual de Ethereum.

El Capítulo 3 comienza haciendo una introducción de los mecanismos y tecnologías generales en las cadenas de bloques para después centrarse en la seguridad en Ethereum y los contratos inteligentes, donde se habla de los principios de seguridad de los contratos inteligentes, los incidentes de seguridad más famosos de Ethereum y los distintos tipos de vulnerabilidades diferenciándolas según si afectan a la cadena de bloques, a los contratos inteligentes o a la máquina virtual de Ethereum. Por último el capítulo termina con un estudio de las herramientas de análisis de vulnerabilidades en Ethereum, haciendo primero una introducción de los métodos de análisis utilizados por las herramientas para después ir describiendo una por una cada herramienta con una comparativa final.

El Capítulo 4 presenta la contribución principal de este trabajo, a saber, la *Meta-Herramienta de Análisis de Vulnerabilidades en Contratos Inteligentes*, empezando por una justificación y el alcance de la herramienta desarrollada, pasando por el entorno y tecnologías utilizadas, el funcionamiento de la herramienta, detalles de implementación y dificultades y limitaciones entre otros apartados.

Para terminar, en el Capítulo 5 se presentan las principales conclusiones de este trabajo así como líneas de trabajo futuro.

Capítulo 2

La Cadena de Bloques

La cadena de bloques es un tipo de tecnología de registro ditribuido (del inglés *Distributed Ledger Technology* (DLT)) Las DLT son unas estructuras de datos replicadas y sincronizadas en múltiples nodos dentro de una red y un conjunto de métodos para manipularlos. Las propiedades más notables de una DLT son la inmutabilidad, la resistencia a la censura, la gestión descentralizada y la ausencia del requisito de un tercero fiable. Existen varios tipos de DLT según el tipo de estructura de datos utilizada (los más comunes son las listas vinculadas y las redes acíclicas dirigidas), pero todos ellos comparten tres características: el uso de criptografía de clave pública, la utilización de redes *Peer to Peer* (P2P) y un mecanismo de consenso.

2.1. Conceptos básicos

2.1.1. Criptomonedas

Cada sistema de cadena de bloques posee un activo o token, comúnmente conocido como criptomoneda como medio de pago en las transacciones y generación de negocio y valor de la propia red. A día de hoy existen multitud de criptomonedas, aunque la más antigua y conocida y en la que se basan muchas es el Bitcoin, del que hablaremos a continuación.

2.1.1.1. Bitcoin

Bitcoin es sistema basado en una arquitectura de cadena de bloques, comúnmente conocido por su criptomoneda, que casualmente tiene el mismo nombre que el propio protocolo (Bitcoin) nacido en 2009. Previamente su creador bajo el nombre de Satoshi Nakamoto lanzó un artículo[Nak08] donde describía la arquitectura del protocolo Bitcoin. En dicho articulo a modo de introducción, Satoshi comentaba lo siguiente: *"Una forma de dinero en efectivo electrónico puramente P2P debería permitir enviar pagos online directamente entre las partes y sin pasar a través de una institución financiera. Las firmas digitales son parte de la solución, pero los beneficios principales desaparecen si un tercero de confianza sigue siendo imprescindible para prevenir el doble gasto. Proponemos una solución para el problema del doble gasto usando una red P2P..."*

Es decir, Bitcoin introdujo la cadena de bloques, (en inglés *Blockchain*), dando la posibilidad de realizar transacciones entre nodos (individuos) dentro de una red descentralizada. Esta red genera una confianza entre nodos que no se conocen entre sí, almacenando las transacciones dentro de un proveedor (base de datos P2P) que garantice la inmutabilidad de la información. Los nodos, como se ha dicho previamente no tienen por qué conocerse lo que garantiza que sea una red descentralizada. Para garantizar la confianza entre los nodos deberá haber un proceso de validación, verificación, consenso y confirmación de las transacciones.

En base a la cita de Satoshi Nakamoto, se irán desgranando las ventajas y bondades un protocolo basado en la cadena de bloques, poniendo como ejemplo Bitcoin (por ser el más conocido) para la compresión total de esta cita.

2.1.2. Cartera Digital

La forma de identificar a una persona o individuo en una transacción es mediante una cartera digital o (en inglés *wallet*). Esta no es más que hash que identifica a un individuo, dicho número es único y no está relacionado con ningún dato público de carácter personal. Hay diferentes formas de poseer una *wallet*, mediante un proveedor o tercero que guarde las criptomonedas por ti, o de forma local en un dispositivo físico.

2.1.3. Transacción

Una transacción (del inglés *Transaction* (TX)) es el proceso de enviar una cierta cantidad de criptomonedas de una cartera digital a otra. Como puede verse en la Figura 2.1 una transacción almacena diferentes tipos de información tales como:

- ID de la transacción: hash del identificador único.
- Descripción y metadatos.
- Salida de transacción sin gastar (del inglés *Unspent Transaction Output* (UTXO)) como valor de entrada.
- UTXO como valor de salida.
- Sello de tiempo (en inglés *timestamp*) de la transacción.
- Número de bloque donde se encuentra.

2.1.4. Salida de Transacción no Gastada

Las UTXO definen las entradas y salidas de una transacción. Es la forma en la que tiene Bitcoin en concreto (puesto que otras monedas como Ethereum no lo posee) de realizar las transacciones.

Por ejemplo, si tenemos un usuario llamado Antonio al cual Alicia envía 1 *Bitcoin* (BTC), el protocolo reconocerá que hay 1 BTC que todavía no se le ha asignado a nadie. Si Antonio poseía previamente 1 BTC, el nuevo balance cuando se valide la transacción será de 1BTC + 1BTC. Por lo tanto, el balance de las carteras de Bitcoin se calcula como

Figura 2.1: Ejemplo de una transacción en la cadena de bloques de Bitcoin

la suma de todos los UTXO de entrada que haya tenido un usuario, menos los UTXO de salida.

En el caso de que Antonio quiera crear una cantidad de 2 BTC a partir del 1+1, deberá de generar otra transacción que reciba 2 UTXOs de entrada (1BTC + 1BTC) y genere UTXO de salida con valor 2.

Por último, hay que añadir que Ethereum, moneda de la cual todavía no hemos hablado, funciona con el método comúnmente conocido de tener un balance inicial y sumar o restar la cantidad que enviamos o recibamos.

2.1.5. Bloque

El concepto de cadena de bloques es un concepto realmente sencillo.

A la hora de ir validando todas las transacciones que se producen en un sistema descentralizado necesitamos garantizar un histórico en caso de que queramos comprobar diferentes datos de las transacciones, además de garantizar que no hay modificaciones o alteraciones que puedan poner en riesgo el sistema. Es aquí donde surge el concepto de bloque, el cual almacena todas las transacciones que quedan pendiente de añadirse a la cadena, y que en un momento determinado, un minero ”mina”dicho bloque, permitiendo añadirlo a la cadena.

Más adelante se hablará de los conceptos de minado y prueba de esfuerzo, por el momento lo más importante es entender que **hay una única cadena raíz que va añadiendo progresivamente los bloques nuevos**, validando las transacciones que quedaban pendientes por realizarse.

Los elementos que forman parte de un bloque como puede verse en la Figura 2.2 son:

- Número de transacciones que contiene el bloque.
- Cantidad de salida de BTCs.
- Altura del bloque dentro de la cadena (muy parecido a un contador).
- *Timestamp* temporal de la generación del bloque.
- Minero que ha resuelto el bloque.
- Dificultad a la hora de minar el bloque.
- Tamaño del bloque en bits.
- *Nonce* (un contador).
- Hash del bloque actual.
- Hash del bloque padre y del bloque hijo.
- Raiz del *Merkle Tree*.

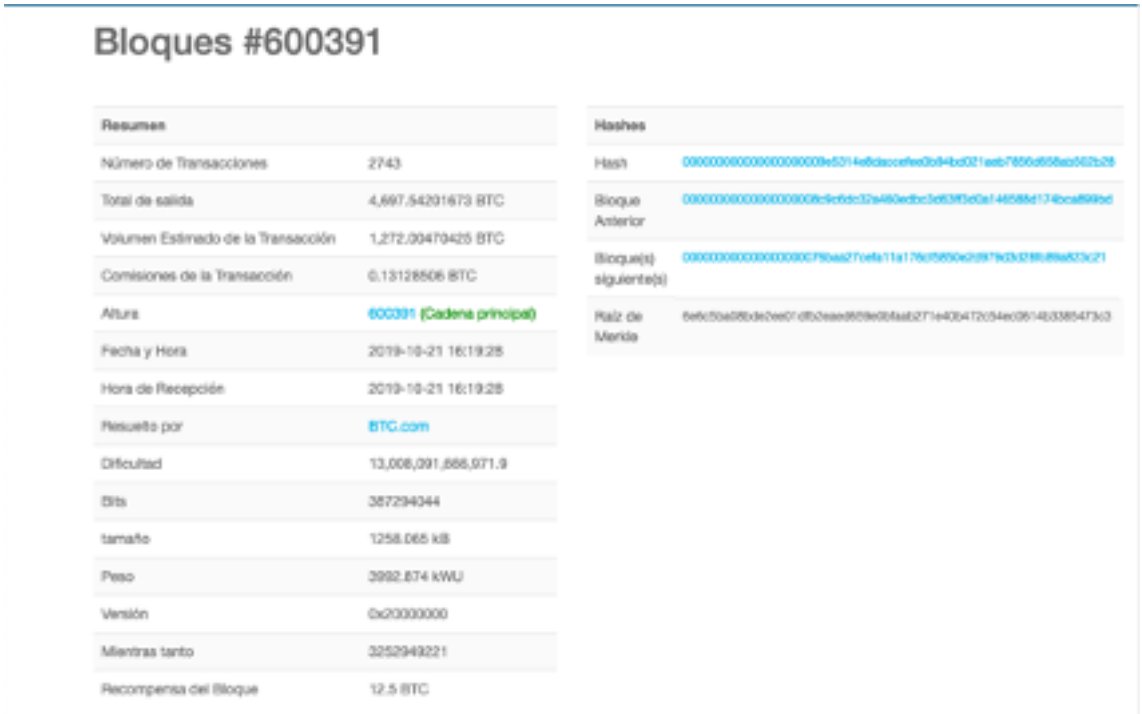

Figura 2.2: Extrato del contenido del bloque 600391 de la red Bitcoin.

2.1.6. Minado y Prueba de Trabajo

El minero es el encargado de hacer posible que los bloques se añaden a la cadena. Los mineros de una cadena de bloques competirán por ser los primeros en minar un bloque, para así poder recibir una recompensa que es fijada según la cadena de bloques donde estemos minando.

Minar es el acto de resolver una prueba de trabajo (del inglés *Proof of Work* (PoW)). A su vez una PoW es el resultado de resolver comúnmente un problema matemático.

Bitcoin en concreto usa el sistema *Hashcash* como prueba de esfuerzo. Dicha función de hashing debe de encontrar un caso donde se cumpla

$$H(s, x, c) < 2^{(n-k)}$$

Los parámetros son los siguientes:

- H: función de *hashing*.
- S: string del libro mayor de la cadena de bloques.
- X: número aleatorio.
- C: contador que el minero modifica.
- K: factor de trabajo o dificultad.
- N: tamaño de la salida del *hash*, por ejemplo, para SHA256; N=256.

Si se satisface la condición previamente descrita el bloque se añadirá a la cadena y los mineros empezarán a competir por ser los ganadores a la hora de minar otro bloque.

Un dato a tener en cuenta en la dificultad de minado K. A medida que se producen más transacciones la dificultad de minado se ajusta para evitar una saturación en la red. Es por ello que a mayor dificultad los mineros deberán de desempeñar un mayor gasto energético y por tanto de computación para poder resolver la PoW[Wik20a].

Como curiosidad, es poco probable, pero no imposible, que dos mineros sean capaces de minar el mismo bloque a la vez. De darse dicho problema, Bitcoin lo maneja de la siguiente forma:

Si un bloque A y un bloque B son minados a la vez, se producirá una rama de la cadena principal, conocido como *fork* generándose 2 ramas. El resto de mineros decidirán que cadena seguir y para cuando el próximo minero mine el siguiente bloque, dependiendo de la elección que haya tomado, se añadirá el bloque a la cadena que corresponda, dejando la otra cadena como una rama "huérfana", obligando al resto de mineros que tomaron el otro camino a volver al principal.

Es por esto por lo que **los bloques de una cadena no van ganando validez y van siendo firmes hasta que no se añaden una serie bloques tras ellos**. Los diferentes protocolos surgidos recientemente intentan evitar este problema de "validez", buscando bloques 100 % válidos desde su introducción a la cadena de bloques.

2.1.7. Proceso de Verificación

Tras haber visto los diferentes conceptos fundamentales, es importante juntar todo y entender como es el proceso completo desde que se generan las transacciones hasta que se añade de forma firme un bloque. La Figura 2.3 lo explica de forma clara.

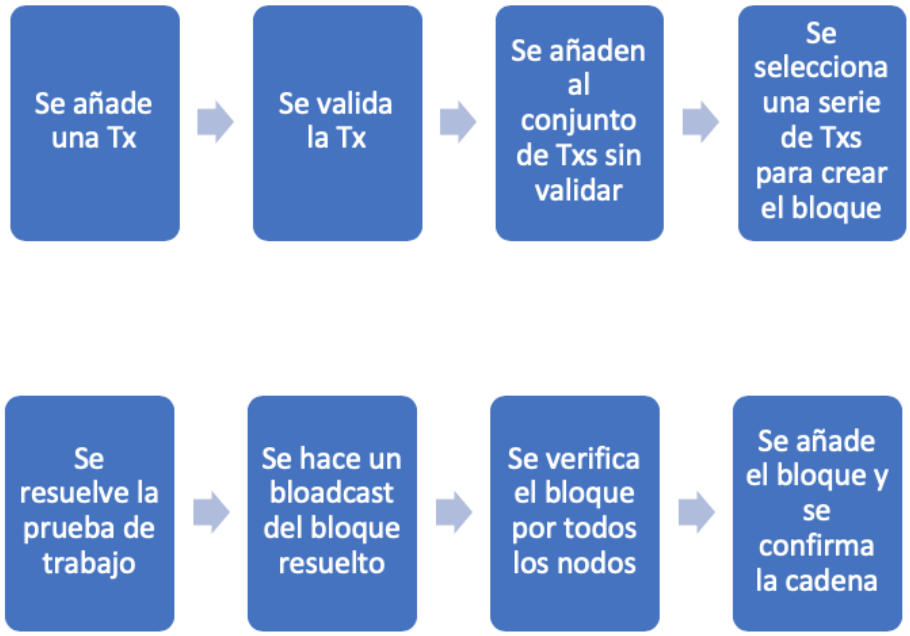

Figura 2.3: Proceso de verificación del protocolo Bitcoin.

2.2. Clases de Cadenas de Bloques

Previamente se ha hablado de Bitcoin ejemplificando el primer sistema de cadena de bloques con una utilidad real. En el caso de Bitcoin se trata de una cadena de bloques de carácter público, y a continuación se verá la diferencia entre esta y los diferentes tipos:

- **Públicas**: son todas las cadenas de bloques con acceso libre desde internet. La principal ventaja que tienen es que son muy seguras, con una gran trazabilidad y no inmutables. La principal desventaja es el tiempo que se tarda en validar un bloque y añadirlo a la cadena, es el precio a pagar de toda la seguridad.

- **Privadas**: las cadenas de bloques de carácter privado son impulsadas por empresas privadas, donde no tienen porque implementar una moneda como por ejemplo hace Bitcoin, los tiempos de minados son más cortos, y la seguridad puede ser menor, debido a que por el momento no están reguladas. Además, el acceso es restringido.

- **Híbridas**: la función principal es la de compartir las bondades de ambos tipos citados previamente. La idea principal es hacer una cadena de bloques principal de forma colaborativa entre diferentes nodos de instituciones que a priori pueden ser independientes. Las identidades comparten sus nodos, pero para poder acceder a dicha cadena de bloques y aportar un nodo, hay una autoridad que regula su acceso.

En el ámbito empresarial el uso más extendido está siendo tanto las cadenas de bloques híbridas de forma que la información final se guarda en una base de datos privada y en la cadena de bloques se aloja un *hash* para la validación de los datos.

2.3. Ethereum

Previamente se ha mencionado de forma leve a Ethereum.

Ethereum es otra cadena de bloques pública que empezó sus andanzas en 2013, cuando su creador Vitalik Buterin realizó la primera *Proof of Concept* (POC).

Su popularidad aumentó considerablemente gracias a la inclusión de los *Smart Contracts* (SCs), que se puede leer en la sección 2.3.2

2.3.1. Características de Ethereum

A parte de ser otra opción a Bitcoin, Ethereum implementó un concepto que ya existía previamente pero no había tenido una utilidad real, SCs, sección que se abordará posteriormente.

Además, Ethereum cuenta con peculiaridades en cuanto a estructura si la comparamos con Bitcoin:

1. Se añade el concepto de "account": una transacción actualiza directamente el balance de una cuenta.
2. Permite transmitir valores, mensajes y datos entre cuentas. Hay dos tipos de cuentas:
   a) *Externally Owned Account* (EOA): controladas por una clave privada.
   b) *Contract Account* (CA): Controladas por código.
3. Un siendo un contador para asegurarse que una transacción solo ha sido procesada una vez.

Por otro lado, las transacciones y los bloques también son diferentes si las comparamos con Bitcoin, una transacción posee:

- El beneficiario.
- Incluye una firma digital del emisor de la transferencia que sirve como autorización.
- La cantidad de *wei*, entendiendo por *wei* la unidad mínima en Ethereum. Un *wei* equivale a 10^{18} ethers.
- Un *gas* inicial, que es el máximo número de pasos computacionales que permite una transacción.
- El precio del *gas*, que es el precio o la tasa que hay pagar por el número de cálculos.

Es importante entender el concepto de *"gas"*. El gas fue introducido para limitar el tiempo efectivo que puede estar ejecutándose un contrato y por tanto la transacción donde esté ejecutado. Este atributo es muy importante porque sino **se podrían generar contratos recursivos interminables** que dejaran la red caída, con un gasto energético infinito.

Por otro lado, un bloque de Ethereum está compuesto por:

1. La cabecera del bloque (Hash).
2. Transacciones.
3. Detalles de un bloque.
4. Altura de la cadena.
5. Bloque previo (Hash).
6. Dificultad.
7. Dificultad total.
8. Tamaño.
9. Gas usado.
10. Limite del gas.
11. Se incluye el nonce previamente dicho.
12. La recompensa al minero que minó dicho bloque.

2.3.2. Contratos Inteligentes

La clave principal de Ethereum son los SCs. Están escritos en un lenguaje llamado Solidity [Fou20], el cual es una mezcla entre C++, Java y Javascript. Además, soporta estructura de datos y *Programación Orientada a Objetos* (POO).

Debemos entender los contratos inteligentes como fragmentos de código que en su interior contienen una lógica, y una vez son añadidos a la cadena de bloques, no hay forma de modificar su estructura o código.

Previamente se mencionaba la existencia de dos tipos de cuentas o roles en Ethereum y en concreto las CA.

Dichas cuentas (que son contratos en sí), proporcionan una dirección para que se puedan invocar sus funciones. Esta dirección se genera generando el *hash* de la cuenta del emisor y su *nonce*. Dicho contrato solo tendrá una única dirección.

Como puede verse en el Código 2.1 los contratos inteligentes están compuestos por diferentes elementos: variables de datos o estado, funciones (privada, pública, constructor, interna, view, fallback, external), estructurados definidos por el usuario, modificadores y eventos.

Código 2.1: Ejemplo de contrato inteligente

```
pragma solidity >=0.4.22 <0.6.0;

contract Ballot {

    struct Voter {
        uint weight;
        boole voted;
        uint8 vote;
        address delegate;
    }

    struct Proposal {
        uint voteCount;
    }

    address chairperson;
    mapping(address => Voter) voters;
    Proposal[] proposals;

    constructor(uint8 _numProposals) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
        proposals.length = _numProposals;
    }

    function giveRightToVote(address toVoter) public {
        if(msg.sender != chairperson || voters[toVoter].voted) return;
        voters[toVoter].weight = 1;
    }
}
```

2.3.3. Máquina Virtual de Ethereum

La máquina virtual de Ethereum EVM es el lugar donde se ejecutan los contratos inteligentes, en concreto el bytecode generado a partir de su compilación.

Debido a la naturaleza a la hora de compilar un contrato inteligente, la EVM se ve en la necesidad de usar una pila para pasar diferentes valores a las funciones de un contrato inteligente. En concreto, EVM usa una pila 256 bits, donde los primeros 16 elementos pueden ser accedidos a la vez. En total la pila almacena 1024 elementos.

Por otro lado, cuando un contrato a finalizado su ejecución, la memoria no es persistente y se borra. Es por ello que todo almacenamiento que quiera ser persistente no deberá de ser almacenado en la pila sino en el “contract storage”, que esencialmente actúa como base de datos pública, sin necesidad de enviar una transacción a un contrato, sin embargo, escribir en el storage es mucho más costoso en tiempo que escritura en memoria.

Capítulo 3

Seguridad

3.1. Seguridad en la Cadena de Bloques

Es importante diferenciar en que lugares se debe hacer hincapié cuando se habla de asegurar una solución de cadena de bloques. Hay múltiples puntos donde un atacante podría aprovecharse de una brecha de seguridad y robar, alterar o eliminar cualquier tipo de dato que pueda estar alojado, provocando que todo el sistema esté en riesgo. Una solución de cadena de bloques está basada en una autoridad descentralizada, lo que puede crear riesgos alrededor del control de la política de toma de decisiones.

3.1.1. Gobiernos y Empresas

- **Control de acceso**: La falta de una autoridad centralizada puede provocar la pérdida del control sobre aquellos que pueden acceder a la plataforma, dejando margen a personas que a priori no deberían poder acceder a hacerlo.

- **Económico**: Riesgo de fraude y pérdida de datos debido a fallos de seguridad que se verán a continuación.

- **Auditoría, legal y compañías de seguros**: Ciertas operaciones pueden requerir de acceso a ciertos tipos de datos que están alojados en la cadena de bloques, provocando riesgos a la hora de auditar y regular cierto tipo de casos que dependan de dicha información.

3.1.2. Procesos Basados en Arquitecturas de Cadena de Bloques

- **Identidad y administración del acceso**: Accesos no autorizados a la plataforma pueden tener consecuencias como pérdida de datos, operaciones abortadas y accesos denegados. Además, podría provocar la ejecución no deseada de determinados contratos inteligentes.

- **Comunicaciones seguras:** La mala comunicación entre los diferentes nodos de la red o entre componentes externos e internos podría dar como resultado un envío incorrecto de la información a agentes no deseados.

- **Código sin testear:** En el lado de los desarrolladores queda testar a fondo la implementación de los contratos inteligentes, de no darse esto, podrían producirse ejecuciones no deseadas de funciones, o aprovechamiento por parte de hackers de las vulnerabilidades para lucrarse. Más adelante se verá a fondo está sección.
- **Gestión de claves de identidad para recursos hardware compartidos:** Cuando se comparte un determinado hardware para alojar las claves de identidad de la cadena de bloques, un conjunto de claves de una determinada organización podría mezclarse con otro de otra organización.
- **Seguridad de una infraestructura:** Si desplegamos una cadena de bloques sobre una determinada infraestructura, deberemos de asegurarnos de aquellos procesos que corren por debajo en orden de evitar que estos intenten acceder a nuestra red.

3.1.3. Tecnologías

- **Almacenamiento, expiración y mal funcionamiento de las claves:** Un potencial riesgo es el manejo de tokens, expiración de certificados digitales y claves.
- **Seguridad en su aplicación:** Los fundamentos de la cadena de bloques hacen que por su naturaleza este sea inmutable y a prueba de suplantación, pero las aplicaciones que aprovechen la red de las cadena de bloques presentan retos en cada nivel.
- **Validación y autenticación del libro mayor:** Es fundamental entender el proceso de validación y autenticidad del libro mayor.
- **Riesgos en contratos inteligentes:** Cualquier fallo en su implementación puede provocar en una validación incorrecta de los contratos o transacciones.
- **Riesgo de borrado, auditoría y consenso:** Es fundamental determinar que protocolos seguir en caso de que una autoridad perteneciente a una red de la cadena de bloques abandone esta, proporcionando un consenso entre el resto de partes para minimizar los riesgos.

3.2. Seguridad en Ethereum y en Contratos Inteligentes

Ethereum por diseño dispone de los siguientes mecanismos para garantizar la seguridad de su funcionamiento:

1. **Hashing**: Se utilizan algoritmos tales como SHA-3, 256 y Keccak-256. Hay dos tipos de ”hashes.[en] Ethereum, simple y estructurado en forma de árbol. El simple agrupa los elementos que se quieran ”hashear.[en] una línea. En cambio, en forma de árbol, comúnmente conocido como Merkle Tree Hash, dado su coste ayuda a obtener un coste Log N en vez de N, ”hasheando”pares de datos. Los ”hashes”se usan en elementos tales como:
    - La dirección de la cuenta.
    - Firmas digitales.
    - Hash de una transacción (Tx).
    - Hash del estado de una transacción.

- Hash de la persona que recibe la transacción.
- Hash de la cabecera del bloque.

2. **Integridad de una transacción:** Para manejar la integridad de una transacción cada usuario debe de poseer una única cuenta además, debemos conocer la autoría de una transacción y debemos poder verificar que el contenido de una transacción no ha sido modificado. El proceso de generación de claves y dirección de wallet para poder verificar que una transacción no ha sido modificada, obtener la autoría de la transacción y garantizar el no repudio se explica en la Figura 3.1.

Figura 3.1: Proceso de generación de las claves de una cuenta

Una vez que la persona recibe la transacción podrá recalcular el *hash* de la información recibida mediante la clave pública para comprobar que no han existido modificaciones en los datos.

3. **Hilo de confianza:** La cadena de bloques ofrece un sistema de confianza basado en la seguridad, validación, verificación y comprobando que todos los recursos necesarios para una transacción están disponibles mediante el siguiente proceso:

   a) Validación de una transacción.

   b) Comprobación de los recursos: sintaxis, firma de una transacción, *timestamp*, nonce, limite del gas, validación de la cantidad de criptomonedas antes de la ejecución del contrato inteligente, firmas de las transacciones, hashes, etc.

   c) Se ejecutan las transacciones, se calcula el Merkle Tree Hash de las transacciones validadas, se calcula el Merkle Tree Hash de las transacciones validadas, todos los mineros ejecutan las transacciones del bloque junto con los smarts contrats y se calcula el Merkle Tree Hash de los estados y el estado ráiz de la cabecera del bloque.

3.2.1. Principios de Seguridad en Contratos Inteligentes

Previamente se hacia una breve introducción a los contratos inteligentes sin hacer énfasis en como están creados y cuales son sus propiedades y características más importantes. Desde el punto de vista de Ethereum, un contrato inteligente debe indicar siempre la versión de Solidity que usa. Además, dado las características de Solidity sique el patrón de diseño orientado a objetos a demás del uso de clases y métodos. Aquellas funciones y atributos más importantes se deben de tener en cuenta al hablar de seguridad son:

- **Constructora:** Es una función especial que se corre en el momento de creación del contrato y que no puede ser llamada después de dicha acción.

- **Funciones del tipo *address*:** en Ethereum hay dos tipos de datos para representar una dirección, a saber:

  - **Address:** almacena 20 bytes, que es la dirección de una dirección de Ethereum.
  - **Address payable:** igual que address, pero además cuenta con las variables transfer y send.

  La diferencia entre ambos tipos de variables es que payable es una dirección a la cual puedes enviar Ethers, mientras que address a secas no [Bit19]. Además dentro de las dirección se encuentran diferentes llamadas las cuales son de especial importancia para entender las vulnerabilidades:

  - **Call:** llamada que no altera el estado de un contrato. Además, no gasta ningún tipo de gas. Esta llamada transfiere una cantidad weis a la dirección a la cual se está llamando.

    ```
    c.call.value(amount)(...);
    ```

  - **Send:** tiene la misma funcionalidad que call, con la diferencia de que hay un gasto de gas y sirve para alterar el estado de u contrato. Además, cuando se finaliza la ejecución, se llama a la función fallback correspondiente.

    ```
    c.send(amount);
    ```

  - **Fallback:** tipo de función especial que se implementa en todos los contratos. Una vez que se finaliza la ejecución de determinadas funciones se realiza una llamada a dicha función, la cual, no tiene nombre. Puede verse en el Código 3.1

Código 3.1: Contrato inteligente de ejemplo

```
1  Contract Example {
2
3      string private name = "Alberto";
4
5      function foo() public view returns (string)
6      {
7          return name;
8      }
9
10     function() public
11     {
12         //Funcion fallback
13     }
14
15 }
```

3.2.2. Incidentes de Seguridad en Ethereum

Tal y como se relata en los principales estudios en los cuales está basada la aplicación, la principal motivación que hay para la creación de dicha aplicación es el principal problema que existe para crear una manera eficiente y concreta de realizar testing y descubrir la mayor cantidad de vulnerabilidades que podemos encontrar en los contratos inteligentes. Ethereum es uno de los mayores principales frameworks de contratos inteligentes, y junto con ello, se han descubierto graves brechas de seguridad mediante diferentes técnicas

analíticas (análisis estático, dinámico y formal).
Debido a estos fallos se han producido graves incidentes de seguridad, con pérdidas económicas muy altas, basadas en errores de programación y de uso de funciones y llamadas en Solidity. Los principales incidentes han sido:

- **El Ataque DAO (del inglés *Decentralized Autonomous Organization* (DAO)):** con el fueron robados más de 3,6 millones de Ethers en junio de 2016. Poniendo en tela de juicio el sistema DAO, modelo en el cual están basados los sistemas de cadena de bloques públicas. Dicho robo se produjo aprovechando una vulnerabilidad llamada "re-entracy problem". Finalmente la comunidad de Ethereum tuvo que hacer un "fork" de la cadena principal para revertir el robo, a partir del cual empezaron a convivir Ethereum Classic como la cadena antigua, y Ethereum como la cadena principal (estando ambas activas a día de hoy).
- **El ataque a las carteras multi-firma "Parity":** dicho ataque se basó en el mal control del acceso a las funciones de una librería, la cual se encargaba de centralizar el uso de diferentes cuentas que podía tener un mismo usuario para poder realizar transacciones sin una dependencia tan clara de las claves públicas de cada librería. El robo total fue de alrededor de 151 carteras digitales, alcanzando 15 millones de Ethers en total.

3.2.3. Vulnerabilidades relacionadas con Solidity

- **Ataque de reentrada:** Aprovechándose de la estructura de Solidity, dicho ataque aprovecha la llamada a la función **fallback**, la cual está presente en todos los contratos inteligentes. Dicha llamada, explicada en la Sección 3.2.1, provoca que un atacante pueda invocar una llamada de un contrato externo. Además, no lanza ninguna excepción ni errores, y en caso de no haber puesto un limite de gas, se ejecuta sin límite. En los Códigos 3.2 y 3.3 puede verse un caso simplificado de este ataque.

Código 3.2: Ejemplo del ataque reentrancy 1/2

```
1  contract Victim {
2
3      boole sent = false;
4      Victim public me = Victim(...);
5
6      function withdraw(address c) public
7      {
8          if(!sent) {
9
10             c.call.value(2)();
11
12             sent = true;
13
14         }
15     }
16 }
```

Código 3.3: Ejemplo del ataque reentrancy 2/2

```
contract Reentrancy_Attack {
    ...

    function() public {

        Victim(msg.sender).withdraw(this);

    }
}
```

En primer lugar, el atacante hace una llamada a la función withdraw de la victima, dicha cantidad envía 2 weis en caso de que no lo haya enviado previamente. Debido a la llamada a la función callback (que se ejecuta automáticamente tras llamar a withdraw), se vuelve a llamar a withdraw. Esto provoca que se haya una llamada recursiva constantemente. En segundo lugar, hay que fijarse en el flag sent, que por un mal uso del programador, no modifica la variable. El lugar adecuado sería modificarlo delante del envío de weis. Por último, dado que call no propaga excepciones, hasta que no se agote el gas de dicho contrato, la ejecución no parará.

- **Excepciones no controladas:** Solidity posee diferentes vulnerabilidades motivadas por las llamadas a funciones externas que no pertenecen al contrato o simplemente propias. Si las excepciones no son tratadas debidamente pueden provocar brechas graves de seguridad. Una excepción puede ser lanzada por los siguientes motivos:

  1 La ejecución se queda sin gas: Una ejecución se queda sin gas cuando se usa una función send para transferir una cantidad de Ether a un contratos.

  2 La pila de llamadas enlazadas alcanza el límite.

  3 El comando *throw* es ejecutado.

  En el caso de que haya llamadas enlazadas si cada elemento de la cadena es una llamada directa la excepción se parará y se revertirá el efecto. En cambio, si hay llamadas enlazadas (usando call o delegatecall), la excepción se propaga revertiendo los efectos hasta que se alcanza la variable que lo haya llamado (call o delegatecall).

Código 3.4: Ejemplo de código Solidity seguro

```
contract Alberto { function feedMe(uint) return (uint)}
contract Profesor {

    bool exceptionStopped = false;

    function exeFeed(Alberto a) {
        x= true;
        a.feedMe(10);
        x = false;
    }
}
```

En el Código 3.4, si el profesor decide ejecutar una función del alumno Alberto que espera consumir un int para devolverlo y dicha función lanza una excepción, se propagará por completo y todas las acciones serán revertidas, quedando la variable *exceptionStopped = false*.

En cambio, en el Código 3.5 de darse una llamada a la función feedMe mediante la variable call, en el momento, en el que se alcanzara la variable sería *exceptionStopped = true*.

Código 3.5: Ejemplo de código Solidity inseguro

```
1  contract Alberto { function feedMe(uint) return (uint)}
2  contract Profesor {
3
4      bool exceptionStopped = false;
5
6      function exeFeed(address a) {
7          x= true;
8          a.methods.feedMe.call(10);
9          x = false;
10     }
11 }
```

Por lo tanto, la solución pasa por controlar las excepciones que se lanzan y restringir al máximo el uso de call para llamar a funciones.

- **Agotamiento del gas:** Debido a las características de Solidity, cuando un contrato envía cierta cantidad de Ethers a otro contrato, puede repercutir en fallbacks que ayuden a aumentar la cantidad de gas que una ejecución necesita.
  Desde el punto del vista del programador, debe de controlarse dicha acción y los patrones de desarrollo que se usan en orden de mejorar la eficiencia del uso del gas

- **Casting de tipos:** El compilador de Ethereum como la gran mayoría de compiladores checkea los tipos de las variables para detectar errores. Hay que tener en cuenta los siguientes fallos dados a la hora de ejecutar cierto tipo de llamadas:

  ```
  contract Alberto { function consume(uint) return (uint)}
  contract Profesor { function callStudent(Alberto a){ a.consume(10); }}
  ```

  1. Si a no fuera una dirección la llamada no ejecutaría nada.
  2. Si a es la dirección (como se presupone), se ejecuta la llamada.
  3. Si a es la dirección y contrato, pero la función no existe, la llamada fallback de a será ejecutada.

  Como podemos comprobar, el punto último es muy importante, puesto que en caso de no existir una llamada y de darse un mal patrón de diseño en la fallback, podría haber un comportamiento no deseado.
  Además, previamente a la versión 0.4.0 de Solidity, no se lanza ningún tipo de excepción en caso de hacer una llamada a una función que no existe.

3.2.4. Vulnerabilidades Inherentes a la Cadena de Bloques

- **Dependencia en un orden de transacción:** Dadas las características asíncronas en la ejecución de los contratos inteligentes, y por ende de las transacciones, se puede dar el caso de que en un mismo bloque existan 2 transacciones que modifiquen el estado de un mismo contrato inteligente. Un usuario no puede predecir cual será el estado de un contrato en tiempo de ejecución.
  Este hecho puede provocar que dependiendo de cual sea la transacción la cual se ejecute primero, condicione el resto de la cadena.

- **Dependencia de los timestamps:** Cuando un bloque se mina el minero debe de generar una marca temporal, en orden de "sellar.[en] que momento se produjo el minado. La responsabilidad recae sobre el minero, dándose la posibilidad de que un minero malicioso pueda alterar en su beneficio personal dicha marca temporal con el fin de obtener algún beneficio sobre los contratos inteligentes que dependan de dicho *timestamp*.

- **Ejecución temporal de los smartcontracts:** La vulnerabilidad radica en la estructura de los sistemas de cadena de bloques. El orden de ejecución de los contratos inteligentes está determinado por el protocolo de consenso de Ethereum. Por tanto, todos los contratos se ejecutan a la vez en todos los nodos de la cadena. Si un usuario introdujera un contrato de largo tiempo de ejecución, provocaría una bajada de rendimiento en la red.

- **Generación de números aleatorios:** Un minero que controle una gran cantidad de bloques y que además debe generar el hash para conseguir que un nuevo bloque que está minando se añada a la cadena, podría alterar o buscar un determinado hash en un bloque, para tener o prever hashes en un futuro que le ayuden a prever el comportamiento de la cadena de bloques y por tanto el comportamiento de la red. Una soluxión muy extendida es el uso de protocolos de compromiso. Cada participante (minero), elige un "secreto" y posteriormente comunica una parte de este al resto de participantes pagando una fianza. Al final, todos los participantes revelan sus secretos y se calcula un número pseudo-aleatorio. En caso de que un participante no revele el secreto, perderá la fianza.

3.2.5. Vulnerabilidades que Afectan a la EVM

- **Límite del tamaño de la pila de llamadas recursivas:** En versiones antiguas de contratos desplegados en Ethereum se presentaba un problema relacionado con las llamadas enlazadas. Por defecto, Ethereum poseía como máximo una profundidad de 1024 llamadas, es decir cada vez que un contrato llamaba a otro o así mismo, el contador aumentaba.En caso de ser excedida, la EVM lanzaría un error. Dicho error permitía ser explotado hasta Octubre de 2016 de una forma sencilla. El atacante provocaba la llamada a sí mismo hasta agotar prácticamente la totalidad de las llamadas, a continuación, llamaba al contrato de la víctima, de tal forma que si este no tenía un tratamiento adecuado de excepciones, podría incurrir en pérdida de información o *Ether, la criptomoneda de Ethereum* (ETH). Dicho fallo fue corregido limitando la cantidad de gas que un emisor de una llamada puede consumir a 63/64 del gas, resultado en que el número máximo de llamadas que se puede hacer siempre es menor de 1024.

- **Persistencia de los contratos:** La principal ventaja que tiene la cadena de bloques es el control férreo que hay. La persistencia de la cadena de bloques provee una seguridad muy alta frente a fraudes pero a la hora de desplegar contratos inteligentes sufre un problema, los contratos no se pueden deshacer. Desde el momento en el que un contrato es desplegado, no hay forma de modificar su comportamiento, en caso de que contenga un fallo, dicho fallo se quedará de forma permanente hasta que se ejecute la función para su borrado y posterior mente despliegue del correcto. Por otro lado, si un atacante anónimo pudiera llamar a la función suicide de una víctima o contrato podría eliminar de Ethereum dicho contrato. El acceso a la función de

destrucción de un contrato debe de estar siempre restringido al propietario de dicho contrato.

3.3. Herramientas de Análisis de Vulnerabilidades en Ethereum

A la hora de estudiar las diferentes herramientas disponibles para el análisis de contratos inteligentes se ha usado una máquina virtual con sistema operativo Ubuntu 18.04 LTS. Se han encontrado cierto tipo de limitaciones en el uso de características que necesitan una interfaz visual. La motivación principal de tener todas las herramientas disponibles en único con entorno consiste en facilitar al usuario las tareas de obtención de todas las herramientas por separado, instalación de cada una de ellas y aprendizaje de las mismas, si los problemas que suelen ocurrir en los procesos de instalación.

3.3.1. Métodos de Análisis

El primer lugar, es importante recordar que hay tres tipos de análisis que se pueden realizar:

- **Análisis estático**: Es el más común. Comprueba qué vulnerabilidades que se producen en el código fuente traducido a *bytecode* sin ejecutarlo.

  Algunas de las técnicas más conocidas son: control flujo de datos, flujo gráfico, análisis léxico, compilar y descompilar y reconocimiento de patrones.

- **Análisis dinámico**: Análisis en tiempo de ejecución en un único contexto. Algunos de los tipos son: análisis en tiempo de ejecución, construcción de grafos de transacciones, análisis simbólico, validación.

- **Verificación formal**: Ese tipo de análisis usa métodos formales matemáticos para refutar condiciones predefinidas determinado si se cumplen o no. Ejemplos de este tipo son: creadores de programas lógicos, probadores teóricos, y traductores a lenguaje formal.

En segundo lugar, las múltiples herramientas poseen diferentes propósitos dependiendo si trabajan con archivos de Solidity, bytecode, desensamblado, decompilado, etc. Es importante saber:

- **Desensamblador**: Es una herramienta de software que transforma el código máquina en un código legible por un ser humano llamado lenguaje montado o expresado en inglés *assembly code*.

- **Descompilador**: Usado para revertir el proceso de compilación. El descompilador coge los archivos binarios como entrada y devuelve un lenguaje estructurado de alto nivel. Principalmente descompila usando archivos binarios.

En tercer lugar, un contrato puede ser analizado de múltiples formas. Los dos estados más vistos son en código de alto nivel, es decir, el código tal cual escrito por el desarrollador y el *bytecode* tras haberlo compilado. En ciertos casos, las herramientas también son

capaces de trabajar con el código producido por la máquina virtual de Ethereum, pero no suele ser lo común.

En cuarto lugar, es importante darse cuenta de la complejidad de analizar un contrato inteligente dado que puede tener diferentes versiones. El compilador de Solidity, llamado, necesita saber qué versión tiene cada contrato analizado. Afortunadamente existen dos herramientas que se encargan de solucionar este problema como son *solc-select* y *py-solc*.

- **solc-select y py-solc**: Uno de los problemas generalizados encontrados a la hora de instalar y usar las herramientas de análisis ha sido lidiar con la versión del código y el propio compilador de Solidity. Existen múltiples versiones[**?**] del código de Solidity. Para poder descargarlo, se debe descargar el compilador propio que ofrece la herramienta[Eth20a]. El compilador depende enormemente de la versión del código que tenga que compilar provocando incompatibilidades si el compilador no acepta la versión de código que vaya a compilar. Aquí radica la capacidad de poder cambiar entre diferentes versiones del compilador y es donde aparecen **Solc-Select** y **py-solc**.
- **Solc-select:** Es un script[Cry19] que permite de forma sencilla intercambiar entre versiones del compilador de Solidity. La herramienta descarga desde el repositorio de Github las diferentes versiones del compilador, para que el usuario en cuestión pueda cambiar de versión cuando lo desee mediante el comando

  ```
  $ solc use <put Solidity version here>
  ```

  El script tiene ciertas limitaciones como **las versiones que puede instalar**, estando el rango desde la versión 4.11 a la 5.5. Además, la propia herramienta te advierte de que las versiones nuevas del compilador no podrán compilar versiones antiguas de Solidity.
- **py-solc**: Es paquete pip3 de Python que realiza la misma función que Solc-Select, siendo py-solc[Eth18] más limitada al dar solo la posibilidad de cambiar entre la versión 4.1 a la 4.25.

3.3.2. Slither

Slither[Pac19] es un framework de análisis estático escrito en Python 3, con dependencias con el compilador de Solidity. La herramienta provee una serie de detectores para diferentes tipos de vulnerabilidades, en concreto 45. Algunas de las funcionalidades de la herramienta son:

- Detección de falsos positivos.
- Localización exacta de los errores en código.
- Capacidad de analizar contratos desde la versión 0.4.
- Cubre el 99,9 % del código de Solidity.
- La media de ejecución de de menos de un segundo por contrato.

La totalidad de las vulnerabilidades con una pequeña descripción de la vulnerabilidad puede encontrarse en la documentación propia[JF17b].

En la Tabla 3.2 pueden verse los detectores más importantes de la herramienta Slither:

Tabla 3.1: Los ocho detectores más importantes de Slither

Detector	**Vulnerabilidad**
name-reused	Nombre del contrato. reutilizado
rtlo	Uso de la vulnerabilidad ”sobre escritura del carácter de izquierda a derecha”.
shadowing-state	Variables de estado públicas.
suicidal	Funciones que permiten destruir el contrato.
uninitialized-state	Varibales de estado sin inicializar.
uninitialized-storage	Variables de almacenamiento sin inicializar.
arbitrary-send	Funciones que envian ether a destinos arbitriarios.
controlled-delegatecall	Control del destino de la ”llamada delegada”.

3.3.3. Oyente

Oyente es una herramienta[LL18] compleja de instalar. Las depencencias de software para poder usarla son las siguientes: Python 2 y Python 3, compilador de Solidity, la máquina virtual de Ethereum (EVM), z3 Theorem Prover v4.5.0, web3 y los requisitos dentro del archivo *requests*.

La herramienta provee bastante documentación que puede consultarse en [JF17a] Oyente es una herramienta de análisis simbólico que trabaja directamente con el *bytecode* de un contrato. En caso de que se quiera analizar un contrato sin compilar, la herramienta también lo admite puesto que compila los contratos inteligentes.

Oyente principalmente trata las siguientes vulnerabilidades[DAS19]:

- Integer Under-Over-flow
- Parity Multisig Bug
- Ataque por pila de llamadas (Parity Multisign Attack)
- Dependencia en el orden de transacciones (Transaction-Ordering Dependence)
- Dependencia en el estampado temporal (Timestamp Dependency)

- Re-entrancy problem.

El fallo principal que tiene Oyente es la **instalación**. Es excesivamente compleja si se quiere hacer desde cero, obligando muchas veces a hacer uso del Docker proporcionado.

Además, hace uso de una versión del compilador antigua, la 0.4.21, lo cual puede provocar en el corto plazo que deje de tener soporte para contratos más modernos.

Su punto fuerte es la **simplicidad en la respuesta**, diciéndonos si ha encontrado fallo o no, aunque si bien es cierto podría proveer de más información (linea del fallo, solución...)

3.3.4. Remix - IDE

El entorno de desarrollo[rem20] permite al usuario desarrollar contratos inteligentes de una forma muy efectiva. Las características principales que ofrece son:

1. Es un editor online que permite escribir contratos inteligentes en Solidity. Además, permite escoger diferentes versiones del lenguaje, compilador, direcciones simuladas de contratos, etc.
2. Permite seleccionar la versión de la máquina virtual de Ethereum EVM.
3. Es una analizador de código estático.
4. Permite la instalación de plugins para mejorar el proceso de desarrollo.

Remix-IDE es un excelente entorno para un usuario de nivel bajo, medio y alto, puesto que permite de forma sencilla sin un entorno local de desarrollo poder desplegar contratos en un nodo virtual de la máquina virtual de Ethereum y además analizar su seguridad con los diferentes módulos.

3.3.5. Solgraph

Cuando se hable de herramientas de seguridad hay que hacer referencia a aquellas que generan grafos y diagramas de control de flujo. Solgraph[Rev19] es una de ellas encargándose de general un archivo con extensión *DOT*[Wik20b].

La extensión DOT es un **lenguaje de descripción de grafos**, es importante no confundirlo con la extension Microsoft Word Document (DOT). Cuando la herramienta analiza un contrato inteligente, lee un archivo de entrada, lo procesa y expulsa un archivo que contiene un grafo con el diagrama de control de flujo del contrato analizado.

Para representar las diferentes acciones de esos diagramas se usan una serie de colores:

1. Rojo: enviar ether a una dirección externa.
2. Azul: llamada a una función constante (constant function).
3. Amarillo: llamada a una función vista (view function).
4. Verde: llamada a una función pure.

5. Naranja: uso del método *call*.
6. Morado: uso del método *tranfer*.
7. Lila: uso del método *payable*.

Tras generar el archivo .dot, es necesario transformar dicho archivo al formato .png.

Solgraph es una herramienta muy sencilla y simple. Tiene un función muy clara y limitada, que solventa de forma sencilla. Si bien es cierto que para detalles de bajo nivel no sirve, da información útil de fujos y posibles fallos en algo nivel.

3.3.6. Manticore

Manticore es una herramienta simbólica. Este tipo de análisis se centra en conocer todos los distintos *çaminos"* que puede encontrar la herramienta en un análisis concreto. Este tipo de análisis tiene algunas limitaciones, la más clara es el problema de obtener vulnerabilidades en contratos de una larga extensión. La herramienta admite dos tipos de entrada, *bytecode* y archivos binarios de tipo Linux (en inglés *Executable and Linkable Format* (ELF)).

3.3.7. Smartcheck

Es una herramienta de análisis estático desarrollada de forma modular que permite hacerla extensible. Smartcheck ofrece la detección de varias vulnerabilidades de forma eficiente mostrando **dónde puede estar el fallo en función de unas reglas predefinidas**. Además, puede ser instalada como paquete npm, o con Java y posee una versión web[Sma18b]. La información que provee Smartcheck es clara, sencilla y útil. Además, si un usuario quiere introducir de forma propia nuevas vulnerabilidades, o incluso patrones que desee encontrar lo podrá hacer, aumentando la capacidad y el futuro de la herramienta.

3.3.8. Mythril

Los creadores la definen como una herramienta de análisis de seguridad centrada en el bytecode, permitiendo el análisis mediante una dirección o pasando un contrato inteligente de forma local.

Los parámetros de entrada de Mythril son:

- Tiempo de espera: –execution-timeout.
- Profundidad (recursividad): –max-depth.
- Número máximo de transacciones a incluir en el análisis.
- Formato de análisis de salida: -o.
- Especificar la versión del compilador Solidity: –solv.

Uno de los detalles más importantes de la herramienta es que se aconseja poner la configuración con un temporizador. Durante la ejecución de prueba sin el flag de espera,

la herramienta funcionó durante más de 5 minutos. Teniendo en cuenta que un contrato puede ser largo, y cuanto más largo mayor dificultad, es muy recomendable utilizar dicho flag.

Las capacidades que tiene son muy grandes aunque posee cierto tipo de problemas si el contrato posee versiones del compilador que no soporta. Para análisis masivos puede que no sea la herramienta ideal al trabajar con recursividad y profundidad lo que la hace bastante lenta. Aún así es un gran recurso como último filtro de seguridad antes de un despliegue a la blockchain oficial de Ethereum. En la Figura 3.2 pueden verse los resultados de una ejecución del herramienta.

```
ubuntu@smartcontracts:~/ALBERTO/oyente/tests$ myth analyze reentrancy-0.5.1.sol --solv 0.5.1
solc, the solidity compiler commandline interface
Version: 0.5.1+commit.c8a2cb62.Linux.g++
^C ==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Medium
Contract: Reentrancy
Function name: withdrawBalance_fixed()
PC address: 507
Estimated Gas Usage: 6583 - 61244
A call to a user-supplied address is executed.
The callee address of an external message call can be set by the caller. Note that the callee can contain arbitrary
code and may re-enter any function in this contract. Review the business logic carefully to prevent averse effects o
n the contract state.
--------------------
In file: reentrancy-0.5.1.sol:29

msg.sender.call.value(amount)("")

--------------------
Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x421c10c05420fffff, nonce:0, storage:{}
Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], data: [CONTRACT_CREATION], value: 0x0
Caller: [ATTACKER], function: withdrawBalance_fixed_3(), txdata: 0x31fbf9ce, value: 0x0
Caller: [ATTACKER], function: withdrawBalance_fixed(), txdata: 0x1b0ca26a, value: 0x0

==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Medium
Contract: Reentrancy
Function name: withdrawBalance_fixed_3()
PC address: 767
Estimated Gas Usage: 6605 - 61266
A call to a user-supplied address is executed.
The callee address of an external message call can be set by the caller. Note that the callee can contain arbitrary
code and may re-enter any function in this contract. Review the business logic carefully to prevent averse effects o
n the contract state.
--------------------
In file: reentrancy-0.5.1.sol:49

msg.sender.call.value(amount)("")

--------------------
Initial State:

Account: [CREATOR], balance: 0x1, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x421c10c05420fffff, nonce:0, storage:{}
Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}
```

Figura 3.2: Parte de los resultados de un análisis con Mythril.

3.3.9. SolMet

SolMet permite realizar análisis estáticos de las métricas que posee un contrato escrito en Solidity. Algunas de las métricas son:

1. SLOC - número de líneas de código fuente.

2. LLOC - número de líneas de código sin espacios y comentarios.

3. CLOC - numero de líneas comentadas.
4. NF - numero de funciones.
5. McCC - complejidad ciclomática de McCabe.
6. WMC- suma del peso de las distintas complejidades de las funciones de un contrato.
7. NL - el nivel más baje de subllamadas dentro de las funciones.
8. NLE - nivel de .ªnidamiento"de sentencias if-else
9. NUMPAR - número de parámetros.
10. NOS - número de sentencias.
11. DIT - profundidad del arbol de herencia.
12. NOA - número de ancestros.
13. NOD - número de descendientes.
14. CBO - acoplamiento entre los objetos del contrato.
15. NA - número de atributos.
16. NOI - número de invocaciones.

Toda esta información se puede encontrar en el propio repositorio.[Sol19] Uno de los problemas al usar la máquina virtual de Ubuntu es abrir de forma clara archivos *Comma Separated Values* (CSV). Para ello hubo que traerlo a una máquina que permitiera visualizar dichos archivos. Después de probar la herramienta con el contrato *uninitialized.sol* los resultados en la línea de comandos fueron los que se ven en la Figura 3.3 y en la Figura 3.4

```
ubuntu@smartcontracts:~/ALBERTO/SolMet-Solidity-parser/target$ java -jar SolMet-1.0-SNAPSHOT.jar -inputFile ../..
/oyente/tests/uninitialized.sol -outFile test1_result.csv
Parsing ../../oyente/tests/uninitialized.sol
ubuntu@smartcontracts:~/ALBERTO/SolMet-Solidity-parser/target$ ls
classes            maven-archiver  original-SolMet-1.0-SNAPSHOT.jar  test1_result.csv
generated-sources  maven-status    SolMet-1.0-SNAPSHOT.jar
ubuntu@smartcontracts:~/ALBERTO/SolMet-Solidity-parser/target$ cat test1_result.csv
SolidityFile;ETHAddress;ContractName;Type;SLOC;LLOC;CLOC;NF;WMC;NL;NLE;NUMPAR;NOS;DIT;NOA;NOD;CBO;NA;NOI;Avg. McC
C;Avg. NL;Avg. NLE;Avg. NUMPAR;Avg. NOS;Avg. NOI
uninitialized.sol;tests;Uninitialized;contract;9;9;0;1;1;0;0;0;1;0;0;0;0;1;1;1.0;0.0;0.0;0.0;1.0;1.0
uninitialized.sol;tests;Test;contract;14;13;1;2;2;0;0;0;2;0;0;0;0;2;1;1.0;0.0;0.0;0.0;1.0;0.5
uninitialized.sol;tests;Lib;library;11;10;2;1;1;0;0;0;2;1;0;0;0;0;0;0;1.0;0.0;0.0;2.0;1.0;0.0
uninitialized.sol;tests;Test2;contract;17;0;0;2;2;0;0;0;2;0;0;0;0;1;3;2;1.0;0.0;0.0;0.0;1.0;1.0
```

Figura 3.3: Ejemplo de análisis con SolMet

SolidityFile	ETHAddress	ContractName	Type	SLOC	LLOC	CLOC	NF	WMC	NL	NLE	NUMPAR	NOS
uninitialized.sol	tests	Lib	library	11	10	2	1	1	0	0	2	1
uninitialized.sol	tests	Test2	contract	17	0	0	2	2	0	0	0	2
uninitialized.sol	tests	Test	contract	14	13	1	2	2	0	0	0	2
uninitialized.sol	tests	Uninitialized	contract	9	9	0	1	1	0	0	0	1

DIT	NOA	NOD	CBO	NA	NOI	Avg. McCC	Avg. NL	Avg. NLE	Avg. NUMPA	Avg. NOS	Avg. NOI
0	0	0	0	0	0	1.0	0.0	0.0	2.0	1.0	0.0
0	0	0	1	3	2	1.0	0.0	0.0	0.0	1.0	1.0
0	0	0	0	2	1	1.0	0.0	0.0	0.0	1.0	0.5
0	0	0	0	1	1	1.0	0.0	0.0	0.0	1.0	1.0

Figura 3.4: Resultados visualizados en excel

Por lo tanto, SolMet ofrece una forma sencilla de obtener una serie de métricas muy claras para determinar que tipos de contratos se despliegan en la cadena de bloques. Dichas métricas ofrecen una compresión siempre y cuando se puedan comparar unos con otros.

3.3.10. Vandal

El Vandal es una herramienta muy potente que se diferencia de las previamente mencionadas. Posee diferentes módulos para desensamblar, descompilar y analizar las bibliotecas usando Datalogs. Todos trabajan con *bytecode*. Hay mucha información sobre la herramienta en la propia enciclopedia del github[van19]. Los puntos claves son:

- Desensamblador (Python): proporciona una funcionalidad extra en comparación con el Ethereum uno.
- Descompilador: los desarrolladores de Vandal tratan a este descompilador como el núcleo. Permite convertir el código de bytes EVM en una representación intermedia, que posteriormente, permite la creación de un diagrama de flujo de control (tal y como hace el programa Solgraph).
- Registro de datos: Vandal utiliza un motor llamado Souffle Datalog, detectando la lógica y los diferentes códigos de operación de Solidity para determinar si existe una vulnerabilidad o no.

Al principio se decidió ejecutar sólo el script masivo ya que ofrecía una forma muy rápida de hacer un test sólido. Sin embargo, se encontró que en 2 sistemas diferentes (Mac Os X y Ubuntu) el script falla. Ante esto, la manera más sencilla de probar la herramienta puede verse en la Figura 3.5

```
MacBook-Pro-de-Alberto:vandal Ture$ bin/disassemble -p examples/dao_hack.hex
0x00 PUSH1  0x60
0x02 PUSH1  0x40
0x04 MSTORE
0x05 CALLDATASIZE
0x06 ISZERO
0x07 PUSH2  0xb9
0x0a JUMPI

0x0b PUSH1  0xe0
0x0d PUSH1  0x02
0x0f EXP
0x10 PUSH1  0x00
0x12 CALLDATALOAD
0x13 DIV
0x14 PUSH4  0x13af4035
0x19 DUP2
0x1a EQ
0x1b PUSH2  0x19e
0x1e JUMPI

0x1f DUP1
0x20 PUSH4  0x26f5a8c9
0x25 EQ
0x26 PUSH2  0x1c1
0x29 JUMPI
```

Figura 3.5: Ejecución de desasemblado para el archivo daohack.hex.

3.3.11. EthIr

EthIR es una evolución de Oyente. Los comandos básicos son bastante similares a la herramienta de Oyente (porque la amplía). Dando un valor extra a Oyente generando una representación basada en reglas de programa (en inglés *Rule Based Representation* (RBR)).

El principal motivo para crear estas reglas es poder inferir en el flujo lógico del contrato mediante la inclusión de nuevas ordenes. Esto se hace gracias al lenguaje de nivel intermedio utilizado a la hora de analizar el contrato. Recordando los comandos de Oyente, los comandos básicos de Ethir son similares:

```
Contrato local:        ./oyente-ethir.py -s <contract filename>
Contrato \textit{bytecode}:     ./oyente-ethir.py -s <contract filename> -b
Contrado desasemblado: ./oyente-ethir -s <contract filename>.disasm -disasm
```

Después de ejecutar el script para generar el archivo RBR hay que hacer una interpretación del archivo de salida. Toda esa información puede ser encontrada en el propio repositorio de la herramienta.

Como conclusión, el RBR produce un archivo para después analizado y explotado con otras herramientas, lo que provoca que por si solo el archivo RBR no proporcione información útil inmediata. En la Figura 3.6 pueden verse las opciones de ejecución de la herramienta.

3.3.12. Maian

Maian es una herramienta realmente fácil de usar siendo un analizador estático de 3 tipos de categorías principales de contratos inteligentes:

- Prodigal: contratos que permiten fugas de activos de forma arbitraria.
- Suicidal: Estos contratos pueden ser eliminados por cualquier usuario.
- Greedy: Estos contratos bloquean los activos indefinidamente.

Las opciones por linea de comandos son: Seleccionar el tipo de entrada

- Sol:

```
$ python maian.py -s <contrato> <nombre del contrato maian>
```

- *Bytecode* fuente:

```
$ python maian.py -bs <contrato en bytecode fuente>
```

- *Bytecode* compilado:

```
$ python maian.py -b <contrato>
```

```
ubuntu@smartcontracts:~/ALBERTO/EthIR/ethir$ ./oyente-ethir.py -h
usage: oyente-ethir.py [-h] -s SOURCE [-glt GLOBAL_TIMEOUT] [-e] [-b]
                       [-disasm] [-d] [-cfg] [-saco] [-c {int,uint}]
                       [-v {cpa,verymax,seahorn}] [-i {array,div0,all}] [-g]
                       [-opt] [-f FIELDS] [-cname CONTRACT_NAME] [-bl BLOCK]
                       [-hashes]

optional arguments:
  -h, --help            show this help message and exit
  -s SOURCE, --source SOURCE
                        local source file name. Solidity by default. Use -b to
                        process evm instead. Use stdin to read from stdin.
  -glt GLOBAL_TIMEOUT, --global-timeout GLOBAL_TIMEOUT
                        Timeout for symbolic execution
  -e, --evm             Do not remove the .evm file.
  -b, --bytecode        read bytecode in source instead of solidity file
  -disasm, --disassembly
                        Consider a dissasembly evm file directly
  -d, --debug           Display the status of the stack after each opcode
  -cfg, --control-flow-graph
                        Store the CFG
  -saco, --saco         Translate EthIR RBR to SACO RBR
  -c {int,uint}, --cfile {int,uint}
                        Translate EthIR RBR to SACO RBR
  -v {cpa,verymax,seahorn}, --verify {cpa,verymax,seahorn}
                        Applies abstraction depending on the verifier
                        (CPAchecker, VeryMax or SeaHorn). Use with -c flag
  -i {array,div0,all}, --invalid {array,div0,all}
                        Translate the specified invalid bytecodes into SV-COMP
                        error labels. Use with -c flag
  -g, --goto            Transform recursive rules into iterative rules using
                        gotos. Use with -c flag
  -opt, --optimize      Fields to be optimized by Gasol
  -f FIELDS, --fields FIELDS
                        Fields to be optimized by Gasol
  -cname CONTRACT_NAME, --contract_name CONTRACT_NAME
                        Name of the contract that is going to be optimized
  -bl BLOCK, --block BLOCK
                        block to be optimized
  -hashes, --hashes     Generate a file that contains the functions of the
                        solidity file
```

Figura 3.6: Listado de diferentes opciones para ethir

Para seleccionar la categoría de contrato inteligente habrá que introducir:

- Suicidal: -c 0
- Prodigal: -c 1
- Greedy: -c 2

3.3.13. Erays

Como la propia descripción del Git de Erays dice, es una herramienta de ingeniería inversa para facilitar el proceso de lectura del bytecode.
El archivo final de salida se genera usando graphviz, una librería que permite la generación de gráficas y diagramas que debe ser instalado previamente.

Tras ejecutar Erays con un contrato la herramienta generó 5 archivos. En la Figura 3.7 pueden verse tres de ellos. Cada archivo contiene la "descompilaciónçorrespondiente de cada función (el creador de Erays menciona que la herramienta no es un descompilador

como tal, pero es una mejor manera de entender los archivos hexadecimales). Por lo tanto, Erays es una herramienta muy simple pero muy útil que además si se complementa con herramientas del tipo Solgraph puede dar una amplia idea del flujo de direcciones que toma un contrato inteligente.

Figura 3.7: Resultados de la ejecución de Erays.

3.3.14. Rattle

Uno de los problemas que tiene la EVM es la dificultad de rastrear y seguir las variables debido a la pila de llamadas implementada en la propia máquina.

Rattle tiene el mismo propósito que Erays (y prácticamente las mismas dependencias), dando como salida un diagrama de flujo de control e intentando eliminar los códigos de operación de la EVM para facilitar el proceso de lectura, transformando pila de llamadas en *Single Static Assignment* (SSA). SSA es una representación intermedia en el diseño del compilador, dando a cada variable un valor antes de ser utilizada. El beneficio más notable de eliminar es eliminar más del 60 % de las instrucciones de la EVM. Rattle intenta proporcionar una manera sencilla de confiar en contratos inteligentes desconocidos rastreando las variables y haciendo que un contrato sea al mismo tiempo auditable y fácil de leer.

Para las pruebas se ha cogido el archivo de prueba que se puede encontrar en el repositorio de la herramienta. Ejecutando.

```
$ python3 rattle-cli.py --input inputs/kingofether/KingOfTheEtherThrone.bin -O
```

Los resultados obtenidos que pueden verse en la Figura 3.8 y la Figura 3.9 son sorprendentes, obteniendo un flujo claro de la ejecución del contrato.

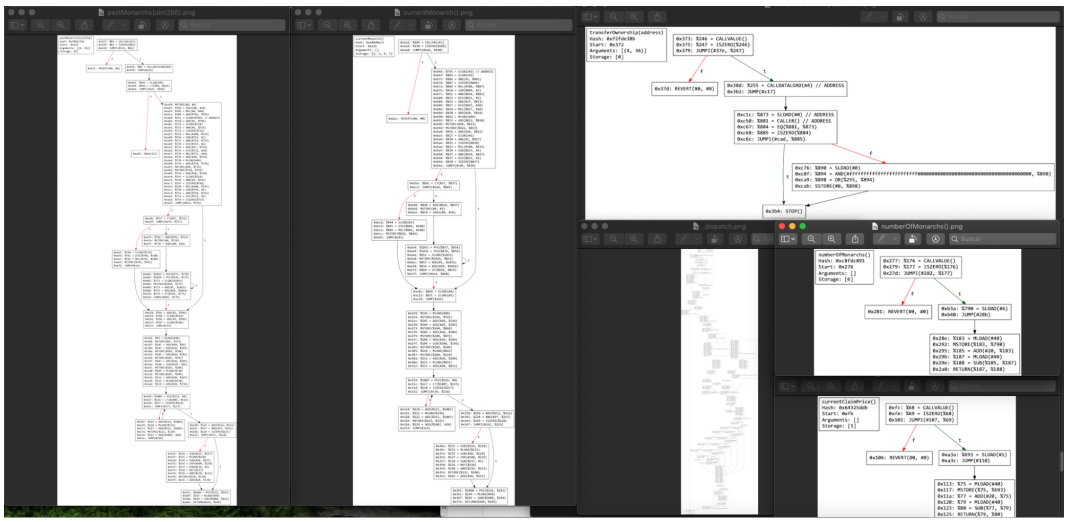

Figura 3.8: Resultados de salida tras el análisis con Rattle

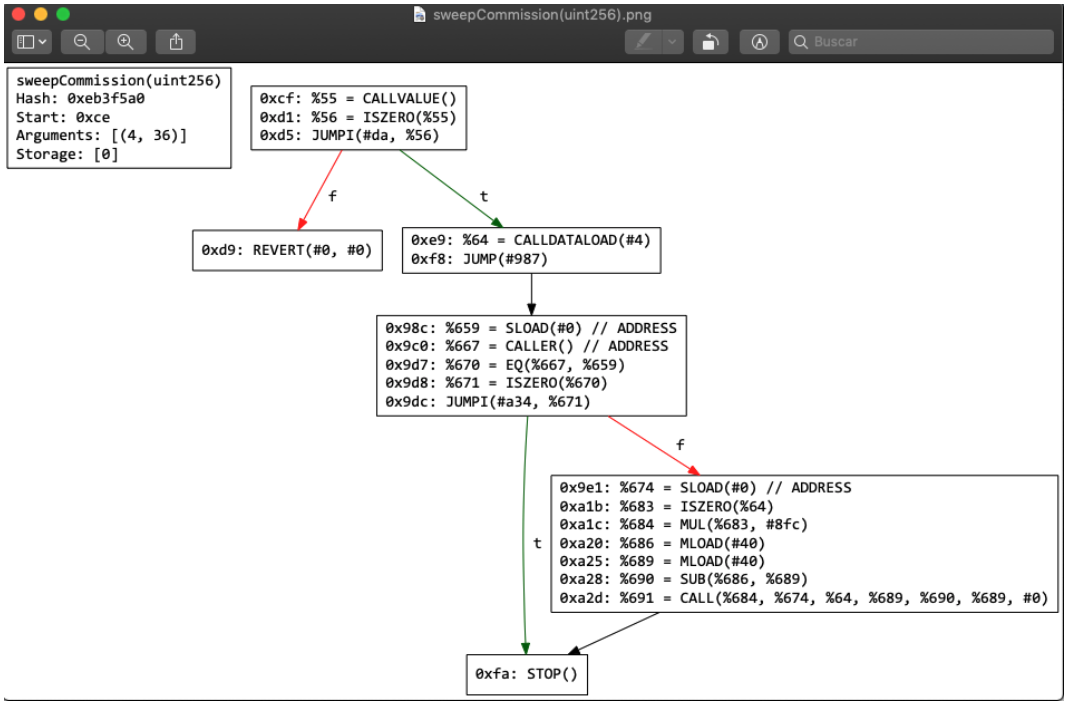

Figura 3.9: Ampliación de un de los archivos de salida

Como se puede comprobar es posible visualizar el diagrama de flujo de control de cada función. Uno de los problemas es entender cada código de operación, asignación, etc. Para ello, es muy recomendable leer la enciclopedia [rat19] para poder entender el gráfico totalmente.

3.3.15. Osiris

Osiris es una extensión de la herramienta Oyente que permite la detección de fallos sobre variables númericas (intergers). Depende de las mismas bibliotecas al igual que Oyente: web3, z3 Theorem Prover (versión 4.6.0), Python y Pysha3. La herramienta muestra la salida de la misma manera que Oyente como puede verse en la Figura 3.10, pero comprobando diferentes vulnerabilidades. Además, muestra un flujo dónde podría estar el fallo en cuestión y da información sobre el tipo de vulnerabilidad.

```
root@fea8b9b2032c:~# python osiris/osiris.py -s datasets/SimpleDAO/SimpleDAO_0.4.19.sol

  .oooooo.             o8o            o8o
 d8P'  `Y8b            `"'            `"'
888      888  .oooo.o oooo  oooo d8b oooo   .oooo.o
888      888 d88(  "8 `888  `888""8P `888  d88(  "8
888      888 `"Y88b.   888   888      888  `"Y88b.
`88b    d88' o.  )88b  888   888      888  o.  )88b
 `Y8bood8P'  8""888P' o888o d888b    o888o 8""888P'


INFO:root:Contract datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     99.7%
INFO:symExec:     Arithmetic bugs:       False
INFO:symExec:     └> Overflow bugs:      False
INFO:symExec:     └> Underflow bugs:     False
INFO:symExec:     └> Division bugs:      False
INFO:symExec:     └> Modulo bugs:        False
INFO:symExec:     └> Truncation bugs:    False
INFO:symExec:     └> Signedness bugs:    False
INFO:symExec:     Callstack bug:         False
INFO:symExec:     Concurrency bug:       False
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        False
INFO:symExec:    --- 0.291406869888 seconds ---
INFO:symExec:   ====== Analysis Completed ======

INFO:root:Contract datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory2:
INFO:symExec:Running, please wait...
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM code coverage:     99.8%
INFO:symExec:     Arithmetic bugs:       False
INFO:symExec:     └> Overflow bugs:      False
INFO:symExec:     └> Underflow bugs:     False
INFO:symExec:     └> Division bugs:      False
INFO:symExec:     └> Modulo bugs:        False
INFO:symExec:     └> Truncation bugs:    False
INFO:symExec:     └> Signedness bugs:    False
INFO:symExec:     Callstack bug:         True
datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory2:61:16
owner.send(this.balance)
^
INFO:symExec:     Concurrency bug:       True
Flow 1:
datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory2:61:16
owner.send(this.balance)
^
Flow 2:
datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory2:55:5
dao.donate.value(1)(this)
^
INFO:symExec:     Time dependency bug:   False
INFO:symExec:     Reentrancy bug:        True
datasets/SimpleDAO/SimpleDAO_0.4.19.sol:Mallory2:56:5
dao.withdraw(1)
^
```

Figura 3.10: Resultados de un análisis con Osiris.

Osiris es una herramienta sencilla, aunque se ha visto la necesidad de usar su propio Docker debido a las fuentes dependencias con librerías de terceros. Ofrece información sobre vulnerabilidades saliéndose el perfil clásico lo cual es un punto a favor además de mencionar donde se encuentra el fallo.

3.3.16. Securify

Securify ofrece una interfaz gráfica de usuario. El proyecto colabora con la propia fundación de Ethereum. Las características de análisis que ofrece son:

- Diferentes formas en las que puede encontrarse el fallo comúnmente conocido como (vulnerabilidad de reentrada).
- Ethers bloqueados.
- Falta la validación de los parámetros de entrada.
- Dependencia de la cantidad de ethers según el orden de entrada en un contrato.
- Excepciones sin capturar.
- Flujo de ether sin renstricciones.

En el apartado de las dependencias, Securify trabaja con Soufflé, que analiza estáticamente el código hexadecimal de un contrato inteligente. Después Securify se encargará de descubrir las violaciones y probar el cumplimiento de los requisitos de seguridad.

Tras ejecutar uno de los archivos de prueba básicos que los creadores proporcionan los resultados del análisis pueden verse en la Figura 3.11. El comando ejecutado fue el siguiente:

```
$ java -jar build/libs/securify.jar -fs \
src/test/resources/solidity/transaction-reordering.sol
```

Y tal y como se menciona previamente, sólo se muestran advertencias y vulnerabilidades, en este caso se puede ver una excepción sin manejar y una violación del flujo de éter sin ninguna restricción.

Usando el flag –json se puede comprobar el número de violaciones, advertencias, funciones de seguridad y conflictos encontrados como puede verse en la Figura 3.12.

Por lo tanto, es una herramienta realmente útil, que cuantifica el número de violaciones y posibles fallos (advertencias) además de posicionar el posible lugar de estos.

```
  Success. Inlining methods...
  Propagating constants...
  Verifying patterns...
Violation for TODAmount in contract 'MarketPlace':
    |    function unsafeTransfer() public {
    |        uint x = address(this).balance;
  > |        msg.sender.send(x);
    |    }
    |}
  at src/test/resources/solidity/transaction-reordering.sol(16)

Warning for TODAmount in contract 'MarketPlace':
    |    function safeTransfer1() payable public {
    |        uint x = msg.value;
  > |        msg.sender.send(x);
    |    }
    |
  at src/test/resources/solidity/transaction-reordering.sol(5)

Violation for UnhandledException in contract 'MarketPlace':
    |    function safeTransfer1() payable public {
    |        uint x = msg.value;
  > |        msg.sender.send(x);
    |    }
    |
  at src/test/resources/solidity/transaction-reordering.sol(5)

Violation for UnhandledException in contract 'MarketPlace':
    |        uint x = 100;
    |        uint y = x * 100;
  > |        msg.sender.send(y);
    |    }
    |
  at src/test/resources/solidity/transaction-reordering.sol(11)

Violation for UnhandledException in contract 'MarketPlace':
    |    function unsafeTransfer() public {
    |        uint x = address(this).balance;
  > |        msg.sender.send(x);
    |    }
    |}
  at src/test/resources/solidity/transaction-reordering.sol(16)

Violation for UnrestrictedEtherFlow in contract 'MarketPlace':
    |        uint x = 100;
    |        uint y = x * 100;
  > |        msg.sender.send(y);
    |    }
    |
  at src/test/resources/solidity/transaction-reordering.sol(11)

Warning for UnrestrictedEtherFlow in contract 'MarketPlace':
    |    function safeTransfer1() payable public {
    |        uint x = msg.value;
  > |        msg.sender.send(x);
    |    }
    |
  at src/test/resources/solidity/transaction-reordering.sol(5)

Warning for UnrestrictedEtherFlow in contract 'MarketPlace':
    |    function unsafeTransfer() public {
    |        uint x = address(this).balance;
  > |        msg.sender.send(x);
    |    }
    |}
  at src/test/resources/solidity/transaction-reordering.sol(16)
```

Figura 3.11: Resultados de un análisis con Securify.

```
[ubuntu@smartcontracts:~/ALBERTO/securify$ java -jar build/libs/securify.jar --json -fs src/test/resources/solidity/transaction-reordering.sol
Processing contract: src/test/resources/solidity/transaction-reordering.sol:MarketPlace
  Attempt to decompile the contract with methods...
  Success. Inlining methods...
  Propagating constants...
  Verifying patterns...
{
  "src/test/resources/solidity/transaction-reordering.sol:MarketPlace": {
    "version": "v0-153-g2ebb518",
    "results": {
      "DAO": {
        "violations": [],
        "warnings": [],
        "safe": [],
        "conflicts": []
      },
      "DAOConstantGas": {
        "violations": [],
        "warnings": [],
        "safe": [
          4,
          10,
          15
        ],
        "conflicts": []
      },
      "LockedEther": {
        "violations": [],
        "warnings": [],
        "safe": [
          1
        ],
        "conflicts": []
      },
      "MissingInputValidation": {
        "violations": [],
        "warnings": [],
        "safe": [
          2,
          7,
          13
        ],
        "conflicts": []
      },
      "RepeatedCall": {
        "violations": [],
        "warnings": [],
        "safe": [
          4,
          15
        ],
        "conflicts": []
      },
      "TODAmount": {
        "violations": [
          15
        ],
        "warnings": [
          4
        ],
        "safe": [
          10
        ],
        "conflicts": []
      },
      "TODReceiver": {
        "violations": [],
        "warnings": [],
```

Figura 3.12: Resultados de un análisis con Securify con el flag json.

3.3.17. MadMax

MadMax es una extensión de la herramienta Vandal, dando la posibilidad de encontrar vulnerabilidades de gas. La herramienta analiza el *bytecode* de la EVM de la misma manera que Vandal (tiene las mismas opciones y no se necesita una biblioteca extra para trabajar).

MadMax proporciona:

- El uso del descompilador estático de *bytecodes* EVM, principalmente por el problema con la naturaleza de bajo nivel de la pila y las estructuras de bajo flujo de control (que se mencionó previamente).
- Permite encontrar las vulnerabilidades de gas.
- Da una abstracción para entender el código de bytes de la EVM. MadMax proporciona una estructura de datos de alto nivel permitiendo la aproximación entre las vulnerabilidades de bajo y alto nivel de la EVM.
- Permite validar un gran número de contratos ya desplegados en la cadena de bloques Ethereum

Si se usa el comando previo la herramienta generará los siguientes resultados para una vulnerabilidad de desbordamiento de enteros, excediendo el tamaño máximo de una variable *uint* como puede verse en la Figura 3.13.

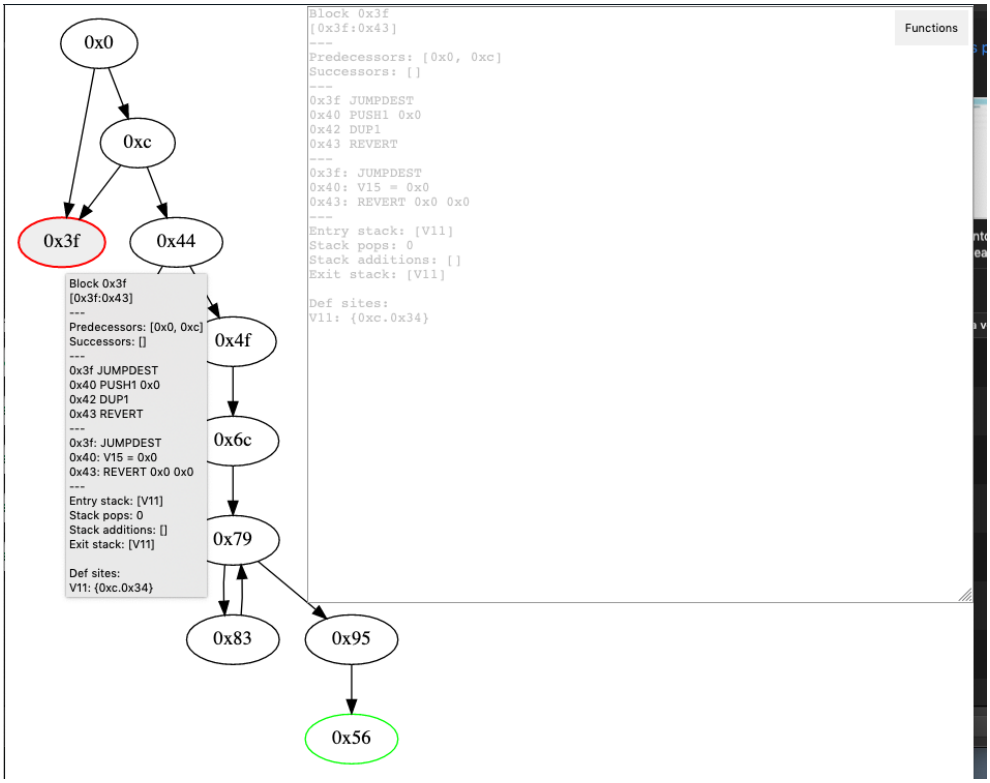

Figura 3.13: Diagrama de ejecución generado por Mad Max

El diagrama de flujo de control proporciona una buena manera de entender las diferentes llamadas, saltos y cómo funciona la pila. Además, el botón superior de "funciones"permite filtrar los nodos del grafo por diferentes tipos: públicos, privados, fallbacks, etc.

3.3.18. ContractLarva

ContractLarva es una herramienta de verificación de tiempo de ejecución, compilada con GHC (Glasgow Haskell Compiler) que es muy impresionante porque rompe el modelo de todas las herramientas anteriores. El código en sí está sin documentar, aunque se puede encontrar algo de información en la carpeta doc. En ella se menciona: *"La herramienta toma las especificaciones utilizando eventos dinámicos automatizados y un contrato inteligente, y produciendo un nuevo contrato de Solidity que actúa igual que el original, pero comprueba el comportamiento en tiempo de ejecución contra la especificación, permitiendo que se realicen reparaciones en caso de violación"*. Es decir, la herramienta coge el contrato pasado como parámetro y lo modifica añadiéndole una serie de eventos o automatizaciones para poder modificar la lógica del mismo.

Tras la ejecución del contrato con el fichero DAE previamente mostrado, se genera un nuevo contrato con los cambios correspondientes. En la Figura 3.14 y la Figura 3.15 se muestran el contrato antes del análisis y después.

```
MacBook-Pro-de-Alberto:tests Ture$ smartcheck -p uninitialized.sol
uninitialized.sol
jar:file:/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle/smartcheck-2.0-jar-with-dependencies.jar!/solidity-rules.xmlrul
eId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 1
column: 16
content: ^

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 19
column: 4
content: functioninit(){balancesInitialized[msg.sender]=0;}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 23
column: 4
content: functionuse(){require(balances[msg.sender]==balancesInitialized[msg.sender]);}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 35
column: 4
content: functionset(MyStructstoragest,uintv){st.val=4;}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 49
column: 4
content: functioninit(){stInitiliazed.set(v);}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 53
column: 4
content: functionuse(){require(st.val==stInitiliazed.val);}

ruleId: SOLIDITY_VISIBILITY
patternId: b51ce0
severity: 1
line: 5
```

Figura 3.14: Contrato antes de usar ContractLarva.

```
pragma solidity ^0.5.11;
contract LARVA_Casino {
  modifier LARVA_DEA_2_handle_after_timeoutBet__no_parameters {
    if ((LARVA_STATE_2 == 2)) {
      LARVA_STATE_2 = 1;
    }
    _;
  }
  modifier LARVA_DEA_2_handle_after_resolveBet__no_parameters {
    if ((LARVA_STATE_2 == 2)) {
      LARVA_STATE_2 = 1;
    }
    _;
  }
  modifier LARVA_DEA_2_handle_after_placeBet__no_parameters {
    if ((LARVA_STATE_2 == 1)) {
      LARVA_STATE_2 = 2;
    }
    _;
  }
  modifier LARVA_DEA_2_handle_after_openTable__no_parameters {
    if ((LARVA_STATE_2 == 0)) {
      LARVA_STATE_2 = 1;
    }
    _;
  }
  modifier LARVA_DEA_2_handle_after_closeTable__no_parameters {
    if ((LARVA_STATE_2 == 1)) {
      LARVA_STATE_2 = 0;
    } else {
      if ((LARVA_STATE_2 == 2)) {
        LARVA_STATE_2 = 3;
      }
    }
    _;
  }
  modifier LARVA_DEA_1_handle_after_assignment_pot {
    _;
    if ((LARVA_STATE_1 == 2) && (LARVA_previous_pot > pot)) {
      LARVA_STATE_1 = 3;
    }
  }
  modifier LARVA_DEA_1_handle_after_timeoutBet__no_parameters {
    if ((LARVA_STATE_1 == 2)) {
      LARVA_STATE_1 = 0;
    }
    _;
  }
  modifier LARVA_DEA_1_handle_after_resolveBet__no_parameters {
    if ((LARVA_STATE_1 == 2)) {
      LARVA_STATE_1 = 0;
    }
    _;
  }
```

Figura 3.15: Contrato después de usar ContractLarva.

En conclusión, ContractLarva es una buena herramienta, todavía en desarrollo, que da una buena forma de modificar la ejecución inicial de un contrato, pudiendo depurar y encontrar diversos problemas añadiendo las especificaciones correspondientes.

3.3.19. Conclusiones Sobre las Herramientas

Tras haber probado y ejecutado todas las herramientas anteriores se han encontrado tanto herramientas útiles como herramientas que o bien porque son complejas en la instalación o porque no aportan valor, no merecen la pena ser utilizada.

Es importante hacer una diferenciación clara entre aquellas que usan *bytecode* y Solidity puesto que se enfocan en diferentes tipos. Por ejemplo, aquellas que solo usan Solidity están más enfocadas a la creación de grafos, inclusión de código nuevo y localización de errores en los puntos de código en base a análisis estáticos más que los que están enfocados en análisis de bytecode que se centran en encontrar una serie de vulnerabilidades específicas.

Uno de los problemas más comunes es la inestabilidad de las herramientas a lo largo del tiempo. Muchas de ellas son creadas con mero fin académico, quedándose desfasadas debido a la evolución de las librerías, frameworks, versiones de Solidity y del compilador, provocando que dejen de ser útiles de aquí a un futuro. Además, muchas de ellas están enfocadas para un usuario avanzado/experto en Solidity y Ethereum por varias razones:

- Debe de conocer Solidity y además, las diferentes modificaciones que se han ido realizando con el paso del tiempo. Esto es así porque cada herramienta trabaja con una versión diferente del código y por tanto, una herramienta puede contemplar un error que otra no.
- Conocimiento completo de los fallos que se pueden dar a la hora de programar. Aquel usuario de las correspondientes herramientas debe de conocer el pasado de Ethereum para saber el por qué de la búsqueda de ese tipo de vulnerabilidades, no solo le basta con haber aprendido Solidity recientemente.
- Uso de librerías específicamente diseñadas para los propósitos de estas librerías. Durante la instalación y utilización de estas herramientas se ha perdido mucho tiempo en los procesos de instalación. Hacer una instalación limpia es costoso puesto que uno depende muchas veces de versiones específicas y en caso de que una herramienta posea una dependencia de la misma herramienta puede llevar a fallos. Eso hace que el uso de los contenedores que proporcionan los desarrolladores sea casi obligatorio. Este tema se abordará en el siguiente capítulo

En la Tabla 3.2 puede verse una comparativa de las herramientas testadas. Algunas de ellas no están incluidas debido a fallos en los procesos de instalación, falta de documentación, desfasadas temporalmente etc. Por último, si es de interés para el lector en el Apéndice A se explica el funcionamiento de cada una de estas herramientas.

Tabla 3.2: Resumen de las herramientas analizadas

Herramienta	Tipo de Análisis	Facilidad de Instalación	Utilidad	Actualizaciones Periódicas	Dependencias	Comentarios
Manticore	Dinámico	5	5	Sí	solc, python3	
Smartcheck	Estático	5	5	Sí	npm	
Manticore	Dinámico	5	5	Sí	solc, python3	
FSolidM*	No	4	5	No	nodejs, mongodb	
Mythril	Estático	5	5	Sí	npm, python3, pip3	Simbólica
ContractLarva	Dinámico	3	3	Sí	haskell (ghc)	La instalación es su punto débil
Solmet	Estático	5	4	No	Java, maven	
Vandal	Estático	5	4	No está claro	csv reader, python3	
EthIr	Dinámico	1	2	Sí	Similar a Oyente	La complejidad del lenguaje y el uso de un segundo lenguaje aumenta la curva de aprendizaje
MAIAN	Estático	1	1	No	solc, z3, python3, web3	
Erays	Estático	5	3	No	graphviz, python2	
Rattle	Estático	5	5	No	graphviz, solc, python3	
Osiris	Dinámico	2	5	No	Similar a Oyente	Hizo falta su docker
Securify	Estático	5	5	Existe una versión 2	soufflé, java8, solc	
Slither	Estático	5	5	Sí	solc,python3, python2, pip2	
Oyente	Dinámico	1	3	No	z3 prover, web3, pip3, solc, go-ethereum	Herramienta simbólica
Renix-IDE	Estático	5	5	Sí	nodejs, npm	
Solgraph	Estático	5	5	No	graphviz	
MadMax	Estático	4	5	No	Basada en Vandal	
EtherTrust	Estático	1	-	No	z3 prover, python2, maven	No se pudo instalar, sin documentación o guía.
Casper	-	-	-	-	-	Obsoleta
Teether	Estático	-	-	No parece	-	Fallo al ejecutar

Capítulo 4

Meta-Herramienta de Análisis de Vulnerabilidades en Contratos Inteligentes

4.1. Justificación y Alcance

Uno de los problemas principales que una persona puede afrontar en cuanto a la seguridad de su contrato inteligente es el gran número de posibles fallos que dicho contrato puede tener. Además, debe hacer frente a una pregunta, ¿cómo puedo analizar y probar de una forma rápida y sencilla todos los casos (a priori) posibles? Es aquí donde surge el propósito de esta herramienta: juntar, unificar y centralizar de una forma sencilla las diferentes herramientas que hay en la actualidad para el análisis de contratos inteligentes. Previamente se han descrito una serie de herramientas con un próposito específico, cada una desarrollada en un lenguaje diferente y versiones de Solidity diferentes. Esto ha provocado que el alcance de la herramienta se vea modificada a medida que se han ido incluyendo dados los requerimientos específicos de cada una. Finalmente la meta-herramienta pretende permitir el análisis masivo de contratos, que posibilite un análisis posterior de todos los resultados para buscar patrones, estadísticas y consenso de forma sencilla. Además, proveerá la posibilidad de análisis también a pequeña escala, de tal forma que si un usuario quiere analizar un único contrato, pueda hacerlo de forma sencilla.

4.2. Entorno y Tecnologías

El entorno de desarrollo y de despliegue de la herramienta ha tenido dos vertientes. En el apartado 2 de esta memoria se ha hablado del estudio de las herramientas usadas. Para ello se ha usado una máquina virtual con Ubuntu 18.04 LS [Incluir referencia] que sirvió de entorno de desarrollo. En el se fueron instalando todas las herramientas con sus diferentes dependencias (z3 prover, python3, python2...) El problema principal que se tuvo fue que cada herramienta trabajaba prácticamente con sus propias dependencias y unas entraban en conclicto con otras llevando el planteamiento tras el análisis de estas al uso de contenedores. A continuación se hace una pequeña descripción y justificación del software utilizado para desarrollar el trabajo:

1. **Python:** Dada la gran cantidad de herramientas que usan Python tanto en su versión 2 como en la 3, se decidió enfocar el desarrollo mediante este lenguaje. Python ha proporcionado una manera cómoda de hacer un script robusto haciendo uso de las librerías que proporciona, en concreto:

   Es la biblioteca que más investigación ha necesitado. La documentación y uso no es muy extendida, seguramente provocada por la corta vida de esta y los cambios que han ido realizando sus creadores. Frecuentemente a la hora de resolver fallos de ejecución las soluciones encontradas en internet no han servido motivadas por los cambios de las funciones, parámetros de entradas, etc. Quitando ese tipo de salvedades, Docker SDK proporciona una forma muy fiable de utilizar y manejar los dockers, proporcionando la mayor parte de los comandos de consola en forma de llamadas a funciones que se pueden encontrar en la documentación oficial.

   Es la librería oficial de MongoDB. Proporciona todas las funciones que podemos encontrar en la CLI de mongo, haciendo muy sencilla la implementación de conexiones a BBDD.

   En el desarrollo de programas escritos en Python hay diferentes entornos que pueden funcionar tremendamente bien. Entre todos ellos se eligió PyCharm dada la gratuidad de la versión profesional, la buena integración y simplicidad a la hora de añadir bibliotecas sin tener que instalarlas por linea de comandos y la posibilidad de elección del interpreter de Python deseado.

2. **Docker:** La tecnología escogida para el uso de contenedores ha sido *Docker*. Docker nos proporciona un software estándar que paquetiza o aísla el código y todas sus dependencias permitiendo que las aplicaciones corren de una forma más rápida y además nos de la posibilidad de distribuir dicha imagen del contenedor sin importar la arquitectura y el sistema operativo del huésped.

3. **MongoDB:** Haciendo frente al almacenaje de información en bruto, se consideró utilizar una base de datos no relacional para el almacenaje de los contratos que se fueran descargando del nodo de Ethereum. Por ello, siguiendo la línea tomada se creó otra base en MongoDB para el guardado de los diferentes análisis que se fueran utilizando.

4. **Git:** El control de versionado es una parte fundamental del software. Pese a haber estado el desarrollo focalizado en una única persona, se ha usado Git para mantener el histórico del código, hacer cambios y tener una linea temporal de los cambios introducidos.

4.3. Funcionamiento

La herramienta desarrollada, pese a desempeñar una serie de casos de usos muy concretos, posee una complejidad por el uso de tecnologías recientes, paradigmas como puede ser el uso de micro-servicios y la dependencia en las propias herramientas desarrolladas por terceros desarrolladores que hay que investigar, probar y valorar su posible implementación. En primer lugar, es importante entender como funciona un sistema basado en microservicios, en concreto los contenedores. Según [Red]

> *"Los microservicios son tanto un estilo de arquitectura como un modo de programar software. Con los microservicios, las aplicaciones se dividen en sus*

> *elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son elementos independientes que funcionan en conjunto para llevar a cabo las mismas tareas. Cada uno de esos elementos o procesos es un microservicio"*

La clave es buscar el aislamiento en búsqueda de la usabilidad conjunta, y es lo que se ha buscado con el uso de contenedores.Todas las herramientas previamente comentadas están instaladas en la imágenes que se pueden encontrar y descargar de forma pública desde el repositorio alojado en hub.docker. El funcionamiento de los repositorios es similar al usado en los repositorios de código abierto. Las imagenes son públicas y cualquiera que tenga una cuenta puede descargar la imagen que desee. Desde este punto se realizaron las siguientes acciones.

4.3.1. Contenedores

4.3.1.1. Dockerfile

El archivo dockerfile es un archivo muy importante a la hora de manejar los contenedores. En dicho archivo que puede verse en el Código 4.1 se define en forma de script lineal los recursos que se van a instalar en nuestra imagen seleccionada.
La cantidad de parámetros que podemos configurar es bastante amplia, algunos ejemplos son :

- FROM: define la imagen desde la cual se comenzará el proceso de generación del contenedor.
- LABEL: mediante el uso de este parámetro se pueden agregar etiquetas específicas para el contenedor: nombre, creador, origen del contenedor etc.
- ENV: permite crear variables de entorno en el contenedor.
- RUN: lanza la ejección de un comandos sobre la imagen elegida. Es útil para instalar paquetes que no estén por defecto sobre la imagen que vayamos a usar, modificar permisos, etc.
- WORKDIR: permite seleccionar el directorio por defecto sobre el que trabajará el contenedor.
- USER: permite añadir un nuevo usuario al contenedor.
- ENTRYPOINT: permite seleccionar el programa, archivo o tipo de comando por defecto al llamar al contenedor.

La información que nos proporciona es fundamental para poder manejar los contenedores de terceros con mayor libertar, permitiendo hacer modificaciones sobre que programas se instalan, que usuarios posee o si queremos o no tener un **entrypoint**.

Código 4.1: Docker file

```
FROM ubuntu:18.04
ARG SOLC=0.5.12

# install basic packages
RUN apt-get update && apt-get install -y %% software-properties-common\
    locales

# set correct locale
RUN locale-gen en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en

# install tools
RUN apt-get update &&\
        apt-get -y install\
        wget\
        gdebi

# install souffle
RUN wget https://github.com/souffle-lang/souffle/releases/download/1.6.2/ \
souffle_1.6.2-1_amd64.deb -O /tmp/souffle.deb &&\
        gdebi --n /tmp/souffle.deb
...
COPY requirements.txt /requirements.txt
WORKDIR /sec

# copy and compile securify
COPY . /sec
ENV PYTHONPATH /sec
...
RUN cd /sec/securify/ && securify staticanalysis/testContract.sol

ENTRYPOINT ["python3.7", "securify/__main__.py"]
```

4.3.1.2. Imagen Genérica

Todos los desarrolladores han proporcionado el uso de dockers propios con sus herramientas ya implementadas listas para ser usadas (en el Código 4.2 puede verse proceso de descarga de imágenes de Docker). El problema principal que ha habido aquí es la dependencia de contenedores que no se sabe como estaban montados, si tenían usuarios propios o como estaban montadas dichas imagenes en el archivo dockerfile. Por lo tanto, en un punto inicial se generó una imagen inicial basada en Ubuntu 18.04, el proceso de creación de la herramienta fue:

1 Creacion de un contenedor a partir de la imagen:

2 Instalación de los programas que la mayor cantidad de herramientas fueran a usar con apt install: nodejs, npm, java, git, curl, wget, unzip, graphviz, solc, sol-script.

3 Instalación por separado de cada herramienta siguiendo los pasos especificos de cada repositorio.

Código 4.2: Descarga de imágenes

```
#Descarga de la imagen
$ docker pull ubuntu
#Arranque del docker a partir de la imagen
$ docker start ubuntu
#Apertura de una interfaz CLI con el contenedor
$ docker exec -it ubuntu bash
```

4.3.1.3. Imágenes Personalizadas y Ejecución

Una vez creadas las herramientas, la primera opción valorada fue crear un pequeño script dentro de cada contenedor al cual se le pasaran las instrucciones correspondientes para que lanzaran la herramienta. El script sería el encargado de enviar la correspondiente herramienta las entradas que necesitara y de guardar los resultados en el correspondiente directorio asignado.
A medida que el desarrollo fue avanzando se descartó dada la cantidad de trabajo que suponía y la cantidad de errores que podía generar. La solución ideada consistía en explotar algunas de las bondades de la librería Docker SDK. Antes de hacer las llamadas correspondientes, se debe crear el contenedor a partir de la correspondiente imagen. El formato de dicha llamada es el siguiente:

```
$ docker run -i --name=vsolgraph -v \
$(pwd)/inputs:/tmp/inputs albeture/isolgraph bash
```

Donde:

- name: indica el nombre que se le da a la imagen.
- v: indica el volumen que se monta sobre el docker con formato:

```
<directorio host>:<directorio dentro del docker o cliente>
```

- albeture/solgraph: nombre de la imagen alojada en nuestro entorno.
- bash: interpreter de los comandos que espera recibir el docker.

Además para asegurarnos de que atacheamos siempre la misma ruta se puede escoger volúmenes previamente añadidos con el parámetro volumen-from:

```
$ docker run -i --name=vsecurify --volumes-from vsolgraph \
albeture/isecurify bash
```

4.3.1.3.1. Subida y Descarga del Repositorio Trabajar con múltiples contenedores e imágenes y sus diferentes versiones provoca en diferentes escenarios el renombrado de dichos elementos, sobretodo para hacer un correcto control de versiones.
Una vez que tenemos un contenedor corriendo y queremos guardar el estado del contenedor, se puede generar una imagen a partir del de forma:

```
$ docker commit {rellenar}
```

Esta imagen generada formará parte de la última versión que este guardada en nuestro entorno local. Para hacer una subida de la imagen al repositorio y poder compartirla deberá correrse:

```
$ docker push <tag>:<version>
```

Para descargar las imagenes se ejecutará:

```
$ docker pull <repositorio/contenedor>:<version>
```

4.3.2. Implementación

La estructura desarrollada del código se ha basado en la modulación y diferenciación de los diferentes archivos según la funcionalidad que lleven acabo. Además, se ha buscado implementar algunos patrones de diseño y mantener algunos estándares como pueden ser la injección de dependencias y la programación orientada a objetos. En la Tabla 4.1 pueden verse los distintos módulos que conforman la herramienta.

Tabla 4.1: Scripts y descripción usados por la herramienta

Archivo	**Uso**
app.py	Módulo principal de la aplicación, lanzamiento de test y parseo de argumentos.
app_engine.py	Genera una array de ejecuciones en base a una entrada y a una herramienta y/o herramientas
dependencies_builder.py	En un primer momento se pensó crear un fichero de salida por cada ejecución que se hiciera sobre un contrato inteligente. El módulo de encargaría de aunar la creación de todos los archivos de los cuales dependiera cada herramienta. Finalmente, al haber alojado los resultados de la base de datos su función principal es la de crear los archivos de entrada en código bytecode y Solidity.
md_contracts.py	Módulo encargado de instancia la conexión a la base de datos, se encarga de devolver los cursor es en función de los parámetros de búsqueda, actualizaciones en base de datos, borrar elementos, etc.
contract_class.py	Modelo que usa la aplicación para instancia de un objeto contrato. Posee métodos ”gettersz ”setters”.
constants.py	Se alojan los diferentes variables constantes del programa, rutas a directorios, herramientas disponibles, etc.
exceptions.py	Fichero con las acciones propias de la herramienta.
stack_trace.log	Fichero donde se alojan los logs (info, warning, exception) de la aplicación.

Para entender el flujo de trabajo de la herramienta se debe comenzar por el primer punto, ¿qué parámetros de entrada contempla?

En la Tabla 4.2 podemos comprobar que los argumentos de entrada son realmente simples. Se debe seleccionar al menos uno de los 4 primeros parámetros para comenzar un análisis, si esto no es así se lanzará una excepción.

4.3.2.1. Diagrama de Flujo y Propiedades Previas al Análisis

Una vez que hemos seleccionado una de las múltiples opciones de análisis se ”parsean”los correspondientes argumentos. A continuación, se lanza la función correspondiente, que deberá pedir a la colección **contratos_full** el cursor correspondiente en base a los

Tabla 4.2: Parámetros de entrada de la meta-herramienta

Parámetro	Función
t : test	Lanza el análisis por defecto de la aplicación. Este análisis lanza de forma genérica todas las herramientas y analiza todos los contratos sin fin.
uc : unique-contract	Analiza el contrato pasado por parámetro.
r : range	Analiza el rango entre 2 IDs pasados como parámetro.
fn : from-number	Analiza desde el id inclusive en adelante.
s : select-tool	Permite seleccionar las herramientas específicas para realizar el análisis.
l : list	Muestra la lista de herramientas disponibles.

parámetros de entrada. El cursor se obtiene con las funciones ofrecidas en md_contracts.py, el cual, ofrece las características de un *Data Access Object* (DAO) para poder acceder a la información correspondiente en ambas bases de datos.

En el Código 4.3 se puede ver el contenido del fichero de forma reducida.

Código 4.3: Manejo de la base de datos

```
1  import constants
2  from pymongo import MongoClient
3
4  def mongo_connection():
5      client = MongoClient("127.0.0.1", 27017)
6      db = client[constants.MONGO_DATABASE]
7      collection = db[constants.MONGO_COLLECTION]
8      return collection
9
10
11 def mongo_results_connection():
12     client = MongoClient("127.0.0.1", 27017)
13     db = client[constants.MONGO_RESULTS_DATABASE]
14     collection = db[constants.MONGO_RESULTS_COLLECTION]
15     return collection
16
17
18 def get_contract(id):
19     collection = mongo_connection()
20     cursor = collection.find({'contract_id': id})
21     return cursor.next()
22
23 def get_range_contract(start_id, end_id):
24     collection = mongo_connection()
25     c_contracts = collection.find({'contract_id': {"$gte": start_id,
           ↪ "$lte": end_id}}, no_cursor_timeout=True)
26     return c_contracts
27
28 #Insert results into outputs_full
29 def insert_result(id, address, field, value, time_elapsed):
30     collection = mongo_results_connection()
31     query = {'contract_id': id}
32     new_values = {"$set": {"{}".format(field): {"output":
           ↪ "{}".format(value), "time_elapsed": time_elapsed}},
           ↪ "$setOnInsert": {"contract_id": id, "address": address}}
33     collection.update(query, new_values, upsert=True)
```

Uno de los principales fallos encontrados a la hora de mantener un flujo de análisis muy elevado fue la caducidad de los cursores. Al parecer, mongo permite mantenter cursores abiertos durante un periodo de tiempo determinado aunque se esté produciendo un flujo contante de datos por él. La opción *no_cursor_timeout=True* viene a resolver este problema. Aun así, en las capas superiores a la hora de recorrer el rango de contratos insertado se mantiene de forma periódica una actualización de un nuevo cursor. También es importante tener en cuenta la librería superior llamada *constants*, dicha librería está explicada en la tabla de archivos. Una de sus variables más importantes es la dedicada a las propiedades de cada herramienta utilizada, con la estructura de la Tabla 4.3

Tabla 4.3: Parámetros de entrada de la meta-herramienta

Clave	Descripción
Name	Nombre de la herramienta
Container	Nombre del contenedor en nuestro entorno local asociado a la herramienta
Workdir	Directorio dentro del contenedor donde vamos a trabajar al crear la conexión
Cmd	Comando principal de ejecución del contenedor
Ext	Extensión que admite la herramienta (sol o hex)

Almacenada en una lista de diccionarios clave-valor que puede verse en el Código 4.4. Continuando con el diagrama de flujo, tras obtener el cursor se deberá comenzar la iteración sobre el mismo, de forma que se genera el fichero de entrada haciendo una llamada al módulo *dependencies_builder.py*. Dicho módulo obtendrá a través del identificador interno del contrato su contenido, si hay bytecode y/o solidity creará los ficheros inputs correspondientes y los guardará en la carpeta *input*. A continuación, **creará un objeto contrato,** que será el que se pase al módulo *app_engine.py*, para comenzar su análisis.

Código 4.4: Ejemplo del contenido para la herramienta solmet

```
...
{
    "name" : "solmet",
    "container": "vsolmet",
    "workdir": "/home/SolMet-Solidity-parser/target/",
    "cmd": "java -jar SolMet-1.0-SNAPSHOT.jar -inputFile
        ↪ ../../../tmp/inputs/{0} -outFile output.csv",
    "ext": ".sol"
},
...
```

4.3.2.2. Diagramas de Flujo y Propiedades Durante el Análisis

Una vez se ha arrancado el análisis correspondiente, la herramienta distinguirá entre los tipos de archivos de entrada que admite la herramienta. En las Figuras 4.1 y 4.2 puede comprobarse el flujo de trabajo de forma general desde que se comienza un análisis hasta que se guarda su resultado en base de datos.

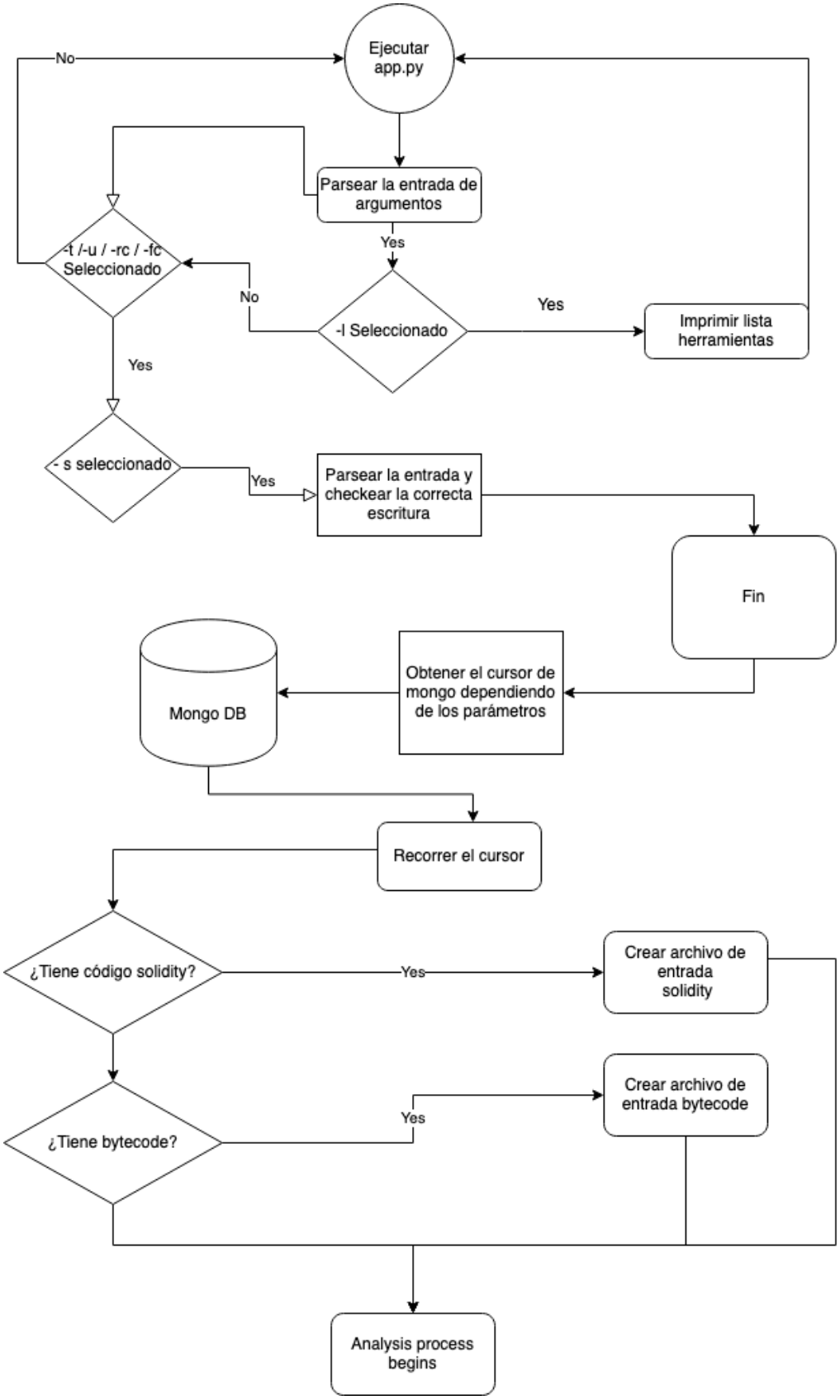

Figura 4.1: Diagrama de flujo: lanzamiento de app.py hasta el comienzo de análisis

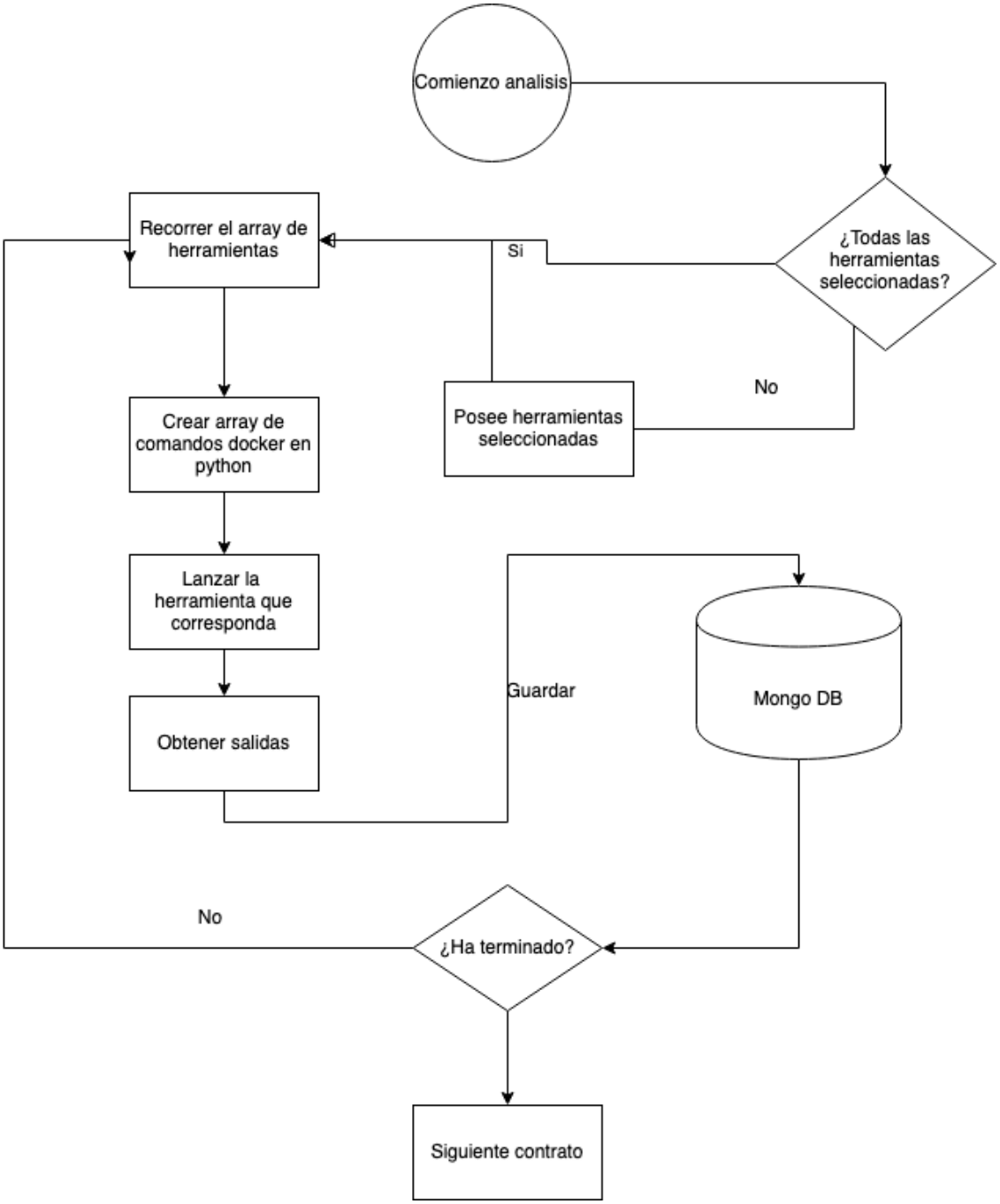

Figura 4.2: Diagrama de flujo: vista general del proceso de análisis

Los puntos clave y a la vez críticos que posee la herramienta se encuentran en la función *Crear un array de comandos para Docker en Python.*

- **Casos de Uso:** Dadas las características de cada herramienta de sebe tener en cuenta los siguientes casos:

  1. Se debe obtener las versiones del código Solidity para las herramientas que analicen por código en alto nivel.
  2. Para las herramientas **Mythril y Slither** se debe de pasar la versión de código ya que dependen de esta para su ejecución.

3. Las herramientas Securify y Slither funcionan con unos rangos específicos del compilador de Solidity. Si la versión de un contrato no corresponde por la esperada no se debe de realizar el análisis.

3. Algunas herramientas no muestran sus resultados por linea de comandos, por lo tanto hay que guardar sus salidas en un archivo temporal y después ejecutar su lectura con el comando **cat**.

4. Todos los comandos que se vayan a lanzar se guardan en un array llamado *executions*. Estas llamadas al docker serán a través de la función **docker_client.exec_create**.

5. Una vez creado el array, se recorre a traves de un *foreach* que lanza la ejecución correspondiente con la función docker **docker_client.exec_start**

6. En último lugar se lee la tupla devuelta y se guarda el resultado correspondiente.

- **Pseudocódigo:** para entender los casos de uso previos, en el Código 4.5 y el Código 4.6 se muestra el pseudocódigo de la herramienta correspondiente al script **app_engine.py**, en concreto las funciones **exec_start y intermediate_exec**.

Código 4.5: Pseudocódigo exec_start

```
1  # Se corren todas las herramientas diferenciando por analisis
2  # files:         array con los archivos input
3  # contract:      objeto contrato para su análisis
4  # tools:         array de herramientas para realizar el análisis
5
6  def exec_start(files, contract, tools):
7      for tool in tools:
8          #selección del archivo de entrada desde files
9          #si existe en el directorio continuar
10
11             # obtención de la versión de Solidity
12             # lanzamiento de la función intermediate
13             result, time_elapsed = intermediate_exec(tool, cmd,
                   ↪ version)
14             # inserción de resulados en la base de datos
15             mdcontracts.insert_result(contract.get_id(),
                   ↪ contract.get_address(), tool.get('name'), result,
                   ↪ time_elapsed)
16
17         # si no, la herramienta no permite el análisis para ese tipo
               ↪ de extension (\texit{bytecode} o Solidity) y se guarda
               ↪ un log genérico para este caso
18             mdcontracts.insert_result(contract.get_id(),
                   ↪ contract.get_address(), tool.get('name'), "Contract
                   ↪ extension doesn't allow this analysis", '0')
```

Código 4.6: Pseudocódigo intermediate_exec

```
...
#tool:      propiedades para una herramienta alojada en constants
#cmd:       comando previo al principal
#version:   version del código analizado
def intermediate_exec(tool, cmd, version):
    if version != '':
        version = int(version.replace(".", ""))
    #comprobar si la herramienta actual depende de la version y si es
        ↪ válida
    #si no, se devuelve un mensaje: de error

    #comando principal
    executions = [docker_client.exec_create(tool.get('container'),
        ↪ cmd=cmd, workdir=tool.get('workdir'),privileged=True)]

    #si la herramienta necesita una lectura de un archivo de
        ↪ resultados dentro del contenedor, crear y añadir al array
        ↪ de ejecuciones la correspondiente función
    if tool.get("name") == 'solmet':
        executions.append(docker_client.exec_create(tool.get('container'),
            ↪ cmd='cat output.csv',workdir=tool.get('workdir'),
            ↪ privileged=True))
    if tool.get("name") == 'ethir':
        ...

    #variable de los resultados de una herramienta
    result = ''
    for exe in executions:
        #Iniciar lap de ejecucion
        #Lanzar la ejecución "exe" que corresponda
        #Una vez acabado para el lap
        for line in res_iter:
            #se recorre el array de resultados y se decodifica
            #se añade al contenido previo de la variable "results"
    return result, (time_end - time_start)
```

Dichas funciones hacen uso de la librería de Python Docker SDK[Inc20] previamente mencionada, proporcionando la posibilidad de abrir un proxy con los docker instalados en nuestra máquina o el lugar donde tengamos el cliente de docker. *Docker_client.exec_create*nos permite configurar una futura ejecución con parámetros como el directorio de trabajo, el contenedor, la conexión para la inserción de comandos, salidas y errores, privilegios, etc. mientras que *Docker_client.exec_start* en cambio, nos permite leer los resultados del docker cliente desde el propio huésped. Los resultados se devuelven mediante diferentes tipos de datos: generator, strings o tuplas. En este caso el código ha sido desarrollado obteniendo los resultados como *generators*. Se debe usar la opción stream=True y demux=True en exec_create para poder leer desde el script ejecutado en el huesped las salidas de las herramientas. Stream devuelve los resultados y demux los separa en dos tipos, mensajes normales y errores. Como resultado final, el diagrama de secuencia de la aplicación puede verse en la Figura 4.3.

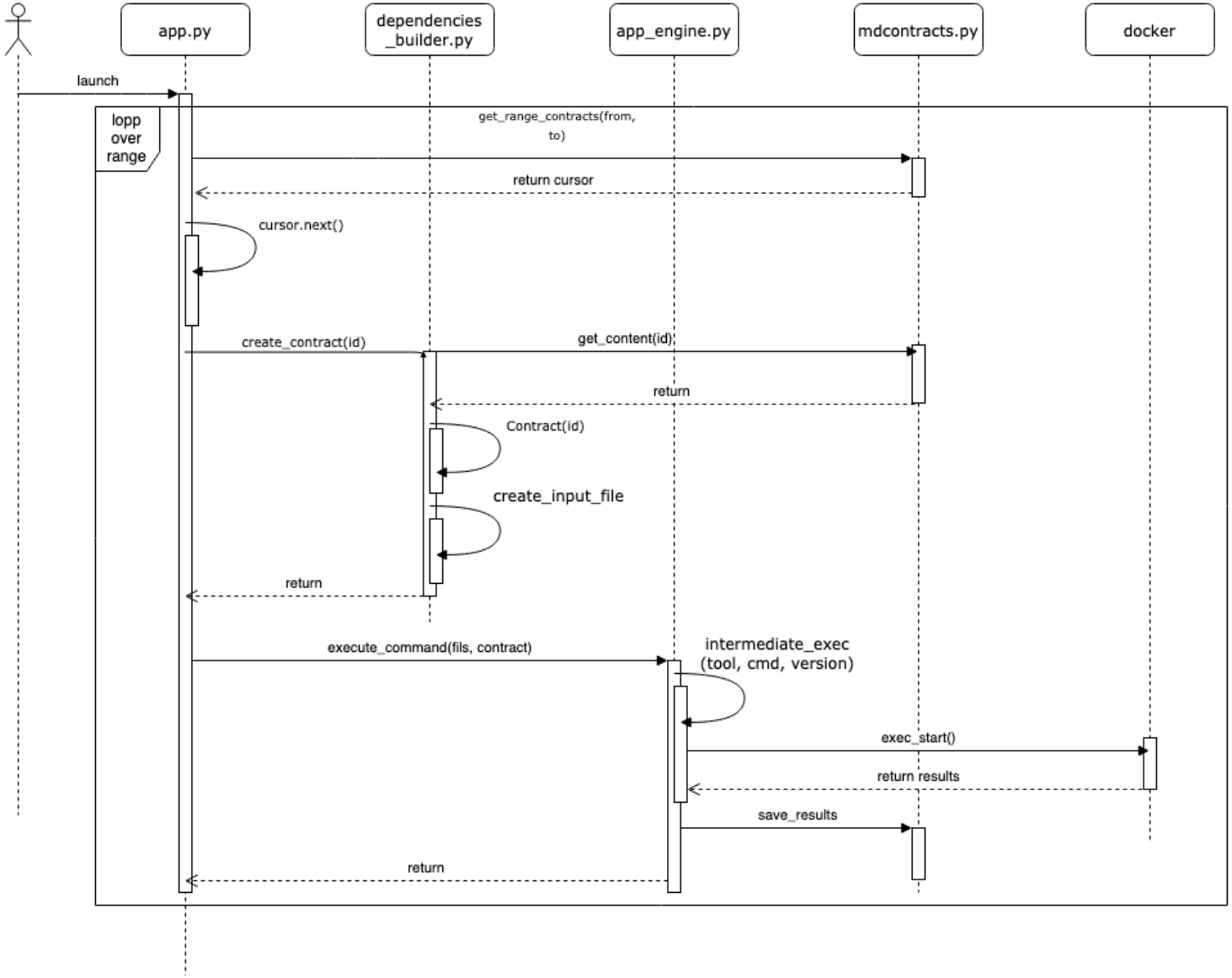

Figura 4.3: Diagrama de secuencia: flujo de trabajo de la aplicación

4.3.3. Interfaz Gráfica

Para simplificar el uso de un usuario medio, se ha desarrollado una interfaz gráfica de usuario que permite el análisis tanto de contratos locales como de contratos que se encuentran ya desplegados en la cadena de bloques de Ethereum. Para el análisis de los contratos se ha hecho uso de la herramienta ya desarrollada previamente de manera local, adaptándola a las necesidades que puede tener una interfaz gráfica como son la inserción de parámetros entrada por componentes visuales, introducción de texto por parte del usuario en los campos correspondientes y selección de aquellas herramientas que quiera usar. En primer lugar se diseñó a modo de guía una maqueta para intentar aunar las diferentes tipos de funcionalidades que dispone actualmente la aplicación y ofrecer una manera sencilla y rápida de proporcionar al usuario las principales funciones.

Posteriormente durante el desarrollo de la aplicación, se modificaron algunos elementos manteniendo la esencia principal.

4.3.4. Funciones y Flujo de Uso

Como puede verse en la Figura 4.4 la interfaz de usuario provisional de la herramienta tiene la siguiente apariencia.

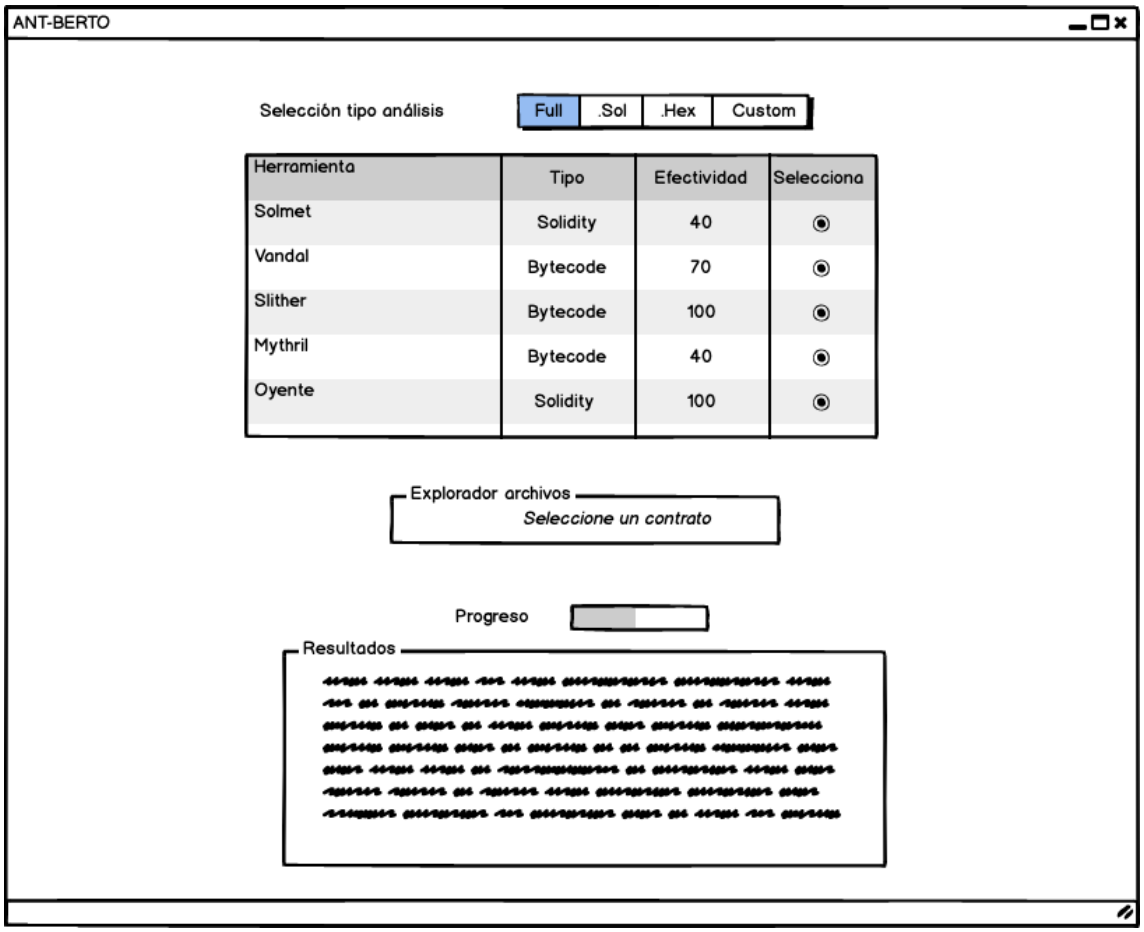

Figura 4.4: Maqueta inicial

En primer lugar el usuario podrá seleccionar qué tipo de análisis desea si análisis de archivos locales (Figura 4.5) o de una dirección en concreto (Figura 4.6). Posteriormente el usuario selecciona aquellas herramientas que quieran que realicen un análisis. Tendrá disponible una serie de botones para el marcado rápido de estos valores, siendo libre de marcar o desmarcar los que desee.

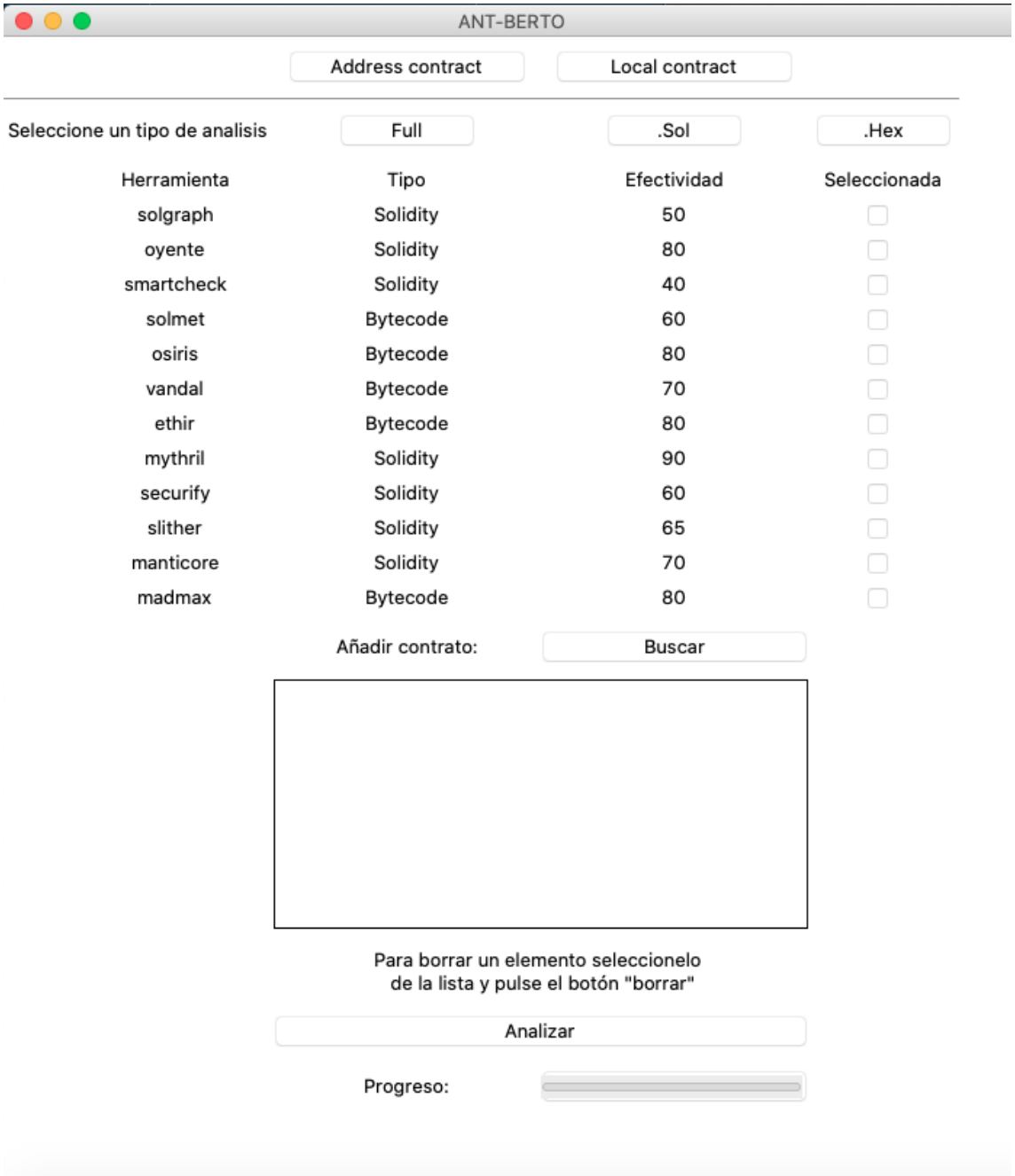

Figura 4.5: Selección de la opción de análisis local

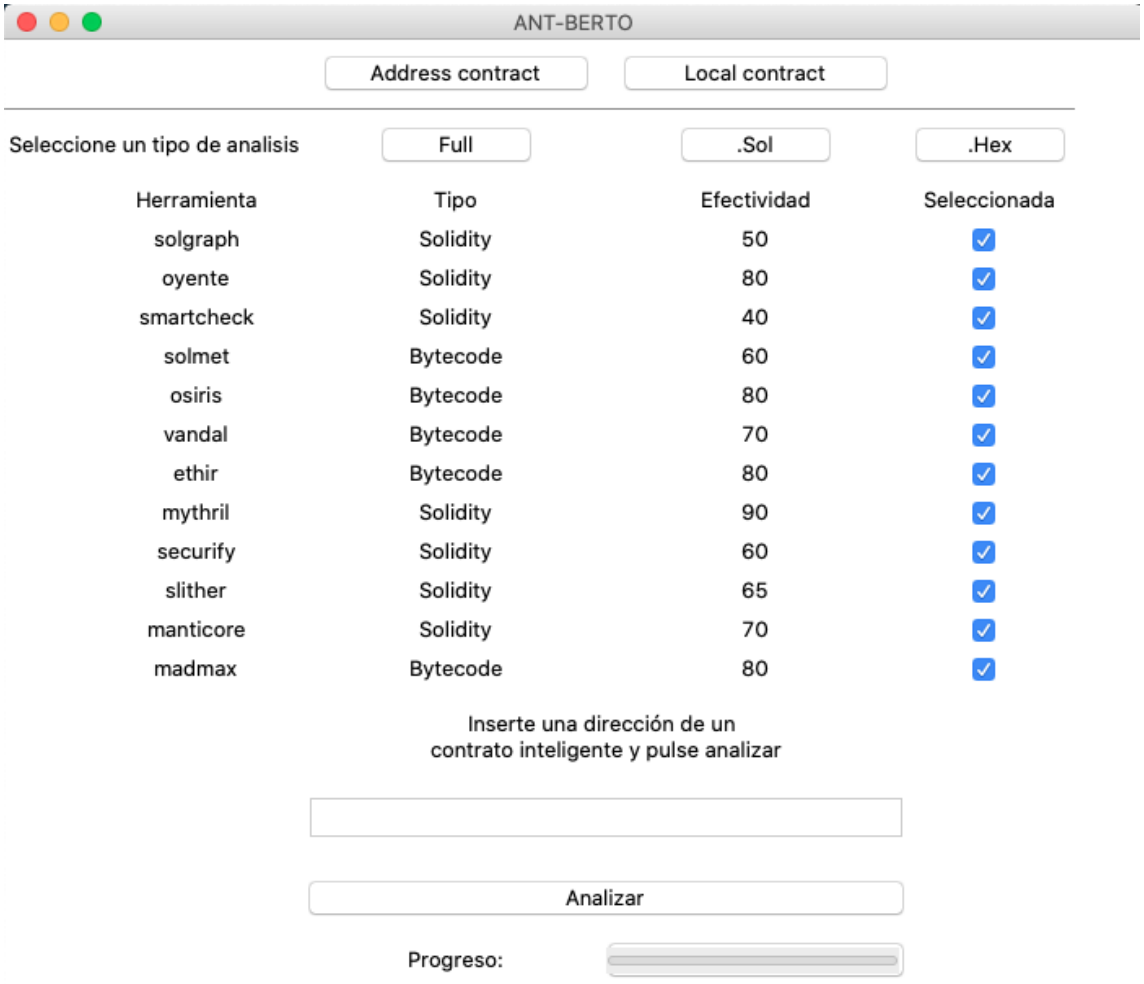

Figura 4.6: Selección de la opción de análisis por dirección

No es que haya marcado las herramientas podrá buscar en el explorador de archivos correspondiente en los contratos locales seleccionarlos y lanzar su ejecución. La herramienta admite tanto contratos del tipo .hex, aquellos que sean del formato .sol.

En el caso de que el usuario seleccione un análisis mediante una dirección se eliminarán aquellos elementos que no dependen de este tipo de análisis como son la selección de contratos por parte del empleador archivos y el listado de estos archivos seleccionados y se añadirá un campo input donde el usuario podrá meter la dirección a analizar. Hay que tener en cuenta que si la de dirección añadida no es una específica de un contrato inteligente, no producirá resultado alguno puesto que este tipo de direcciones no poseen ningún tipo de código, son cuentas de usuario.

4.3.5. Código fuente

Para la creación de la interfaz visual se ha hecho uso como previamente se ha mencionado de la libería Tkinter. Siguiendo un desarrollo basado en una clase principal, **gui.py**, además de un controlador, **gui_controller.py**, para poder comunicar entre el motor de la aplicación por CLI, el archivo **app_engine.py**. Las funciones principales de cada uno de estos archivos son:

- **Gui.py:** es la vista de la aplicación. Se encarga de instanciar la vista principal mediante el uso de dos clases, MainGuiApplication y ResultsFrame. Puede verse en el Código 4.7.

Código 4.7: Pequeño fragmento del archivo gui.py

```
class MainGuiApplication(Frame):
    def __init__(self, parent, *args, **kwargs):
        Frame.__init__(self, parent, *args, **kwargs)
        self.parent = parent
        self.elements = []
        self.tool_checkboxes = []
        ...
        self.buttonRecover.grid(column=3, row=0, padx=10, pady=10,
            ↪ sticky=W + E)

    def __hide(self):
        self.is_local_analysis = False
        self.file_list.grid_remove()
        ...
    def __show(self):
        self.is_local_analysis = True
        for e in self.elements_to_hide:
            e.grid_remove()
        ...
    def __create_header(self, parent):
        sep = Separator(parent, orient=HORIZONTAL)
        sep.grid(row=1, column=0, columnspan=6, sticky=W + E)
        label = Label(parent, text="Seleccione un tipo de analisis",
            width=20).grid(row=2, column=0, columnspan=2,pady=10)
        self.elements.append(label)
        for i in range(0, len(button_headers)):
            __full_option = partial(self.__full_option, i)
            button = Button(parent, text=button_headers[i],
                ↪ command=__full_option,
            width=10, activebackground='red').grid(row=2, column=i +
                ↪ 2, padx=5, pady=5)
            self.elements.append(button)
        ...
    def __analyse(self):
        code = 0
        tool_check = False
        for i in range(0, len(self.tool_checkboxes)):
            if self.tool_checkboxes[i].get():
                tool_check = True
                break
        if (tool_check and len(self.file_paths) > 0) or
            (not self.is_local_analysis and tool_check):
            results, code =
                ↪ gui_controller.start_analysis(self.file_paths,
                self.tool_checkboxes, self.is_local_analysis,
                    ↪ self.entry_address.get())
        else:
            tkmb.showinfo(message="Debe seleccionar al menos una
                ↪ herramienta
                y elegir un contrato",
                          title="Error al introducir los datos")
        if code == 200:
            tkmb.showinfo(message="Informative message")
        ...
```

- **Gui_controller.py:** se encarga de pasarle al motor de la aplicación los archivos que tienen que ser analizados así como las herramientas que ha seleccionado el usuario. En caso de que el usuario seleccione análisis por dirección, se encargará de llamar a la función encargada de ello. En el Código 4.8 puede verse el código de este módulo.

Código 4.8: Funciones del archivo gui_controller.py

```
import ...
def start_analysis(files, tools, is_local_analysis, address):
    app_engine.init()
    if not constants.output_created:
        dependencies_builder.create_output_dir(
            constants.DEFAULT_OUTPUT)
        constants.output_created = True
    if is_local_analysis:
        local_analysis(files, tools)
    else:
        address_analysis(address, tools)
def local_analysis(files_path, tools):
    files = []
    results = []
    for file in files_path:
        ext = str(file).split('.')[1:][0]
        filename = str(file).split('/')[-1]
        files.clear()
        if ext == 'sol':
            files.append(filename)
            files.append('nofile')
        else:
            files.append('nofile')
            files.append(filename)
        path = str(file).replace('/' + filename, '')
        if constants.DEFAULT_DIRECTORY != path:
            dst = os.path.join(constants.DEFAULT_DIRECTORY +
                ↪ constants.DEFAULT_INPUT, filename)
            copyfile(file, dst)
        results.append(app_engine.exec_start_gui(files,
            ↪ get_tools_selected(tools)))
    code = '200'
    return results, code
def address_analysis(address, tools):
    data = contract_downloader.walk_contracts_dir(address)
    contract = dependencies_builder.create_contract_from_address(address,
        ↪ data)
    input_hex_file = 'input_' + contract.get_name() + '.hex'
    if contract.get_extensions().count() > 1:
        input_sol_file = 'input_' + contract.get_name() + '.sol'
    else:
        input_sol_file = 'nofile'
    files = [input_sol_file, input_hex_file]
    results = app_engine.exec_start_gui(files, get_tools_selected(tools))
    code = '200'
    return results, code
def get_tools_selected(tools):
    temp = []
    for i in range(0, len(tools)):
        if tools[i].get():
            temp.append(constants.TOOLS_PROPERTIES[i])
    return temp
```

Para poder analizar una dirección es necesario poder descargar el contrato de la cadena de bloques en tiempo real. Como puede verse en el Código 4.9 se hace uso de la API de Etherescan[Eth20b] que ofrece una forma rápida y sencilla de consumir sus recursos. De la propia API se obtiene el código Solidity (si se encuentra disponible) junto con el bytecode. Dichas llamadas están alojadas en el archivo **contract_downloader.py**, una versión reducida del script utilizado para bajar la cadena de bloques de Ethereum.

Código 4.9: Función de descarga dentro de contract_downloader.py

```
1  def walk_contracts_dir(dir, solidity=True, bytecode=True):
2      contract_error = False
3      log = {}
4
5      etherscan_api_urls = []
6      etherscan_api_url_to_label = {}
7      etherscan_api_responses = {}
8
9      if solidity:
10         url=('https://api.etherscan.io/api?module=contract&action='+
11         'getsourcecode&address='+dir+'&apikey='+ETHERSCAN_APIKEY)
12         etherscan_api_urls.append(url)
13         etherscan_api_url_to_label[url] = 'solidity'
14
15     if bytecode:
16         #generar la petición url
17         #añadir al array de peticiones
18         #añadir una etiqueta
19
20     #exe queries
21
22     if etherscan_api_responses['solidity']:
23         #get data or set error
24
25     if etherscan_api_responses['bytecode']:
26         #get data or set error
27
28     return [solidity_code, bytecode, log, contract_error]
```

4.3.6. Almacenamiento en Base de Datos

La aplicación hace uso de 2 bases de datos.

4.3.6.1. Mydb

Como se muestra en el Código 4.10, se almacena el contenido de la cadena de bloques de Ethereum. Dadas las características de la cadena de bloques no se puede descargar de forma sencilla el contenido completo. En primer lugar se debe sincronizar el nodo de Ethereum de la máquina donde queramos descargar los contratos. Posteriormente podremos comenzar la descarga de los contratos, dicha descarga es lenta debido al uso de APIs. Para hacerse una idea del volumen de contratos alojados, tras dos meses y medio de descarga ininterrumpida el contenido de la base de datos es de 8 millones de bloques sincronizados. Se experimenta además un aumento del tiempo de sincronización y descarga mayor a medida que los bloques van aumentando. Esto se debe al aumento de información que contiene cada bloque de la cadena en comparación con los primeros contratos.

Código 4.10: Ejemplo en formato json de un bloque

```
{
   "_id":"ObjectId("   "5e5d8e64b745e70a1afeef1f"   ")",
   "address":"0x8b180cf51193c6799b79b032dc2510a4b6c084df",
   "bytecode":"0x",
   "function_sighashes":"",
   "is_erc20":"False",
   "is_erc721":"False",
   "contract_id":1000,
   "solidity":"",
   "creation_tx":{
      "blockNumber":"174032",
      "timeStamp":"1441145513",
      "hash":"0x301321c3a216b63096dcafe88394a2de80ba69f5ed9baa668c6072d0f7b
         ↪ 06cb5",
      "nonce":"78",
      "blockHash":"0x9a98685ad088b7a3b6402e27ebadc7243f60ca0d6219e7811bd76
         ↪ aa10a806961",
      "transactionIndex":"8",
      "from":"0xcf684dfb8304729355b58315e8019b1aa2ad1bac",
      "to":"",
      "value":"3",
      "gas":"1000000",
      "gasPrice":"50000000000",
      "isError":"0",
      "txreceipt_status":"",
      "input":"0x60606040525b336000600061........9056", #Abreviado
      "contractAddress":"0x8b180cf51193c6799b79b032dc2510a4b6c084df",
      "cumulativeGasUsed":"987614",
      "gasUsed":"819614",
      "confirmations":"9420468"
   }
}
```

La imagen superior muestra un contrato dentro de la base de datos mydb y a su vez dentro de la colección *contratos full*.
Los parámetros que realmente son representativos para la herramienta son:

- Bytecode: es el código hexadecimal del contrato ya compilado.
- Solidity: código Solidity sin compilar.
- Contract id: contador interno de de insercción.
- Address: dirección del contrato.

4.3.6.2. Results db

Tal como se ve en el Código 4.11 se lmacenan los resultados de los sucesivos análisis que haya realizado la aplicación. Se mantiene fuera de **mydb** ya que el script de descarga de nodos de Ethereum sigue corriendo en el ordenador, por lo tanto buscamos no interferir en el propio proceso de inserción de esa base de datos. A su vez los resultados se almacenan en una colección llamada *outputs_full*

Código 4.11: Ejemplo simplificado en formato json de un análisis


```
{
   "_id":"ObjectId("   "5e9c6c97574ce3bef34d9f15"   ")",
   "contract_id":200,
   "address":"0x2194b1734ee0f67440884da49952a45b34ba832d",
   "solgraph":{
      "output":"strict digraph {\n  mul [color=gray]\n  div [color=gray]\n
          ↪  sub [color=gray]\n  add [color=gray]\n  totalSupply [color=
          ↪ yellow]\n  ...n",
      "time_elapsed":0.021505345000008447
   },
   "smartcheck":{
      "output":"input_contract_200.sol\n\njar:file:/usr/local/lib/
          ↪ node_modules/@smartdec/smartcheck/jdeploy-bundle/smartcheck-2.
          ↪ 0-jar-with-dependencies.jar!/solidity-rules.xml\nruleId: \
          ↪ nSOLIDITY_ERC20_APPROVE\npatternId: \naf782c\n\nseverity: \n2\
          ↪ n\nline: \n90\n\ncolumn: \n4\ncontent: \nfunctionapprove(
          ↪ address_spender,uint256_value)publicreturns(bool){allowed[msg.
          ↪ sender][_spender]=_value;emitApproval(msg.sender,_spender,
          ↪ _value...",
      "time_elapsed":0.05430566500000111
   },
   "solmet":{
      "output":"Parsing /tmp/inputs/input_contract_200.sol\n\nSolidityFile;
          ↪ ETHAddress;ContractName;Type;SLOC;LLOC;CLOC;NF;WMC;NL;NLE;
          ↪ NUMPAR;NOS;DIT;NOA;NOD;CBO;NA;NOI;A...",
      "time_elapsed":0.02934941899999899
   },
   "oyente":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "osiris":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "ethir":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "vandal":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "mythril":{
      "output":"The analysis was completed successfully. No issues were
          ↪ detected.\n\n\n",
      "time_elapsed":0.026984572999992906
   },
   "securify":{
      "output":"",
      "time_elapsed":0
   },
   "slither":{
      "output":"INFO:Detectors:\\x1b[92m\ncustomIcoToken.createTokens()
          ↪ (../../../tmp/inputs/input_contract_200.sol#127-142) compares
          ↪ to a boolean constant:\n\t-require(bool)(isFinalized == false)
          ↪  (../../../tmp/inputs/input_contract_200.sol#128)\
          ↪ ncustomIcoToken.finaliz...",
      "time_elapsed":0.03750889799999868
```

```
    },
    "manticore":{
        "output":"2020-04-19 15:34:44,416: [6] m.main:\\x1b[34mINFO:\\x1b[0m
            ↪ Registered plugins: DetectIntegerOverflow,
            ↪ DetectExternalCallAndLeak, DetectEnvInstruction,
            ↪ DetectUninitializedStorage, DetectUninitializedMemory,
            ↪ DetectReentrancySimple, DetectReentrancyAdvanced,
            ↪ DetectManipulableBalance, DetectUnusedRetVal, DetectSuicidal,
            ↪ DetectInvalid, DetectDelegatecall\n\n2020-04-19 15:34:44,417:
            ↪ [6] m.main:\\x1b[34mINFO:\\x1b[0m Beginning analysis\n\n2020-0
            ↪ 4-19 15:34:44,444: [6] m.e.manticore:\\x1b[34mINFO:\\x1b[0m
            ↪ Starting symbolic create ...",
        "time_elapsed":0.02271556600001645
    },
    "madmax":{
        "output":"Contract extension doesn't allow this analysis",
        "time_elapsed":"0"
    }
}
```


Todas las herramientas muestran un resultado. En caso de que no se pueda analizar por compatibilidad del fichero de entrada el resultado será *contract extension doesn't allow this analysis*. En último lugar, cada herramienta muestra un valor *time_elapsed*, que viene a ser el tiempo de ejecución del análisis.

4.4. Descarga e Instalación

En caso de querer probar la herramienta se deberá de ir al repositorio de código donde se encuentra alojada https://github.com/Ture2/ANT-BERTO y seguir los pasos de instalación que se encuentran en el fichero *readme*.
Algunos de los requisitos más importantes que se hacen mención son:

1 Tener instalado Python 3 y las librerías que usa la herramienta.

2 Tener instalado Docker.

3 Encender en caso de Windows el Hyper-V.

4 Tener instalado MongoDB e importada la base de datos de contratos inteligentes.

Con esas premisas y siguiendo los pasos en dicho archivo puede hacerse uso de la herramienta.

4.5. Dificultades y Limitaciones

Durante el desarrollo de la herramienta surgieron múltiples problemas que dificultaron las labores de desarrollo. Las razones generales de estos problemas son las siguientes:

1 Cada herramienta y proceso de instalación es completamente diferente.

2 La documentación y los procesos de instalación en ocasiones no eran lo suficientemente claros.

3 Muchas de las herramientas tienen una gran dependencia en el uso del contenedor que viene por defecto.

4 Algunas de ellas están desfasadas.

5 Dependencias muy fuertes de la versión del compilador que una herramienta maneje.

6 Instalación de diferentes versiones del compilador para una misma herramienta.

7 Conocimientos de Docker en caso de querer modificar el propio contenedor que nos proporcionan los desarrolladores.

8 Tener que aprender a usar Docker y Python (con sus librerías) para poder desarrollar de cero.

Si nos centramos principalmente herramienta por herramienta se pueden observar dificultades en prácticamente cada una de ellas.

- **Solgraph**: Solgraph debe de instalarse con el siguiente *flag* para evitar errores.

  ```
  --unsafe-perm=true --allow-root
  ```

- **Smartcheck**: Ninguna dificultad.

- **Solmet**: Al tener una dependencia clara con la versión número 8 de Java, en caso de tener alguna otra versión instalada dentro del equipo donde estemos ejecutando la herramienta deberemos instalar dicha versión. Para poder seleccionar una versión diferente si tenemos varias instaladas en el dispositivo es recomendable usar el siguiente comando.

  ```
  $   sudo update-alternatives --config java
  ```

- **Oyente**: Presenta bastantes complicaciones si se quiere hacer una instalación limpia de cero. Se debe usar especificamente la versión de *Z3 Prover* que nos pide el creador. Además, hay que tener cuidado con la versión del puesto que versiones futuras puede provocar que la herramienta deje de funcionar.

- **Osiris**: Funciona igual que Oyente puesto que está basada en ella.

- **EthIr**: Esta herramienta posee una dependencia similar con la librería *Z3 Prover*. El problema es que la coexistencia de las anteriores herramientas con EthIr puesto que las versiones de la librería que soportan son diferentes. Este es uno de los motivos principales de escoger los contenedores para desarrollar la herramienta. Además, la última versión testada del compilador fue la 0.4.25, siendo la última la 0.6.

- **Vandal**: Dados los requerimientos para la instalación de la herramienta no presenta una dificultad grande salvo su propio uso. Vandal es una herramienta para usuarios avanzados no recomendable para desarrolladores nóveles.

- **Mythril**: Dentro de las herramientas que han dado problemas a la hora de la instalación de forma nativa Mythril ha sido una de ellas.

  Por otro lado el contenedor que proporcionan poseen ciertas limitaciones para su uso debido a la configuración del archivo *Dockerfile* que puede verse en la Figura

4.7. La herramienta falla debido a la versión del compilador (0.5.1). Para contratos antiguos (versiones del compilador como la 0.4.24) es necesario introducir la versión del contrato para que la herramienta pudiera descargarse el correspondiente.

Además, para poder ejecutar la herramienta desde *Python* se ha eliminado el *Entrypoint* fijado por los desarrolladores. Esta variable genera que el comando introducido deja de ser de tipo *bash* para ser solo introducibles comandos que entienda Mythril.

```
FROM ubuntu:bionic

# Space-separated version string without leading 'v' (e.g. "0.4.21 0.4.22")
ARG SOLC

RUN apt-get update \
  && apt-get install -y \
     libsqlite3-0 \
     libsqlite3-dev \
  && apt-get install -y \
     build-essential \
     locales \
     python-pip-whl=9.0.1-2 \
     python3-pip=9.0.1-2 \
     python3-setuptools \
     software-properties-common \
  && add-apt-repository -y ppa:ethereum/ethereum \
  && apt-get update \
  && apt-get install -y \
     solc \
     libssl-dev \
     python3-dev \
     pandoc \
     git \
     wget \
  && ln -s /usr/bin/python3 /usr/local/bin/python

COPY ./requirements.txt /opt/mythril/requirements.txt

RUN cd /opt/mythril \
  && pip3 install -r requirements.txt

RUN locale-gen en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US.en
ENV LC_ALL en_US.UTF-8

COPY . /opt/mythril
RUN cd /opt/mythril \
  && python setup.py install

WORKDIR /home/mythril

RUN ( [ ! -z "${SOLC}" ] && set -e && for ver in $SOLC; do python -m solc.install v${ver}; done ) || true

COPY ./mythril/support/assets/signatures.db /home/mythril/.mythril/signatures.db

ENTRYPOINT ["/usr/local/bin/myth"]
```

Figura 4.7: Dockerfile sin modificar.

- **Securify**: Securify en su primera versión presentó bastantes problemas. Dado que actualmente se maneja la versión 2 que es más estable, no se ha encontrado en esta una gran dificultad si se siguen las instrucciones.
- **Slither**: Al igual que Mythril se ha modificado el archivo *Dockerfile* para poder eliminar ciertas restricciones. Entre ellas se ha eliminado un usuario que creaba por defecto el contenedor e impedía poder hacer uso de los volúmenes para la lectura de los contratos.
- **Manticore**: Hay que especificar todas las versiones del compilador con las que tra-

baja. En caso de que no se especifique aquella que posea el contrato inteligente dará un error y no se podrá analizar. Presenta limitaciones también con el número de versiones puesto que no trabaja con todas.

- **Madmax**: Ninguna dificultad.
- **Contract Larva**: Pese a no estar incluida dentro de la herramienta, se ha considerado interesante incluirla puesto que es una herramienta muy útil. El principal problema que presenta es la instalación de Haskell dado que el instalador por defecto instala una versión que no es la correcta. Para ello se aconseja usar:

```
$curl https://get-ghcup.haskell.org -sSf | sh
```

4.6. Pruebas de la meta-herramienta

Tras el proceso de instalación de los contenedores, creación de la base de datos y clonación del repositorio podremos ejecutar la herramienta. Para la prueba por línea de comandos se hará uso del análisis unitario, en concreto del contrato número 100 en la base de datos. Tras ejecutar el comando:

```
app.py -t -i 100
```

La herramienta proporcionará una serie de información en el archivo *stack_trace.log*. En este caso dado que el contrato solo tiene *bytecode* solo analizará aquellas herramientas diseñadas para ello. Tras realizar el análisis dicho archivo tiene la siguiente apariencia.

```
25/06/2020 10:13:37 PM Execution begins
25/06/2020 10:13:37 PM Current working directory set to /Users/Ture/Documents/TFG
25/06/2020 10:13:38 PM Unique contract test selected. Contract number 100
25/06/2020 10:13:49 PM Analyzing contract number 100
25/06/2020 10:13:49 PM Running oyente ...
25/06/2020 10:13:56 PM Completed. Saving results...
25/06/2020 10:13:56 PM Completed.
25/06/2020 10:13:56 PM Running osiris ...
25/06/2020 10:14:47 PM Completed. Saving results...
25/06/2020 10:14:47 PM Completed.
25/06/2020 10:14:47 PM Running ethir ...
25/06/2020 10:14:48 PM Completed. Saving results...
25/06/2020 10:14:48 PM Completed.
25/06/2020 10:14:48 PM Running vandal ...
25/06/2020 10:14:54 PM Completed. Saving results...
25/06/2020 10:14:54 PM Completed.
25/06/2020 10:14:54 PM Running madmax ...
25/06/2020 10:15:04 PM Completed. Saving results...
25/06/2020 10:15:04 PM Completed.
25/06/2020 10:15:04 PM 100 contract analyze ended.
```

Dado que no se ha mostrado ningún tipo de error, a priori la herramienta no ha dado ningún fallo y el resultado ha sido satisfactorio. Para obtener los resultados que se ven el el Código 4.12 ejecutamos una consulta con *mongodb*

Código 4.12: Resultado del análisis para el contrato 100


```
{
{
   "_id":"ObjectId("   "5e9c72dc574ce3bef34da3d9"   ")",
   "contract_id":21985,
   "address":"0xfdbbc5346605ed3b1799fd059e2f7cd9bed4f49a",
   "solgraph":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "smartcheck":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "solmet":{
      "output":"Contract extension doesn't allow this analysis",
      "time_elapsed":"0"
   },
   "oyente":{
      "output":"WARNING:root:You are using evm version 1.8.2. The supported
         ↪  version is 1.7.3\n\nWARNING:root:You are using solc version 0
         ↪ .4.21, The latest supported version is 0.4.19\n\nINFO:symExec:
         ↪ \t============ Results ===========\n\nINFO:symExec:\t  EVM
         ↪ Code Coverage: \t\t\t 63.3%\n\nINFO:symExec:\t  Callstack
         ↪ Depth Attack Vulnerability:  False\n\nINFO:symExec:\t
         ↪ Transaction-Ordering Dependence (TOD): False\n\nINFO:symExec:\
         ↪ t  Timestamp Dependency: \t\t False\n\nINFO:symExec:\t  Re-
         ↪ Entrancy Vulnerability: \t\t False\n\nINFO:symExec:\t======
         ↪ Analysis Completed ======\n\n",
      "time_elapsed":0.023337632999997027
   },
   "osiris":{
      "output":"INFO:symExec:Running, please wait...\n\nINFO:symExec:\t
         ↪ ============ Results ===========\n\nINFO:symExec:\t  EVM code
         ↪ coverage: \t 39.8%\n\nINFO:symExec:\t  Arithmetic bugs: \t
         ↪ False\n\nINFO:symExec:\t     > Overflow bugs: \t False\n\nINFO
         ↪ :symExec:\t     > Underflow bugs: \t False\nINFO:symExec:\t
         ↪     > Division bugs: \t False\nINFO:symExec:\t     > Modulo
         ↪ bugs:   \t False\nINFO:symExec:\t     > Truncation bugs: \t
         ↪ False\nINFO:symExec:\t     > Signedness bugs: \t False\n\nINFO
         ↪ :symExec:\t  Callstack bug: \t False\nINFO:symExec:\t
         ↪ Concurrency bug: \t False\n\nINFO:symExec:\t  Timedependency
         ↪ bug: \t False\n\nINFO:symExec:\t  Reentrancy bug: \t False\
         ↪ nINFO:symExec:\t --- 50.5489561558 seconds ---\nINFO:symExec:\
         ↪ t====== Analysis Completed ======\n\n\n",
      "time_elapsed":0.008884562999995183
   },
   "ethir":{
      "output":"Traceback (most recent call last):\n  File \"oyente-ethir.
         ↪ py\", line 499, in <module>\n\n
         ↪ ***********************************************************\
         ↪ nCompilation time: 0.238485097885s\n
         ↪ ***********************************************************\
         ↪ n\n    main()\n  File \"oyente-ethir.py\", line 481, in main\n
         ↪     exit_code = analyze_bytecode()\n  File \"oyente-ethir.py\"
         ↪ , line 173, in analyze_bytecode\n    result, exit_code =
         ↪ symExec.run(disasm_file=inp[\\'disasm_file\\'],cfg = args.
         ↪ control_flow_graph,saco = args.saco,debug = args.debug,
         ↪ evm_version = evm_version_modifications,cfile = args.cfile,svc
         ↪ =svc_options,go = args.goto)\n  File \"/home/EthIR/ethir/
```

```
            ↪ symExec.py\", line 3220, in run\n\n    analyze(evm_version)\n
            ↪  File \"/home/EthIR/ethir/symExec.py\", line 3013, in analyze\
            ↪ n\n    run_build_cfg_and_analyze(evm_v = evm_version,
            ↪ timeout_cb=timeout_cb)\n  File \"/home/EthIR/ethir/symExec.py
            ↪ \", line 2965, in run_build_cfg_and_analyze\n\n
            ↪ build_cfg_and_analyze(evm_v)\n  File \"/home/EthIR/ethir/
            ↪ symExec.py\", line 304, in build_cfg_and_analyze\n
            ↪ full_sym_exec()  # jump targets are constructed on the fly\n
            ↪ File \"/home/EthIR/ethir/symExec.py\", line 766, in
            ↪ full_sym_exec\n\n    return sym_exec_block(params, 0, 0, 0, -1
            ↪ , 0,[(0,0)])\n  File \"/home/EthIR/ethir/symExec.py\", line 10
            ↪ 52, in sym_exec_block\n\n    sym_exec_block(new_params,
            ↪ left_branch, block, depth, func_call,current_level+1,path)\n
            ↪ File \"/home/EthIR/ethir/symExec.py\", line 1052, in
            ↪ sym_exec_block\n\n    sym_exec_block(new_params, left_branch,
            ↪ block, depth, func_call,current_level+1,path)\n  File \"/home/
            ↪ EthIR/ethir/symExec.py\", line 1052, in sym_exec_block\n\n
            ↪ sym_exec_block(new_params, left_branch, block, depth,
            ↪ func_call,current_level+1,path)\n  File \"/home/EthIR/ethir/
            ↪ symExec.py\", line 963, in sym_exec_block\n\n
            ↪ sym_exec_block(new_params, successor, block, depth, func_call,
            ↪ current_level+1,path)\n  File \"/home/EthIR/ethir/symExec.py\"
            ↪ , line 1009, in sym_exec_block\n\n    sym_exec_block(
            ↪ new_params, successor, block, depth, func_call,current_level+1
            ↪ ,path)\n  File \"/home/EthIR/ethir/symExec.py\", line 1052, in
            ↪  sym_exec_block\n\n    sym_exec_block(new_params, left_branch,
            ↪  block, depth, func_call,current_level+1,path)\n  File \"/home
            ↪ /EthIR/ethir/symExec.py\", line 963, in sym_exec_block\n\n
            ↪ sym_exec_block(new_params, successor, block, depth, func_call,
            ↪ current_level+1,path)\n  File \"/home/EthIR/ethir/symExec.py\"
            ↪ , line 1112, in sym_exec_block\n\n    sym_exec_block(
            ↪ new_params, right_branch, block, depth, func_call,
            ↪ current_level+1,path)\n  File \"/home/EthIR/ethir/symExec.py\"
            ↪ , line 1009, in sym_exec_block\n\n    sym_exec_block(
            ↪ new_params, successor, block, depth, func_call,current_level+1
            ↪ ,path)\n  File \"/home/EthIR/ethir/symExec.py\", line 1112, in
            ↪  sym_exec_block\n\n    sym_exec_block(new_params, right_branch
            ↪ , block, depth, func_call,current_level+1,path)\n  File \"/
            ↪ home/EthIR/ethir/symExec.py\", line 1015, in sym_exec_block\n
            ↪    update_depth_level(successor,current_level+1,[])\n\n  File
            ↪ \"/home/EthIR/ethir/symExec.py\", line 2602, in
            ↪ update_depth_level\n\n    update_depth_level(falls,level+1,
            ↪ updated)\n  File \"/home/EthIR/ethir/symExec.py\", line 2593,
            ↪ in update_depth_level\n\n    if jump != 0 and vertices[jump].
            ↪ get_depth_level()<(level+1):\nKeyError: 2\n\ncat: rbr.rbr: No
            ↪ such file or directory\n\n",
28      "time_elapsed":0.025814652999997634
29   },
30   "vandal":{
31      "output":"+++ dirname ./analyze.sh\n\n++ cd .\n\n++ pwd\n\n+ DIR=/
            ↪ home/vandal/bin\n+ rm -rf facts-tmp\n\n+ /home/vandal/bin/
            ↪ decompile -o CALL JUMPI SSTORE SLOAD MLOAD MSTORE -d -n -t
            ↪ facts-tmp /tmp/inputs/input_contract_21985.hex\n\n+ souffle -F
            ↪  facts-tmp ../datalog/demo_analyses.dl\n\nWarning: Deprecated
            ↪ type declaration used in \nfile types.dl at line 30\n.type
            ↪ Statement\n^---------------\nWarning: Deprecated type
            ↪ declaration used in \nfile types.dl at line 31\n.type Variable
            ↪ \n^--------------\nWarning: \nDeprecated type declaration used
            ↪ \n in \nfile types.dl at line 32\n.type Opcode\n^------------\
            ↪ nWarning: Deprecated type declaration used\n in \nfile types.
            ↪ dl at line 33\n.type Value\n^-----------\n\n+ rm -rf facts-tmp
```

```
            ↪ \n\n",
32          "time_elapsed":0.03626212099999293
33       },
34       "mythril":{
35          "output":"Contract extension doesn't allow this analysis",
36          "time_elapsed":"0"
37       },
38       "securify":{
39          "output":"Contract extension doesn't allow this analysis",
40          "time_elapsed":"0"
41       },
42       "slither":{
43          "output":"Contract extension doesn't allow this analysis",
44          "time_elapsed":"0"
45       },
46       "manticore":{
47          "output":"Contract extension doesn't allow this analysis",
48          "time_elapsed":"0"
49       },
50       "madmax":{
51          "output":"Se ha omitido debido al gran tama o generado",
52          "time_elapsed": "0.06253403499999877"
53       },
54   }
55   }
```


Si nos fijamos, tanto Oyente como Osiris no han encontrado ningún tipo de fallo relacionado con las vulnerabilidades que ellas manejan. También es cierto que el porcentaje de la EVM que ha cubierto Vandal en relación al contrato has ido del 63.3 %. En cambio la herramientas como Vandal y EthIr están dando avisos de posibles flujos no deseados de código junto con malas prácticas. El resto de herramientas no han reportado análisis.

Con la posibilidad de hacer análisis sobre contratos directamente descargados de la red de Ethereum, el siguiente análisis se realiza sobre la dirección

`0x469503159DdF6Bfd0a9ec8Eba8e97A84Fd3eaE5b`

Como herramientas has sido seleccionadas: Solmet, Osiris, Vandal, Securify y Slither. Los resultados obtenidos pueden verse en las Figuras 4.8, 4.9 y 4.10.

```
Herramienta: solmet Resultado: Parsing /tmp/inputs/input_contract_0x469503159DdF6Bfd0a9ec8Eba8e97A84Fd3eaE5b.sol

SolidityFile;ETHAddress;ContractName;Type;SLOC;LLOC;CLOC;NF;WMC;NL;NLE;NUMPAR;NOS;DIT;NOA;NOD;CBO;NA;NOI;Avg. McCC;Avg. NL;Avg. NLE;Avg. NUMPAR;Avg. NOS;Avg. NOI

input_contract_0x469503159DdF6Bfd0a9ec8Eba8e97A84Fd3eaE5b.sol;inputs;BulkSender;contract;18;15;0;2;3;1;1;2;8;0;0;0;0;0;4;1.5;0.5;0.5;1.0;4.0;2.0


 Tiempo ejecución: 0.021090371999999746
-----------------------!!!-----------------------


Herramienta: osiris Resultado: INFO:symExec:Running, please wait...

INFO:symExec: ============ Results ===========

INFO:symExec:   EVM code coverage:   14.9%

INFO:symExec:   Arithmetic bugs:   False

INFO:symExec:   └> Overflow bugs:   False

INFO:symExec:   └> Underflow bugs:   False

INFO:symExec:   └> Division bugs:   False

INFO:symExec:   └> Modulo bugs:     False

INFO:symExec:   └> Truncation bugs:   False

INFO:symExec:   └> Signedness bugs:   False

INFO:symExec:   Callstack bug:   False

INFO:symExec:   Concurrency bug:   False

INFO:symExec:   Timedependency bug:   False

INFO:symExec:   Reentrancy bug:   False

INFO:symExec:  --- 0.14600610733 seconds ---

INFO:symExec: ====== Analysis Completed ======


  .oooooo.             o8o            o8o
 d8P'  `Y8b            `"'            `"'
888      888  .oooo.o oooo  oooo d8b oooo   .oooo.o
888      888 d88(  "8 `888  `888""8P `888  d88(  "8
888      888 `"Y88b.   888   888      888  `"Y88b.
`88b    d88' o.  )88b  888   888      888  o.  )88b
 `Y8bood8P'  8""888P' o888o d888b    o888o 8""888P'

UNKNOWN INSTRUCTION: SHR
```

Figura 4.8: Primera parte del análisis.

```
Herramienta: vandal Resultado: +++ dirname ./analyze.sh

++ cd .

++ pwd

+ DIR=/home/vandal/bin
+ rm -rf facts-tmp

+ /home/vandal/bin/decompile -o CALL JUMPI SSTORE SLOAD MLOAD MSTORE -d -n -t facts-tmp /tmp/inputs/input_contract_0x469503159DdF6Bfd0a9ec8Eba8e97A84Fd3eaE5b.hex

+ souffle -F facts-tmp ../datalog/demo_analyses.dl

Warning: Deprecated type declaration used in
file types.dl at line 30
.type Statement
^~~~~~~~~~~~~~
Warning: Deprecated type declaration used in file types.dl at line 31
.type Variable
^~~~~~~~~~~~~
Warning: Deprecated type declaration used in
file types.dl at line 32
.type Opcode
^~~~~~~~~~~
Warning: Deprecated type declaration used in file types.dl at line 33
.type Value
^~~~~~~~~~

+ rm -rf facts-tmp

 Tiempo ejecución: 0.012614247999998844
-----------------------!!!-----------------------


Herramienta: securify Resultado: pragma directive defines a prior version to 0.5.12. Changing pragma version to 0.5.12....
[0;31mSeverity:    HIGH
Pattern:     External call in loop
Description: If a single call in the loop fails or revers, it will
             cause all other calls to fail as well.
Type:        Violation
Contract:    BulkSender
Line:        14
Source:
>             require(value > 0);
>             address(uint160(_to)).transfer(value);
>             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>         }

[0m
[0;31mSeverity:    INFO
Pattern:     Solidity Naming Convention
Description: Reports declarations that do not adhere to Solidity's
             naming convention.
```

Figura 4.9: Segunda parte del análisis.

```
[0m
[0;31mSeverity:    CRITICAL
Pattern:     Transaction Order Affects Ether Amount
Description: The amount of ether transferred must not be influenced by
             other transactions.
Type:        Violation
Contract:    BulkSender
Line:        19
Source:
>     function() external payable {
>         msg.sender.transfer(address(this).balance);
>         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>     }

[0m
[0;31mSeverity:    INFO
Pattern:     Uninitialized Local Variables
Description: A variable is declared but never initialized.
Type:        Violation
Contract:    BulkSender
Line:        10
Source:
>
>         for (uint256 i; i < addresses.length; i++) {
>                                                 ^
>             uint256 value = amounts[i];

[0m
[0;31mSeverity:    CRITICAL
Pattern:     Unrestricted Ether Flow
Description: The execution of ether flows should be restricted to an
             authorized set of users.
Type:        Violation
Contract:    BulkSender
Line:        14
Source:
>             require(value > 0);
>             address(uint160(_to)).transfer(value);
>             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
>         }

[0m
[0;31mSeverity:    CRITICAL
Pattern:     Unrestricted Ether Flow
Description: The execution of ether flows should be restricted to an
             authorized set of users.
Type:        Violation
Contract:    BulkSender
```

Figura 4.10: Tercera parte del análisis.

Como se puede comprobar, los resultados han sido satisfactorios para las herramientas de flujo como son Securify y Vandal. Solmet proporciona las métricas del contrato y Osiris no ha encontrado ninguna vulnerabilidad.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

Del uso de la herramienta creada durante la realización de este trabajo de fin de grado se puede sacar una visión muy amplia y general de la situación en cuanto a seguridad de la red en lo que se refiere. Ethereum es una cadena de bloques realmente segura, de por si el protocolo protege al propio usuario pero un uso incorrecto de los contratos o de las direcciones virtuales puede desencadenar errores fatales.

Durante los diferentes artículos analizados y lecturas se ha visto una falta de información clara de cuál es el estado actual de la red, cuántos contratos hay vulnerables, si dichos contratos han sido o no destruidos y cuales de ellos siguen hoy en día funcionando. La herramienta provee una introducción muy importante para aprovechar los análisis de contratos para fomentar la búsqueda de tendencias, número de vulnerabilidades totales, qué vulnerabilidad es la más cometida y si hoy en día sigue habiendo fallos de ese tipo.

Las diferentes herramientas utilizadas presentan ventajas unas frente a la otras, pero por norma general, debido a las continuas modificaciones del compilador de Solidity hay un gran acoplamiento a la versión con la cual sea desarrollada la herramienta. En el año en el que estamos es un gran paso atrás, el funcionamiento de una herramienta de seguridad no puede estar ligada a la versión del compilador. Cuantos más modificaciones se hagan en el código primitivo (dado que aún no ha llegado a la versión 1.0) más problemas habrá con las herramientas que dependan de ella, comprometiendo la seguridad.

Las herramientas frente a trabajos de alta carga (diez mil análisis a la vez) han presentado un rendimiento óptimo pese a no tener la paralelización implementada. De añadir la paralelización y aumentar el número de contenedores y recursos, la velocidad de análisis crecería enormemente. Los análisis arrojados durante las pruebas eran satisfactorios, encontrase vulnerabilidades en la mayoría de contratos. Esto es así porque la primera parte de la base de datos de contratos inteligentes descargada presentaban versiones del compilador antiguas.

En cuanto a la efectividad de análisis masivos la red presenta ciertas limitaciones. No se puede descargar de forma sencilla una gran número de contratos. La principal razón es el alto coste de la sincronización del nodo con la red y la alta demanda de información. Debido al volumen de datos la descarga de contratos está limitada, seguramente como medida de seguridad.

Por todo lo descrito anteriormente se concluye que la red ha avanzado cada vez más a una seguridad mayor (estrechamente ligada a una adopción mayor). Ya no se producen fallos tan grandes como el Ataque DAO 3.2.2. Las herramientas actualizadas cada vez son menores primando aquellas que han tenido desarrollos más serios como son Securify2, Slither O Remix IDE. Los usuarios que quieran comenzar a desarrollar contratos inteligentes y desplegarlos necesitarán tener un conocimiento, al menos general, de los fallos que posee la versión actual del código con la que trabajen así cómo evitar dichas vulnerabilidades.

5.2. Trabajo Futuro

La meta-herramienta desarrollada para este trabajo es completamente funcional aunque como todo software es susceptible de mejoras y ampliación de funcionalidad. Es muy probable que sigan apareciendo nuevas herramientas de análisis de vulnerabilidades en contratos inteligentes que podrían ser añadidas a las ya actuales, además de otras herramientas que se actualicen o dejen de existir. La interfaz gráfica, ahora mismo en una versión inicial también se podría ampliar con opciones de análisis por rangos temporales por ejemplo.

Por otro lado, hay un artículo a punto de ser enviado a una revista científica sobre la meta-herramienta y otro trabajo en progreso para el desarrollo de una herramienta de análisis de vulnerabilidades en Ethereum usando algoritmos de aprendizaje automático (inspirada en [WSX+20]) donde se usará la meta-herramienta aquí presentada en la fase de etiquetado de contratos vulnerables, con lo que se pretende conseguir abarcar un mayor número de vulnerabilidades.

Capítulo 6

Introduction

6.1. Motivation

The short period of time since the creation of Blockchain, 12 years, together with the recent expansion of the technological paradigm has led to a stream of new systems based on its architecture. In 2020 it is positioned among the most promising technologies together with artificial intelligence, automatic learning, macro-data or 5G among others.

The qualities that make it special (decentralization, interchangeability and security) create new businesses, jobs and solutions that make it a great emerging market niche for technology companies. The different solutions that have been adopted for its implementation are very varied since their applicability is also giving rise to new protocols different from Bitcoin, specifically Ethereum which provides the ability to program and introduce such code to the network so that external applications can interact with the block chain, automate processes and exchange information with external agents.

Ethereum, as well as the Blockchain, is a very modern technology. After the first proof of concept, other tests were carried out in order to make the network more efficient with the creation of, for example, the language that today handles the network, Solidity. The open source project has had to face a growing adoption. Right now it is immersed in a complex evolutionary process called Ethereum 2.0. This new version of the network brings very significant changes, it will leave behind its current consensus protocol, it will allow the use of and it will improve the transaction rate per second which is one of the main current problems compared to other networks.

This growth and updating of the network is also accompanied by the adoption of Solidity. The current language version is 0.6, which is still less than 1. Solidity is still subject to changes that can naturally cause vulnerabilities due to the use of wrong programming flows, use of directives in points where they should not be used, wrong assignments or excessive use of gas. Therefore, the importance of having tools that help programmers to cope with their developments is fundamental. Over time, different tools have emerged, all of them without a common thread or a central point where to carry out a single analysis that guarantees to pass some tests for all the found vulnerabilities. Furthermore, no study has been carried out that has obtained positive results of the current situation of the contracts deployed on the network, number of failures per contract, failure rates, most common errors, percentage of failure reduction depending on the version of Solidity, etc.

6.2. Context

This Final Degree Project has been carried out within The Group of Analysis, Security and Systems (GASS) (https://gass.ucm.es/, Group 910623 of the catalogue of groups recognized by the UCM) as part of the activities of the research project THEIA (Techniques for Integrity and Authentication of Multimedia Files of Mobile Devices) with reference FEI-EU-19-04.

6.3. Object of the Investigation

With the rapid growth of Solidity within the Ethereum network there is a need for tools that provide all the analysis potential in an easy way for a programmer who wants to analyze his contract. Right now the information that exists is dispersed, and for a novice programmer it can be very complex to get started in the world of intelligent contract security. That is why it is necessary to have a single place where to synthesize all the studies that have been created separately, giving quick answers to which are the failures that can be found and which are the tools that can detect them. When we talk about tools we are talking about both a technical document and a computer tool that guides the user in the analysis process.

6.4. Work Plan

The development of this end-of-degree work has been carried out in three phases:

1. **Learning**: The main motivation for the selection of Blockchain as an end-of-grade work came as a reading from a book that talked about the birth of Bitcoin, Ethereum and Ethereum 2.0 and all the benefits that technology provides. From that moment on, with the first meeting, we looked for approximately 4 months to learn everything possible about Ethereum. For this purpose, an online course was made in the Coursera platform called 'Blockchain Basics' by the University of Buffalo, as well as the part two of the program called 'Smart Contracts'. In addition, as complementary training, we looked more specifically for the different aspects that make up the project, basic elements of security, different systems of consensus, use of hash... Finally, it was decided to unite the aspect of computer security together with the chain of blocks, considering that a very solid and complete work could come out.

2. **study and experimentation of tools**: After the election began the process of studying what was the exact situation in the aspect of security. For this purpose, different studies that have already been published were read, which have served as the basis for the realization of this project. These studies are based on the vulnerabilities found, serious historical flaws in Ethereum such as the "DAO Attack.ºr the reentrancy vulnerability and the tools that are currently available. As for the tools, they focus more on the study below each one, i.e. how they work, but do not provide a clarification of whether or not their use is worthwhile. After sufficient research to understand what the current context was, a full study was made of the installation, commands, supported vulnerabilities, and a number of points of each tool. Thanks to this process a paper was published of which I am co-author [LVCSOGV20].

3. **Development**: After publication the last step was the creation of a tool that would simplify the developer's individual analysis. For this purpose, a solution based on containers and their modularization was proposed. The container management system has been *Docker* and to be able to interact with each container the *Python* library and the Docker for Python SDK have been used. During this period many technical obstacles have arisen due to the dependencies and developments of each of the creators. The final solution has always been conditioned by the limitations that the tools offered (compiler versions, file extension management, formats, Solidity code versions). Finally, for a more simplified use it was decided to add a small interface to be able to analyze local contracts and contracts already deployed in Ethereum, in addition to the corresponding contracts already inserted in the database.

6.5. Structure of the Work

The rest of the work is organized in 5 chapters and an annex with the structure that is commented on below:

Chapter 2 introduces some concepts that are elementary to understanding the operation of the block chain and intelligent contracts. Thus, the cryptocurrencies, the Bitcoin, the digital wallets, the transactions, the outputs of the transactions, the blocks, the mining process, the verification of the blocks, the types of blockhains are described and the Ethereum is presented, with its characteristics, the intelligent contracts and the virtual machine of Ethereum.

Chapter 3 begins with an introduction to general Blockchain mechanisms and technologies, and then focuses on Ethereum and Smart Contracts security. It discusses the security principles of Smart Contracts, the most famous Ethereum security incidents, and the different types of vulnerabilities, differentiating them according to whether they affect Blockchain, the Smart Contracts, or the Ethereum virtual machine. Finally, chapter ends with a review of the vulnerability analysis tools in the Ethereum, first introducing the analysis methods used by the tools and then describing each tool individually with a final comparison.

The Chapter 4 presents the main contribution of this work, namely the *Meta Tool for Analyzing Vulnerabilities in Smart Contracts*, starting with a justification and the scope of the developed tool, going through the environment and technologies used, the functioning of the tool, implementation details and difficulties and limitations among other sections.

Finally, the main conclusions of this work as well as future lines of work are presented in Chapter 5.

Capítulo 7

Conclusions and Future Work

From the use of the tool created during the completion of this end-of-degree work, a very broad and general vision of the situation regarding network security can be obtained. Ethereum is a really secure blockchain, in case the protocol protects the user himself but a wrong use of the contracts or the virtual addresses can trigger fatal errors.

During the different articles analyzed and readings we have seen a lack of clear information about what the current state of the network is, how many contracts are vulnerable, whether or not those contracts have been destroyed and which ones are still working today. The tool provides a very important introduction to take advantage of the contract analysis to encourage the search for trends, number of total vulnerabilities, which vulnerability is the most committed and whether such failures still exist today.

The different tools used have advantages over each other, but generally, due to the continuous modifications of the Solidity compiler there is a great coupling to the version with which the tool is developed. In the year we are in is a big step backwards, the operation of a security tool cannot be linked to the compiler version. The more modifications made to the primitive code (since it has not yet reached version 1.0) the more problems there will be with the tools that depend on it, compromising security.

The tools against high load jobs (ten thousand analysis at a time) have presented an optimal performance in spite of not having the parallelization implemented. If parallelization is added and the number of containers and resources is increased, the speed of analysis will increase enormously. The scans thrown during the tests were satisfactory; vulnerabilities were found in most of the contracts. This is because the first part of the downloaded smart contract database contained older versions of the compiler.

As for the effectiveness of mass scans, the network has certain limitations. A large number of contracts cannot be easily downloaded. The main reason is the high cost of node synchronization with the network and the high demand for information. Due to the volume of data the download of contracts is limited, probably as a security measure.

For all the above reasons, it can be concluded that the network has advanced more and more to a higher security (closely linked to a higher adoption). There are no longer failures as big as the ”DAO Attack”3.2.2. The updated tools are less and less, giving priority to those that have had more serious developments such as Securify2, Slither or Remix IDE. Users who want to start developing smart contracts and deploying them will need to have at least a general knowledge of the bugs in the current version of the code they are working with so that they can avoid such vulnerabilities.

7.1. Future Work

The meta-tool developed for this work is fully functional, although like all software, it can be improved and its functionality extended. It is very likely that new tools for vulnerability analysis in smart contracts will continue to appear and could be added to the current ones, in addition to other tools that will be updated or cease to exist. The graphical interface, right now in an initial version could also be extended with options for analysis by time ranges for example.

On the other hand, there is an article about to be sent to a scientific journal about the meta-tool and another work in progress for the development of a vulnerability analysis tool in Ethereum using machine learning algorithms (inspired by [WSX+20]) where the meta-tool presented here will be used in the phase of tagging vulnerable contracts, in order to cover a greater number of vulnerabilities.

Apéndice A

Funcionamiento de las Herramientas

A.1. Slither

Cada comando esta basado en la librería *crytic-compile*, desarrollada por los creadores de Slither.

Por defecto, el último compilador de Solidity acepta contratos desde la versión 0.6, por lo tanto es importante disponer de una forma sencilla de cambiar de versión, que ya integra Slither.

Para la ejemplificación de un análisis, se ha analizado un contrato con la vulnerabilidad de variables sin inicializar. La forma en que tiene Slither de mostrar el resultado se puede ver en la Figura A.1 Como se puede comprobar, no se puede ver de forma sencilla los resultados, aunque si que se puede ver como la opción *INFO:Detectors* muestra las líneas con los errores específicos de variables sin inicializar.

Para verlo de forma más clara se le puede añadir el flag *print humman-summary* como se muestra en la Figura A.2

A.2. Oyente

Oyente tiene algunas opciones de configuración pasadas por línea de comandos que puedes obtenerse mediante el uso de la opción *–list*. Algunas de las más importantes son las referentes al límite de gas, el límite de bucles, límite de profundidad, límite de tiempo, etc.

Todas estas opciones permiten personalizar los análisis de gran manera. Si se corre la herramienta con un contrato de ejemplo, Oyente nos dará los resultados de la siguiente forma como puede verse en la Figura A.3

```
ubuntu@smartcontracts:~/ALBERTO/slither/slither/tests$ slither uninitialized.sol
Compilation warnings/errors on uninitialized.sol:
uninitialized.sol:19:5: Warning: No visibility specified. Defaulting to "public".
    function init() {
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:23:5: Warning: No visibility specified. Defaulting to "public".
    function use() {
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:35:5: Warning: No visibility specified. Defaulting to "public".
    function set(MyStruct storage st, uint v){
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:35:39: Warning: Unused function parameter. Remove or comment out the variable name to silence this warning.
    function set(MyStruct storage st, uint v){
                                      ^----^
uninitialized.sol:49:5: Warning: No visibility specified. Defaulting to "public".
    function init(){
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:53:5: Warning: No visibility specified. Defaulting to "public".
    function use(){
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:23:5: Warning: Function state mutability can be restricted to view
    function use() {
    ^ (Relevant source part starts here and spans across multiple lines).
uninitialized.sol:53:5: Warning: Function state mutability can be restricted to view
    function use(){
    ^ (Relevant source part starts here and spans across multiple lines).

INFO:Detectors:
Uninitialized.destination (uninitialized.sol#5) is never initialized. It is used in:
        - Uninitialized.transfer() (uninitialized.sol#7-9)
Test.balances (uninitialized.sol#15) is never initialized. It is used in:
        - Test.use() (uninitialized.sol#23-26)
Test2.st (uninitialized.sol#45) is never initialized. It is used in:
        - Test2.use() (uninitialized.sol#53-56)
Test2.v (uninitialized.sol#47) is never initialized. It is used in:
        - Test2.init() (uninitialized.sol#49-51)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
Pragma version^0.4.24 (uninitialized.sol#1) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Test2.v (uninitialized.sol#47) should be constant
Uninitialized.destination (uninitialized.sol#5) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
transfer() should be declared external:
        - Uninitialized.transfer() (uninitialized.sol#7-9)
init() should be declared external:
        - Test.init() (uninitialized.sol#19-21)
use() should be declared external:
        - Test.use() (uninitialized.sol#23-26)
init() should be declared external:
        - Test2.init() (uninitialized.sol#49-51)
use() should be declared external:
        - Test2.use() (uninitialized.sol#53-56)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-as-external
INFO:Slither:uninitialized.sol analyzed (4 contracts with 40 detectors), 12 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Figura A.1: Ejemplo de un análisis con Slither

```
INFO:Printers:
Compiled with 1
Number of lines: 58 (+ 0 in dependencies)
Number of contracts: 4 (+ 0 in dependencies)

Number of optimization issues: 7
Number of informational issues: 1
Number of low issues: 0
Number of medium issues: 0
Number of high issues: 4

Contract Uninitialized
        Complex code? No
        Number of functions: 1

Contract Test
        Complex code? No
        Number of functions: 2

Contract Lib
        Complex code? No
        Number of functions: 1

Contract Test2
        Complex code? No
        Number of functions: 2
```

Figura A.2: Uso del flag *print human-summary*

```
ubuntu@smartcontracts:~/ALBERTO/oyente$ sudo docker run -i -t luongnguyen/oyente
root@e833e8ab3246:/oyente# cd /oyente/oyente && python oyente.py -s greeter.sol
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.21, The latest supported version is 0.4.19
INFO:root:contract greeter.sol:greeter:
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                    99.5%
INFO:symExec:     Integer Underflow:                    False
INFO:symExec:     Integer Overflow:                     False
INFO:symExec:     Parity Multisig Bug 2:                False
INFO:symExec:     Callstack Depth Attack Vulnerability: False
INFO:symExec:     Transaction-Ordering Dependence (TOD): False
INFO:symExec:     Timestamp Dependency:                 False
INFO:symExec:     Re-Entrancy Vulnerability:            False
INFO:symExec:   ====== Analysis Completed ======
```

Figura A.3: Ejemplo de análisis con Oyente

En dicho análisis no muestra ningún positivo, lo cual es bueno, el contrato analizado está libre de las vulnerabilidades buscadas, además de haber cubierto el 99.5 % del código de la EVM.

A.3. Remix - IDE

Puesto que esta herramienta no permite la instalación en un entorno aislado como puede ser un contenedor, se va a proceder a introducir los puntos fuertes en cuando a seguridad que ofrece Remix-IDE. Como puede verse en la Figura A.4 en el menú lateral, en la opción número cuatro se puede encontrar el *Analizador de código estático*, esta opción da la posibilidad de seleccionar diferentes opciones de seguridad, algunas de ellas son:

- Reestrancy bug.
- Low level calls (llamadas de bajo nivel).
- Origen de transacciones.

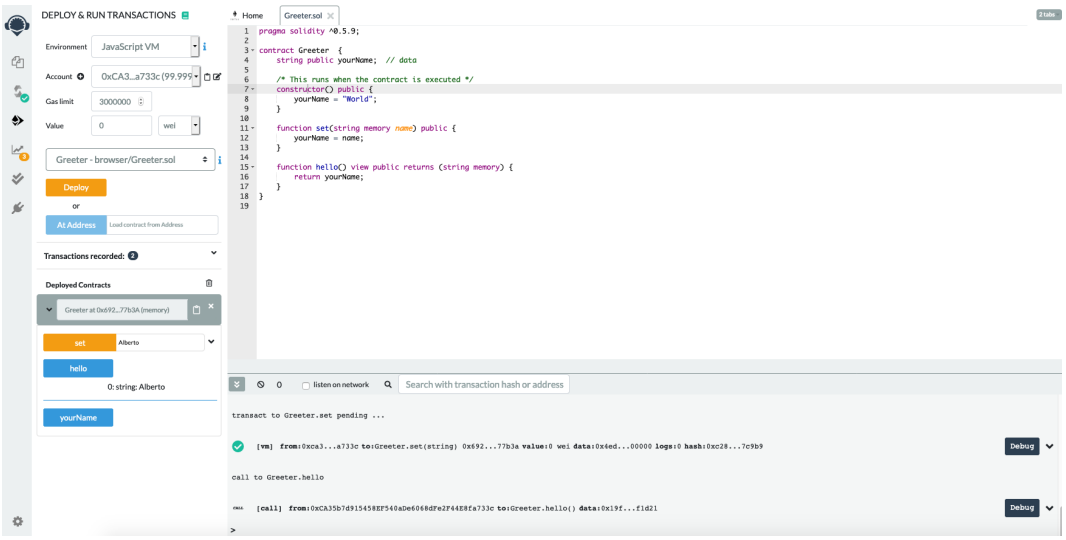

Figura A.4: Vista principal del entorno.

La herramienta de gestión de plugins permite intalar diferentes módulos que añaden nuevas herramientas de seguridad como son: Venom, GasProfiler, verificación del código de un contrato inteligente usando Etherscan API[Eth20b], etc. En la Figura A.5 se ven los parámrtros de configuración.

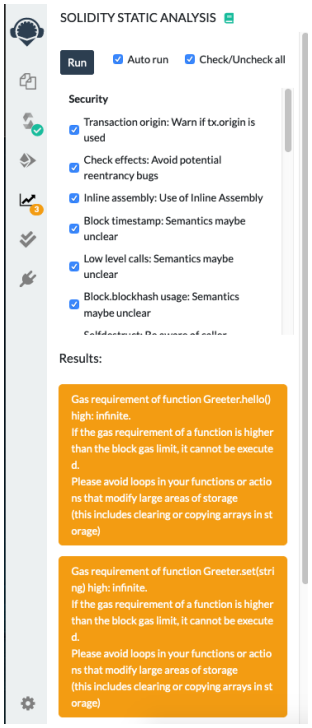

Figura A.5: Parámetros de configuración.

A.4. Solgraph

La herramienta está escrita en javascript y puede descargarse como un paquete de NodeJs. El comando para lanzar la aplicación es:

```
$ solgraph <MyContract>.sol > <MyContract>.dot
```

Y para convertir el archivo .dot en .png se usará la herramienta graphviz:

```
$ dot -Tpng MyContract.dot -o MyContract.png
```

En la Figura A.6 se ve un contrato de ejemplo para ser analizado y los resultados se muestran en la Figura A.7 A.6

A.5. Manticore

Manticore trabaja con Python 3, teniendo algunas dependencias en librerías de dicho lenguaje, aunque no resulta complicada de instalar. La metodología que sigue está basada en baterías de pruebas o *test cases* y para que un contrato sea 100 % seguro deberá de pasarlas todas.

A.6. Smartcheck

A efectos de prueba se ha usado la versión local que permiten instalar los creadores de la herramienta y que puede ser encontrado en el repositorio oficial[Sma18a].

```
contract Example {
  uint balance;

  function Example() {
    callingFunction(1000000);
  }

  function callFunction(uint amount) internal {
    balance = amount;
  }

  function Retirada() {
    msg.sender.send(balance);
  }

  function ConseguirBalance() constant returns(uint) {
    return balance;
  }
}
```

Figura A.6: Contrato para ser analizado.

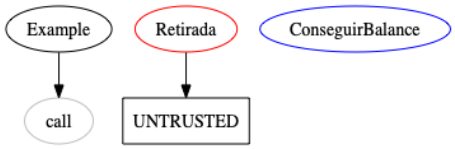

Figura A.7: Resultado de la ejecución de Solgraph

Para ejecutar la herramienta se debe de lanzar de la siguiente forma:

```
$ smartcheck -p <contract_name>.sol -r <xml_rules_namefile>.xml
```

La forma en la que tiene Smartcheck para mostrar las diferentes vulnerabilidades y diferencias es la siguiente:

- RuleId: es el nombre de la regla (o vulnerabilidad) que se ha encontrado.
- PatternId: es el identificador numérico específico de la regla previamente dicha.
- Severtity: es el nivel de riesgo que Smartcheck por defecto da a la vulnerabilidad.
- Line: número de linea donde se encuentra la vulnerabilidad.
- Column: número de columna donde se encuentra la vulnerabilidad.
- Content: contenido de la linea donde se encuentra la vulnerabilidad.

Los desarrolladores ponen a disposición una **pequeña librería** con las diferentes vulnerabilidades que Smartcheck se encargará de buscar. Esta librería tiene el aspecto que se ve en la Figura A.8. La gestión que tiene de los errores la basa en la gravedad que puedan tener puntuándola del 1 al 5. Muestra la línea, el identificador del patrón, la línea, la columna y el contenido.
Los resultados tras analizar un contrato cualquiera han sido los mostrados en la Figura A.9

```
MacBook-Pro-de-Alberto:tests_not_used_now Ture$ smartcheck -p
SOLIDITY_CONSTANT_OUTCOME_OF_LOOP.sol          SOLIDITY_OPAQUE_PREDICATE.sol
SOLIDITY_CONSTANT_STATE_VARIABLE.sol           SOLIDITY_PITFALLS_IN_RACE_CONDITION.sol
SOLIDITY_CROSS_FUNCTION_RACE_CONDITION.sol     SOLIDITY_REFACTOR_CONFUSING_PRECONDITION.sol
SOLIDITY_DEAD_CODE.sol                         SOLIDITY_REPEATED_COMPUTATION_IN_LOOP.sol
SOLIDITY_DIVISION_BY_ZERO.sol                  SOLIDITY_SEND_BALANCE_BEFORE_CHANGE.sol
SOLIDITY_DOS_WITH_BLOCK_GAS_LIMIT.sol          SOLIDITY_SHORT_ADDRESS_ATTACK.sol
SOLIDITY_DUPLICATED_STORAGE.sol                SOLIDITY_SIMILAR_VARIABLE_NAMES.sol
SOLIDITY_EXPENSIVE_OPERATIONS_IN_LOOP.sol      SOLIDITY_TRANSACTION_ORDERING_DEPENDENCE.sol
SOLIDITY_IMMUTABILITY.sol                      SOLIDITY_UNCHECKED_MATH.sol
SOLIDITY_INTEGER_OVERFLOW_UNDERFLOW.sol        SOLIDITY_USING_UBTRUSTED_INPUTS.sol
SOLIDITY_LOOP_FUSION.sol                       file.sol
SOLIDITY_MISSING_INPUT_VALIDATION.sol          testgrammarfile.sol
MacBook-Pro-de-Alberto:tests_not_used_now Ture$ smartcheck -p SOLIDITY_SHORT_ADDRESS_ATTACK.sol
SOLIDITY_SHORT_ADDRESS_ATTACK.sol
jar:file:/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle/smartcheck-2.0-jar-with-dependencies.jar!/solidity-rules.xmlrul
eId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 2
column: 4
content: functionsendMoney(addressto,uintamount){to.transfer(amount);}

SOLIDITY_VISIBILITY :1
```

Figura A.8: Librería de Smartcheck.

```
MacBook-Pro-de-Alberto:tests Ture$ smartcheck -p uninitialized.sol
uninitialized.sol
jar:file:/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle/smartcheck-2.0-jar-with-dependencies.jar!/solidity-rules.xmlrul
eId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 1
column: 16
content: ^

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 19
column: 4
content: functioninit(){balancesInitialized[msg.sender]=0;}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 23
column: 4
content: functionuse(){require(balances[msg.sender]==balancesInitialized[msg.sender]);}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 35
column: 4
content: functionset(MyStructstoragest,uintv){st.val=4;}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 49
column: 4
content: functioninit(){stInitiliazed.set(v);}

ruleId: SOLIDITY_VISIBILITY
patternId: 910067
severity: 1
line: 53
column: 4
content: functionuse(){require(st.val==stInitiliazed.val);}

ruleId: SOLIDITY_VISIBILITY
patternId: b51ce0
severity: 1
line: 5
```

Figura A.9: Resultados con Smartcheck.

A.7. Mythril

Dada su naturaleza modular, Mythril[myt18] proporciona un directorio donde se alojan los diferentes módulos encargados de un tipo específico de vulnerabilidad a encontrar.

La herramienta está preparada para ser extensible por módulos, cualquiera puede crear su propio módulo y añadirlo a la herramienta. El proceso de instalación consiste en descargar una imagen del contenedor específico proporcionado por los creadores o instalar un módulo Python 3. Para ejecutar la herramienta se deberán de usar los siguientes comandos:

```
$ myth analyze <solidity-file>.sol
$ myth analyze -a <contract-address>
```

Añadiendo opciones de línea de comando que uno desee.

A.8. SolMet

Para usar la herramienta, es necesario instalar el paquete Maven dado que el archivo de ejecución es una extensión .jar, para ello:

```
$ sudo apt install maven
```

El uso es realmente simple, SolMet comprueba tanto un contrato inteligente como una ruta, permitiendo insertar múltiples archivos al mismo tiempo y haciendo un archivo CSV para representar los resultados. Para ejecutarlo sólo hay que abrir una interfaz de comando y escribir:

```
$ java -jar solmet-0.1.jar <archivo solidity o ruta> <resultado>.csv
```

A.9. Vandal

Para ejecutar la herramienta de desensamblado, sólo hay que ejecutar en la línea de comandos el siguiente comando:

```
$ /bin/disassemble p examples/dao_hack.hex
```

Tanto la implementación del descompilador como los archivos de las bibliotecas se encuentran en la carpeta src. Para ejecutarlo:

```
$ /bin/decompile examples/dao_hack.hex
```

El registro de datos se almacena en el directorio datalog/directory. Además, las diferentes funcionalidades que ofrece se pueden ver con el flag –help como se ve en la Figura A.10 y la Figura A.11

A.10. EthIr

Tras analizar el archivo con extension .disasm se crea un archivo de salida con el nombre y extension: *rbr.rbr*.

A.11. Erays

Para ejecutar la herramienta tan solo es necesario introducir el comando correspondiente de Python de la siguiente forma:

```
$ python structurer.py <file>.hex
```

Una vez se ejecute se creará un archivo temporal. El archivo de salida es una extensión pdf debido a que simplifica la lectura, dicho archivo deberá de ser después procesado.

```
[ubuntu@smartcontracts:~/ALBERTO/vandal$ bin/disassemble --help
usage: disassemble [-h] [-p] [-s] [-o OUTFILE] [-v] [-vv]
                   [infile [infile ...]]

An EVM bytecode disassembler

positional arguments:
  infile                file from which decompiler input should be read (stdin
                        by default).

optional arguments:
  -h, --help            show this help message and exit
  -p, --prettify        colourize the disassembly and separate block
                        boundaries with a newline
  -s, --strict          halt and produce no output when malformed input is
                        given
  -o OUTFILE, --outfile OUTFILE
                        file to which decompiler output should be written
                        (stdout by default).
  -v, --verbose         emit verbose debug output to stderr.
  -vv, --prolix         emit debug output to stderr at higher verbosity level.
```

Figura A.10: Opciones de desasemblado.

A.12. Rattle

Rattle funciona con código en tiempo de ejecución, obligando a la conversión de archivos hexadecimales en bytes. Este proceso se puede llevar a cabo usando la opción xxd o –bin-runtime disponible en el compilador de Solidity.

Para ejecutar el análisis:

```
$ head contract.bin 60806040526004361061008357600035 7c0100000000000000000000000...
$ cat contract.bin | xxd -r -ps > <archivo de salida>.bytecode

o
\hl{se sale de la linea}
$ solc --bin-runtime <archivo de entrada>.sol 2>/dev/null | tail -n1 | xxd -r -ps > <
```

Y después.

```
$ python3 rattle-cli.py --input <file to analyze>.bin -O
```

A.13. Osiris

La funcionalidad es prácticamente similar a la de Oyente aunque contiene comandos nuevos. Si corremos la opción help se mostrarán los siguientes comandos:

A.14. Securify

Securify necesita la biblioteca java 8, Soufflé (igual que Vandal) y Solc. Los comandos básicos de análisis permiten el uso de archivos de código Solidity, decompilados o binarios de la EVM.
El archivo de salida será un archivo json, aunque para poder obtener más información a parte de las advertencias y vulnerabilidades será necesario el uso del comando –co flag.

```
[ubuntu@smartcontracts:~/ALBERTO/vandal$ bin/decompile --help
usage: decompile [-h] [-a] [-b] [-g [FILE]] [-t [DIR]] [-d]
                 [-o [OPCODE [OPCODE ...]]] [-c CFG_STRING] [-C FILE] [-v]
                 [-vv] [-n] [-V]
                 [infile] [outfile]

An EVM bytecode disassembly decompiler that generates three-address code for
program analysis. Use config.ini to set further configuration options.

positional arguments:
  infile                file from which decompiler input should be read (stdin
                        by default).
  outfile               file to which decompiler output should be written
                        (stdout by default).

optional arguments:
  -h, --help            show this help message and exit
  -a, --disassembly     decompile dissassembled input. Overrides '-b'.
  -b, --bytecode        disassemble and decompile bytecode input. This is the
                        default mode.
  -g [FILE], --graph [FILE]
                        generate a visual representation of basic block
                        relationships with the format inferred from the given
                        output filename (cfg.dot by default). Non-DOT formats
                        like pdf, png, etc. require Graphviz to be installed.
                        Use html to generate an interactive web page of the
                        graph.
  -t [DIR], --tsv [DIR]
                        generate tab-separated .facts files for Souffle and
                        write files to the specified directory, which will be
                        recursively created if it does not exist (current
                        working directory by default).
  -d, --dominators      If producing tsv output, also include graph dominator
                        relations.
  -o [OPCODE [OPCODE ...]], --opcodes [OPCODE [OPCODE ...]]
                        If producing tsv output, also include relations
                        encoding all occurrences of the specified list of
                        opcodes. Opcode X will be stored in op_X.facts.
  -c CFG_STRING, --config CFG_STRING
                        override settings from the configuration files in the
                        format "key1=value1, key2=value2..." (with the
                        quotation marks).
  -C FILE, --config_file FILE
                        read the settings from the given file; any given
                        settings will override the defaults. Read from
                        config.ini if not set.
  -v, --verbose         emit verbose debug output to stderr.
  -vv, --prolix         higher verbosity level, including extra debug messages
                        from dataflow analysis and elsewhere.
  -n, --no_out          do not output decompiled graph.
  -V, --version         show program's version number and exit.
```

Figura A.11: Opciones de decompilado.

A.15. MadMax

Actuando de la misma manera que Vandal generando un diagrama de flujo gracias a la utilidad del decompilador.

La manera de ejecutar la herramienta es muy sencilla.

```
$ bin/decompile -n -v -c "remove_unreachable=1" -g loop.html loop.hex
```

Donde:

```
  .oooooo.             o8o            o8o
 d8P'  `Y8b            `"'            `"'
888      888  .oooo.o oooo  oooo d8b oooo   .oooo.o
888      888 d88(  "8 `888  `888""8P `888  d88(  "8
888      888 `"Y88b.   888   888      888  `"Y88b.
`88b    d88' o.  )88b  888   888      888  o.  )88b
 `Y8bood8P'  8""888P' o888o d888b    o888o 8""888P'

usage: osiris.py [-h] (-s SOURCE | -ru REMOTE_URL) [--version] [-b] [-j] [-e]
                 [-p] [--error] [-t TIMEOUT] [-v] [-r] [-gb] [-dl DEPTH_LIMIT]
                 [-gl GAS_LIMIT] [-st] [-ll LOOP_LIMIT] [-w]
                 [-glt GLOBAL_TIMEOUT] [-a] [--debug] [--generate-test-cases]
                 [-c] [-m]

optional arguments:
  -h, --help            show this help message and exit
  -s SOURCE, --source SOURCE
                        local source file name. Solidity by default. Use -b to
                        process evm instead. Use stdin to read from stdin.
  -ru REMOTE_URL, --remoteURL REMOTE_URL
                        Get contract from remote URL. Solidity by default. Use
                        -b to process evm instead.
  --version             show program's version number and exit
  -b, --bytecode        read bytecode in source instead of solidity file.
  -j, --json            Redirect results to a json file.
  -e, --evm             Do not remove the .evm file.
  -p, --paths           Print path condition information.
  --error               Enable exceptions and print output. Monsters here.
  -t TIMEOUT, --timeout TIMEOUT
                        Timeout for Z3 in ms (default 100 ms).
  -v, --verbose         Verbose output, print everything.
  -r, --report          Create .report file.
  -gb, --globalblockchain
                        Integrate with the global ethereum blockchain
  -dl DEPTH_LIMIT, --depthlimit DEPTH_LIMIT
                        Limit DFS depth (default 50).
  -gl GAS_LIMIT, --gaslimit GAS_LIMIT
                        Limit Gas (default 4000000).
  -st, --state          Get input state from state.json
  -ll LOOP_LIMIT, --looplimit LOOP_LIMIT
                        Limit number of loops (default 10).
  -w, --web             Run Osiris for web service
  -glt GLOBAL_TIMEOUT, --global-timeout GLOBAL_TIMEOUT
                        Timeout for symbolic execution in sec (default 50
                        sec).
  -a, --assertion       Check assertion failures.
  --debug               Display debug information
  --generate-test-cases
                        Generate test cases each branch of symbolic execution
                        tree
  -c, --cfg             Create control flow graph and store as .dot file.
  -m, --model           Output models generated by the solver.
root@fea8b9b2032c:~/osiris#
```

Figura A.12: Comandos disponibles en Osiris.

- C especifica la configuración del ajuste, usando una entrada de valor clave.

- G especifica el archivo gráfico de salida.

A.16. ContractLarva

Uno de los mayores problemas encontrados al tratar de compilar la herramienta fue la instalación en Ubuntu siguiendo los estándares. ContractLarva usa una versión de Haskell y esta a su vez utiliza funciones que la versión por defecto del Ubuntu no tiene. La herramienta se invoca escribiendo el siguiente comando:

```
$ contractlarva <specification.dae> <fichero entrada.sol> <fichero salida.sol>
```

Si analizamos el comando punto por punto veremos:

- Especificación dae: son las propiedades para capturar los eventos. ContractLarva permite capturar sólo dos tipos: eventos de flujo de control (puntos de función de entrada y salida) y eventos de flujo de datos correspondientes a cambios en los valores de las variables.
- Fichero entrada: el contrato de entrada.
- Fichero salida: resultados del análisis.

El aspecto que tiene un fichero DAE puede verse en la Figura A.13

La estructura del DAE y explicación de los campos puede encontrarse en la documentación oficial.

Nota: las herramientas que son obsoletas, no pueden ser probadas o no se han incluido.

```
monitor Casino{

    declarations{
        uint total;
    }

    DEA NoReduction {
        states {
            TableOpen: initial;
            TableClosed: accept;
            BetPlaced;
            PotReduced: bad;
        }
        transitions {
            TableOpen -[after(closeTable) | pot == 0 ]-> TableClosed;
            TableOpen -[after(placeBet(_value)) | _value <= pot ~> total += _value;]-> BetPlaced;
            BetPlaced -[after(timeoutBet)]-> TableOpen;
            BetPlaced -[after(resolveBet)]-> TableOpen;
            BetPlaced -[pot@(LARVA_previous_pot > pot)]-> PotReduced;
        }
    }

    DEA OpenUntilResolution {
        states {
            TableClosed: initial;
            TableOpen;
            BetPlaced;
            TableCloseDuringBet: bad;
        }
        transitions {
            TableClosed -[after(openTable)]-> TableOpen;
            TableOpen -[after(closeTable)]-> TableClosed;
            TableOpen -[after(placeBet)]-> BetPlaced;
            BetPlaced -[after(resolveBet)]-> TableOpen;
            BetPlaced -[after(timeoutBet)]-> TableOpen;
            BetPlaced -[after(closeTable)]-> TableCloseDuringBet;
        }
    }
```

Figura A.13: Contenido de un fichero DAE

Bibliografía

[Bit19] Bitsofcode. Differences between call and send. https://bitsofco.de/calling-smart-contract-functions-using-web3-js-call-vs-send/, 2019.

[Cry19] Crytic. Solc-select: A script to quickly switch between solidity compiler versions. https://github.com/crytic/solc-select, 2019.

[DAS19] Monika Di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78. IEEE, 2019.

[Eth18] Ethereum. Py-solc: Python wrapper around the solc solidity compiler. https://github.com/ethereum/py-solc, 2018.

[Eth20a] Ethereum. Solidity 0.66 compiler documentation. https://solidity.readthedocs.io/en/v0.6.6/installing-solidity.html, 2020.

[Eth20b] Etherscan. Etherscan api. https://etherscan.io/apis, 2020.

[Fou20] Ethereum Foundation. Solidity documentation. https://solidity-es.readthedocs.io/es/latest/, 2020.

[Inc20] Docker Inc. Docker sdk python library. https://docker-py.readthedocs.io/en/stable/, 2020.

[JF17a] Alex Groce Josselin Feist, Gustavo Grieco. Oyente documentation github. https://github.com/melonproject/oyente, 2017.

[JF17b] Alex Groce Josselin Feist, Gustavo Grieco. Slither documentation github. https://github.com/crytic/slither, 2017.

[LL18] Hrishi Olickel Prateek Saxena Aquinas Hobor Loi Luu, Duc-Hiep Chu. Oyente: Making smart contracts smarter. https://www.comp.nus.edu.sg/~prateeks/papers/Oyente.pdf, 2018.

[LVCSOGV20] Antonio López Vivar, Alberto Turégano Castedo, Ana Lucila Sandoval Orozco, and Luis Javier García Villalba. An analysis of smart contracts security threats alongside existing solutions. *Entropy*, 22(2), 2020.

[myt18] Mythril github. https://github.com/ConsenSys/mythril, 2018.

[Nak08] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. *Bitcoin.org*, 2008.

[Pac19] G. Pace. Slither: A static analysis framework for smart contracts. https://arxiv.org/abs/1908.09878, 2019.

[rat19] rattle. Eciclopedia de códigos y asginaciones de rattle. https://www.trailofbits.com/presentations/rattle/, 2019.

[Red] Redhat. Microservicios. https://www.redhat.com/es/topics/microservices.

[rem20] Remix ide github. https://github.com/ethereum/remix-ide, 2020.

[Rev19] Raine Revereh. Solgraph github. https://github.com/raineorshine/solgraph, 2019.

[Sma18a] SmartDec. Smartcheck github. https://github.com/smartdec/smartcheck, 2018.

[Sma18b] SmartDec. Smartcheck página web. https://tool.smartdec.net/, 2018.

[Sol19] SolMet. Solmet github. https://github.com/chicxurug/SolMet-Solidity-parser, 2019.

[van19] vandal. Vandal github. https://github.com/usyd-blockchain/vandal, 2019.

[Wik20a] Bitcoin Wiki. Proof of work. http://en.wikipedia.org/w/index.php?title=Proof%20of%20work&oldid=950649470, 2020.

[Wik20b] Wikipedia. DOT (graph description language) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=DOT%20(graph%20description%20language)&oldid=956133055, 2020. [Online; accessed 07-June-2020].

[WSX+20] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 2020.