Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Fuzzing de *smart contracts*: EchidnaAFL

Tesis de Licenciatura en Ciencias de la Computación

Dago de Renteria y Melissa Regnier

Directores: Juan Pablo Galeotti y Diego Garbervetsky

Buenos Aires, 2023

FUZZING DE *SMART CONTRACTS*: ECHIDNAAFL

En los últimos años, han cobrado popularidad y gran capital financiero las tecnologías *blockchain* y en particular *Ethereum*, que introdujo el concepto de *smart contracts* que permitió el desarrollo de una gran variedad de proyectos. Esto dio lugar a la investigación y desarrollo de herramienta de *testing* de *smart contracts*, en particular *fuzzing*. En esta tesis, en búsqueda de una herramienta para extender, llevamos a cabo en primer lugar un análisis del *state-of-the-art* de las herramientas de *fuzzing* existentes, lo que nos llevó a elegir a Echidna, debido a su popularidad, simpleza de extensión y disponibilidad de código y desarrolladores. En base a esto, realizamos una investigación de *reverse-engineering* que nos permitió documentar extensamente el algoritmo de Echidna, obteniendo diversos diagramas que disponibilizamos para el uso de la comunidad. Una vez hecho esto, implementamos una versión de Echidna que incorpora conceptos de la herramienta *AFL Fast*, que asigna energía a los elementos del corpus en función de la frecuencia del camino explorado por cada elemento. Además, implementamos una versión *random* que asigna energías uniformes para utilizar como testigo en el análisis. Finalmente, mediante el uso de tres distintos *benchmarks*, comparamos los resultados obtenidos por las tres distintas versiones y concluimos que la adaptación implementada obtiene resultados equivalentes a la original pero con un overhead de tiempo significativo, mientras que la versión *random* también obtiene resultados similares sin ningún *overhead* de tiempo. Esto nos lleva a concluir que la estrategia de elección de elementos de corpus no pareciera influir lo suficiente en la eficiencia de la herramienta ya que su comparación con *random* que representa la elección trivial de elementos de corpus resulta en el mismo nivel de *performance*. Como trabajo futuro, se pueden seguir explorando los resultados observados haciendo uso de *benchmarks* más complejos y extensos que permitan otorgarle mayor confianza a lo concluido o incluso proponer nuevas estrategias de *feedback* utilizando la documentación provista en esta tesis para facilitar el desarrollo.

Palabras claves: *Ethereum*, *smart contracts*, *fuzzing*, Echidna, AFL-Fast, *feedback*, elección de elementos de corpus.

AGRADECIMIENTOS

A Santi, por todo

A nuestras familias, por todo el apoyo en estos años de horas de estudio y sufrimiento

A todos nuestros amigos de la facultad, por las risas y momentos compartidos

A JP y a Diego, por aceptar ser nuestros directores y acompañarnos durante todo el proceso

A los jurados, por tomarse el tiempo

Índice general

1.. Introducción 1
1.1. Blockchain 1
1.2. Ethereum: Smart contracts 3
1.3. Testing automático: Fuzzing 3
1.3.1. *Blackbox fuzzing* 4
1.3.2. *Greybox fuzzing* 4
1.3.3. *Boosted greybox fuzzing* 4

2.. *Estado del arte* 7

3.. Investigación sobre Echidna 13
3.1. Resultados del análisis en profundidad 15
3.1.1. ***Run campaign*** 15
3.1.2. ***Create sequence*** 16
3.1.3. ***Generate transaction*** 17
3.1.4. ***Sequence mutator*** 19
3.1.5. ***Run sequence*** 21
3.1.6. ***Execute transaction*** 22
3.1.7. ***Update test*** 22

4.. Extensión de Echidna 25
4.1. Implementación 27
4.1.1. EchidnaAFL 27
4.1.2. Random 30

5.. Evaluación 33
5.1. Preguntas de investigación 33
5.2. Casos de estudio 33
5.2.1. Setup general 33
5.2.2. Contratos autogenerados: *Maze Benchmark* 35
5.2.3. Contratos básicos: *Smart-Pulse Benchmark* 35
5.2.4. Contratos de aplicación real: *Uniswap Benchmark* 36
5.3. Análisis y resultados 36
5.3.1. ¿Podemos afirmar que EchidnaAFL supone una mejora en comparación con la versión original? 36
5.3.2. ¿Influye de manera relevante la elección de elementos del corpus en Echidna? 42
5.3.3. ¿Cuál es el *overhead* de tiempo asociado a EchidnaAFL? 45

6.. Conclusiones y trabajo futuro 49

1. INTRODUCCIÓN

En los últimos años se popularizó el uso de blockchains y entre ellas Ethereum [1], que propone una blockchain con un lenguaje de programación Turing completo integrado permitiendo publicar en la misma blockchain *smart contracts* (código que se auto ejecuta, más detalles en la sección 1.2) [4]. Las grandes inversiones en este sector incrementaron exponencialmente su crecimiento a la vez que provocaron que el mismo se convirtiera en el blanco de muchas amenazas cibernéticas. Debido a la inmutabilidad del código publicado en Ethereum, se vuelve extremadamente importante validar correcta y exhaustivamente los *smart contracts* antes de publicar los mismos en la red. Para esto, hoy en día existen múltiples proyectos de *testing* automatizado. En esta tesis nos proponemos evaluar diversas herramientas existentes, en particular, de *fuzzing* (técnica de *testing* automatizado, explicado en 1.3), con el objetivo de lograr extender alguna de ellas utilizando técnicas de *fuzzing* convencionales con la idea de que esto resulte en una mejora para la comunidad que la utiliza.

Llevamos a cabo un análisis detallado (sección 2), en el cual estudiamos las distintas herramientas existentes sopesando la viabilidad, facilidad y utilidad de extensión, entre otros aspectos. En base a este estudio, decidimos utilizar ***Echidna***, una herramienta *open-source* con un buen grado de aceptación en la comunidad que se encuentra actualmente mantenida. Una vez elegido el proyecto en el que nos enfocaríamos, realizamos un análisis en profundidad (sección 3) de la herramienta para poder entender minuciosamente el actual funcionamiento de la misma. Esto nos permite determinar en qué consiste el algoritmo de *fuzzing* de *Echidna* para luego intentar determinar puntos de mejora. El resultado del proceso que llevamos a cabo fueron múltiples diagramas y documentación del funcionamiento de la herramienta, lo que constituye la mayor contribución de esta tesis ya que permite un mayor entendimiento para futuros desarrolladores que deseen extenderla o mejorarla.

Es así que, utilizando técnicas de boosted greybox fuzzing (sección 1.3.3), implementamos dos nuevas versiones para analizar su rendimiento (detalles en sección 4), con la hipótesis de que la aplicación de una técnica de feedback tradicional a Echidna lograría obtener mejores resultados (mayor exploración del código). Finalmente, para poder llevar a cabo una comparación de estas con respecto al algoritmo original, ejecutamos diversas campañas de fuzzing sobre distintos casos de estudio y analizamos los resultados obtenidos (como se puede ver en sección 5).

A continuación introducimos conceptos claves para el entendimiento del desarrollo de esta tesis.

1.1. Blockchain

Las tecnologías *blockchain* proponen un mecanismo descentralizado de comunicación entre distintos nodos de una red, con un determinado mecanismo de consenso que permite que los nodos lleguen a un acuerdo sin necesidad de una entidad centralizada. Principalmente, estas tecnologías cobraron importancia como opuestas a entidades financieras tradicionales, que pueden ser influenciadas o llevar a cabo decisiones arbitrarias sobre las cuales no se tiene ningún control como usuario.

Una red *blockchain* consiste, entonces, en nodos distribuidos que mantienen un historial de transacciones, que se divide en una secuencia de bloques conocida como *blockchain*. Las transacciones nuevas son registradas en un *pool* de donde se eligen las que formarán parte del siguiente bloque, como podemos ver en la figura 1.1. Una vez un bloque es añadido y aceptado por los nodos, el mismo no puede ser alterado ni eliminado. Cada red tiene definido su propio mecanismo de consenso que determina cómo se agrega un nuevo bloque a la blockchain y en caso de coexistir múltiples versiones cuál es la cadena aceptada. Un ejemplo de mecanismo de consenso es *Proof-of-Work* [13], utilizado por Bitcoin, que consiste en realizar cómputos intensivos (relacionados con operaciones de hash) para lograr crear el próximo bloque; en caso de recibir dos versiones de cadenas, un nodo siempre mantendrá aquella que sea más larga. Otro ejemplo es *Proof-of-Stake* [14], en donde los nodos bloquean un cierto monto de la moneda relacionada a la blockchain como garantía para participar en el proceso de validación de bloques. Dependiendo del monto bloqueado, un nodo tendrá más o menos probabilidad de ser elegido y así recompensado por la creación de dicho bloque. Este mecanismo es el que utiliza actualmente Ethereum.

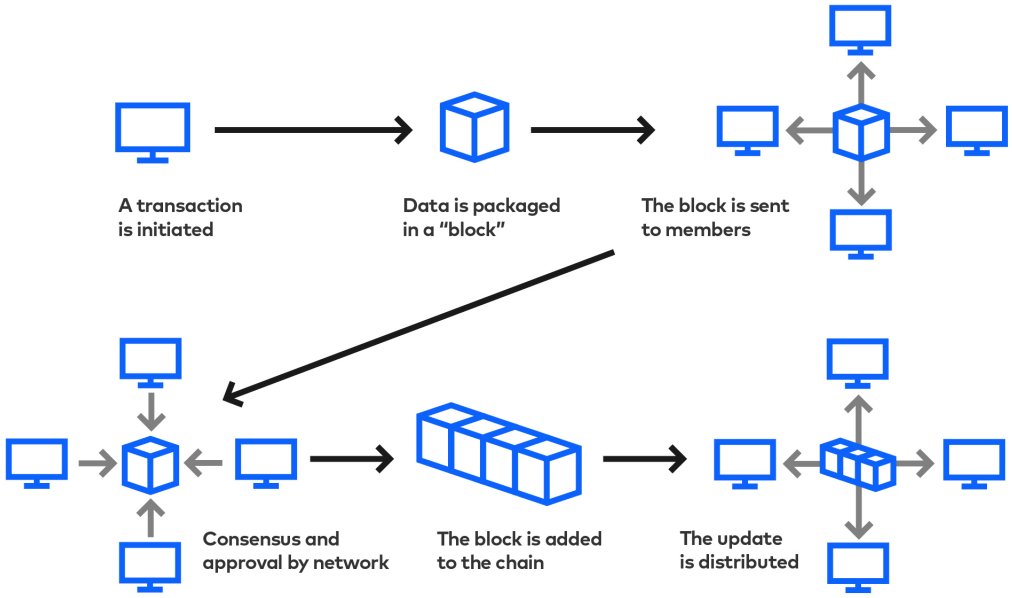

Fig. 1.1: Flujo de una red de blockchain. Fuente: https://www.slalom.com/insights/how-blockchain-will-disrupt-your-industry

Esto les permite a las tecnologías *blockchain* [15] ofrecer diversos beneficios:

- **confidencialidad**: los participantes de la red no están necesariamente vinculados con una entidad física
- **transparencia**: el historial de transacciones es público, compartido e inmutable
- **bajos costos de operación** (asociados con el costo de computación)
- **descentralización**: no cuenta con un único punto de falla, ya que al ser un sistema distribuido la caída de un solo nodo (no importa cuál) no causa la caída del sistema

- **transacciones rápidas**: al no existir intermediarios y eliminarse procesos manuales, las transacciones pueden realizarse en un tiempo menor que las tradicionales.

Estas tecnologías se hicieron muy populares con la adopción de los criptoactivos, que permiten definir criptomonedas y otros activos (entre ellos, NFTs [16]) utilizando como infraestructura una *blockchain* dada.

1.2. Ethereum: Smart contracts

En el año 2015, se lanzó un proyecto conocido como Ethereum [1], que proponía una *blockchain* que buscaba proveer un protocolo generalizado para la creación de aplicaciones descentralizadas facilitando este proceso quitando la necesidad de desarrollar una nueva infraestructura. Para ello, provee un lenguaje de programación Turing completo integrado, llamado Solidity [8], que permite publicar en la misma red de Ethereum *smart contracts*, que son ejecutados en una máquina virtual, lo que se conoce como Ethereum Virtual Machine (EVM)[2]. Un ***smart contract*** define un contrato entre distintas partes que se encuentra embebido en código auto-ejecutable. El contrato define una interfaz mediante la cual las distintas partes pueden interactuar, siempre que se respeten las precondiciones definidas. Así, en Ethereum existen distintos tipos de transacciones: regulares (de transferencia de *ether* entre dos billeteras), de despliegue de contratos (que llevan a cabo la publicación de los mismos) y de ejecución de un contrato (que permite ejecutar funciones públicas de contratos ya publicados).

Esta modalidad provee una mayor transparencia al poder visualizarse el código que se está ejecutando a la vez que elimina la necesidad de intermediarios. Además, al publicarse el contrato en la *blockchain*, el mismo es inmutable por lo que garantiza a los usuarios su cumplimiento. Esto permite que se desarrollen aplicaciones y se completen transacciones de manera descentralizada y ha concentrado mucho capital financiero; en febrero de 2023 se registran más de 200 millones de *addresses* [3], un número que sigue creciendo.

El gran capital invertido en el mercado de las criptomonedas, que alcanzó el trillón de dólares para el final de 2022 [5], presenta un gran incentivo para entidades maliciosas que buscan atacar las aplicaciones desarrolladas. Desde el comienzo de las criptomonedas se han registrado hackeos millonarios como el famoso ataque DAO en 2016 [6] (pérdida de 70 millones de USD) y el más reciente ataque a Axie Infinity [7] (pérdida de más de 600 millones de USD).

Sumado a esto, la inmutabilidad del código de los *smarts contracts* una vez deployados en la *blockchain* implica que el código publicado no pueda ser actualizado [4] y si bien existen determinados mecanismos para mitigarlos [17], estos pueden llevar un tiempo hasta que los cambios se vean reflejados. Esto hace que llevar a cabo un correcto testeo de los smarts contracts para evitar vulnerabilidades y *bugs* se vuelva fundamental.

1.3. Testing automático: Fuzzing

En la industria del desarrollo de software, existen múltiples técnicas de validación de programas que buscan chequear la consistencia entre implementación y especificación, conocidas como *testing*. Si bien existen muchos tipos de *testing*, podemos distinguir entre dos grandes categorías: manual y automático. En nuestro caso nos concentraremos en esta última, que consiste en la creación de casos de test de manera automática y provee una

test-suite de manera rápida y que no requiere actualizaciones manuales de los casos de prueba.

Una técnica muy utilizada dentro del testing automatizado es aquella conocida como *fuzzing*, que consiste en, dado un programa, ejecutar el mismo muchas veces generando inputs aleatorios para cada ejecución. Esta técnica suele además ser útil para descubrir bugs y errores en casos que un humano podría no testear debido a que su conocimiento preexistente le otorga una mirada lógica, pero sesgada. El verdadero poder de *fuzzing* yace en la correcta generación de inputs, ya que si se lleva a cabo sin ningún criterio, la mayor cantidad de los casos que testearemos serán inputs inválidos o pertenecerán a la misma clase de equivalencia de *coverage*.

Dentro de los distintos tipos de *fuzzing*, queremos distinguir tres clases descriptas a continuación.

1.3.1. *Blackbox fuzzing*

Se trata de una de las versiones más simples de *fuzzing* que no tiene en cuenta ningún tipo de *feedback* sobre los resultados que se van obteniendo. Este tipo de *fuzzing* parte de un conjunto de inputs (*semilla*) que se ejecutan y luego mientras se tenga presupuesto de computo se elige uno de esos inputs al azar para ser mutado y probado.

1.3.2. *Greybox fuzzing*

Se diferencia de la anterior categoría ya que si un nuevo input mutado logra aportar cobertura de líneas previamente no exploradas entonces el mismo es agregado al conjunto semilla para futuras mutaciones. De esta forma, mientas que en *blackbox-fuzzing* el conjunto semilla se mantiene estático durante la ejecución en *greybox-fuzzing* este se modifica dinámicamente. Por otro lado, para implementar este tipo de *fuzzing* se necesita un mecanismo que permita determinar cobertura de código a partir de una ejecución (algo que *blackbox-fuzzing* no requiere).

1.3.3. *Boosted greybox fuzzing*

Este tipo de *fuzzing* agrega a lo anterior la noción de **energía** o probabilidad de un input de la semilla de ser elegido. Los fuzzers de este tipo buscan asignar mayor energía a aquellos *inputs* que sean más prometedores, es decir, que parezcan poder conseguir una mayor cobertura.

Así, una posibilidad es definir la energía de un input dado en función a la frecuencia del camino que recorrió el mismo en comparación al resto. La idea detrás de esta definición es que al otorgarle una mayor probabilidad al input que logró recorrer un camino poco explorado, lograremos al mutarlo seguir recorriendo en más profundidad la rama correspondiente del *control flow graph* [18].

Dado un input s, su camino explorado $p(s)$ y una función f de frecuencia de un camino dado, definimos la energía del mismo como:

$$e(s) = \frac{1}{f(p(s))^a}$$

Donde a nos permite controlar el decaimiento de un camino cuanto más visto es. Mientras más grande sea a, más rápido decaerá la energía del *input* a medida que su recorrido en el árbol de ejecución se vuelva más frecuente.

De esta forma, luego de cada ejecución, se actualizan las energías de cada input ya que la frecuencia de los caminos se ve afectada.

2. *ESTADO DEL ARTE*

Dado que nuestro propósito es realizar una extensión de una herramienta de *fuzzing* de *smart contracts* existente, primero debemos llevar a cabo un análisis del *estado del arte* de las mismas. Esto nos permite tomar una decisión informada sobre la elección de la herramienta y a la vez tener un entendimiento de las diversas técnicas de *fuzzing* utilizadas actualmente en la industria. En este análisis y en el marco de esta tesis, es importante definir tanto los aspectos que queremos tener en cuenta como la prioridad de cada uno de ellos. En este sentido, la primer cualidad que buscamos es que el proyecto sea *open-source* ya que en caso de no serlo, no sería posible ningún tipo de extensión del código. En segundo lugar, debemos tener en cuenta si logramos compilar la herramienta a partir del código fuente y la documentación proveída. Y por último existen otras características a sopesar que influyen de forma similar en nuestra elección:

- **usado por la comunidad**: el nivel de adopción nos permite determinar qué tan útil y performante es la herramienta y cuánto su mejora impactaría a los usuarios de *smart contracts* en general.

- **mantenimiento**: el mantenimiento indica que la misma no ha sido abandonada o deprecada por los autores y que existe un equipo detrás que podría resolver dudas y problemas con los que nos podamos encontrar en el desarrollo.

- **extensibilidad**: el diseño, la arquitectura y las buenas prácticas utilizadas en el código existente tienen un impacto directo en la facilidad de extender la herramienta.

Teniendo en cuenta estos aspectos clave y utilizando la investigación existente en *Ethereum Smart Contract Analysis Tools: A Systematic Review* [19], además de una búsqueda de papers del entorno académico de fuzzing de smart contracts, pudimos confeccionar una lista de herramientas existentes que plasmamos en la tabla comparativa 2.1. Es importante mencionar que las características plasmadas en esta tesis acerca de las herramientas analizadas se corresponden con la información disponible al momento de realizarse esta comparación (en noviembre de 2022), y como tal podrían diferir del estado actual de las mismas.

En la misma podemos observar que para cada herramienta especificamos características de las mismas relevantes para nuestro análisis: si la herramienta es o no *open-source*, si analiza Solidity *code* o directamente el *bytecode* compilado a partir del contrato y el lenguaje en el cual se encuentra desarrollado. A partir de esta información, descartamos entonces en un primer lugar todos los fuzzers que no son *open-source*. Esto nos deja 8 fuzzers por analizar y comparar:

Herramienta	*Open source*?	Analiza Solidity *code*?	Analiza sólo *bytecode*?	Lenguaje
GasGauge	Sí	Sí	No	Python
EthPloit	No	Sí	No	Python
Etherolic	No	Sí	No	Rust
Harvey	No	No	Sí	-
ReGuard	No	Sí	No	C++
SoliAudit	No	No	Sí	Python
Bran (integrado a Harvey)	No	No	Sí	Go
xFuzz	No	Sí	No	Python y C
Echidna	Sí	Sí	No	Haskell
ContractFuzzer	Sí	No	Sí	Go
EVMFuzz	Sí	Sí	No	Python
sFuzz	Sí	No	Sí	C++
Vultron	Sí	Sí	No	JavaScript
Smartian	Sí	Sí	No	F#
Foundry	Sí	Sí	No	Rust

Tab. 2.1: Tabla comparativa de las distintas herramientas analizadas de fuzzing de smart contracts de Ethereum

- **GasGauge**[22]: es una herramienta de análisis estático que se dedica a detectar vulnerabilidades de *out-of-gas denial of service* utilizando *fuzzing* de *smart contracts*. Debido a que la misma está dedicada a vulneralidades relacionadas exclusivamente con consumo de *gas*, decidimos descartarla ya que buscábamos una herramienta de uso más genérico en el *fuzzing* de *smart contracts*.

- **ContractFuzzer**[21]: fuzzer que busca detectar vulnerabilidad de seguridad en smart contracts. Cuenta con siete oráculos distintos que utiliza para poder detectar siete tipos de vulnerabilidades predefinidas. Utiliza una *test net* en la que deploya los contratos a testear además de otros extraídos de Ethereum (mediante un web crawler de Etherscan [3]) que sirven tanto para testearse como para poder utilizarse como input de funciones de otros contratos que requieren *addresses* como parámetro. En este último caso,se analiza el cuerpo de la función para poder determinar cuál es la firma que deben cumplir los métodos de dicho contrato usado como input. Además, se define y deploya un contrato como atacante que se utiliza específicamente para detectar *reentrancy bugs*. Los logs obtenidos luego de ejecutar los test cases se utilizan para determinar la presencia de vulnerabilidades. Sin embargo, no incorpora ningún tipo de feedback para la definición de nuevos casos de test, no soporta el chequeo de propiedades que no deben ser violadas definidas por el usuario y tampoco genera un reporte de *code coverage*. Por otro lado, al utilizarse una *test net* local, los tiempos de setup para la ejecución son mayores. Por otro lado, en lo que respecta a su mantenimiento, la última actualización registrada del repositorio de Github fue en 2018. Sin embargo, a pesar de esto último, en Github cuenta con 212 stars y 84 forks

[1], por lo que no carece totalmente de popularidad. Debido mayormente a la falta de uso de feedback en el algoritmo de *fuzzing* y las distintas falencias mencionadas, decidimos descartar esta opción.

- **EVMFuzz**[20]: si bien se trata de una herramienta de *fuzzing*, no realiza *fuzzing* de *smart contracts* sino de la EVM para encontrar bugs en la implementación de la misma, por lo que la descartamos como opción.

- **Vultron**[24]: busca construir un oráculo capaz de distinguir inconsistencias entre el estado del contrato y las transacciones realizadas. Si bien este oráculo que se construye a partir del deploy de contratos en una test net podría ser utilizado en un fuzzer de smart contracts, lo desechamos debido a que no se trata de un fuzzer de smart contracts en sí mismo.

- **sFuzz**[23]: implementa un fuzzer guiado por *feedback* que utiliza un algoritmo genético a alto nivel para conseguir una *test suite* que busca maximizar el *code coverage* del contrato dado. De esta forma, el problema de generación de tests se transforma en un problema de optimización. Si bien el algoritmo está basado en la idea de AFL [12] e incorpora varios conceptos del mismo, introduce como nueva abstracción las *just-missed branches*: aquellas branches alcanzadas por un test que poseen una rama no explorada. Estas son utilizadas a la hora de evaluar la función de fitness de manera tal que se calcula la branch distance de cada just-missed branch y aquel test con menor distancia se agrega al pool de test seeds a seguir explorando. En el contexto de esta herramienta, un test es una configuración inicial de una *test net* y una secuencia de llamadas a funciones públicas de contratos. El enfoque propuesto mencionado anteriormente permite entonces no tener que contar con un CFG (es decir, puede sólo tomar *bytecode*) ni tener que construirlo, lo cual requeriría simular el stack de la EVM. Si bien el repositorio de Github de la herramienta muestra que el proyecto no está siendo activamente mantenido y la adopción de la comunidad es moderada (73 stars y 26 forks [1]), decidimos no desecharlo por el momento debido al interesante modelo de *feedback* que propone.

- **Echidna** [9]: se trata de una herramienta de *fuzzing* de smart contracts desarrollada en Haskell y que admite tres modos de ejecución con distintos objetivos: utilizando propiedades definidas por el usuario que se buscan romper, comprobando aserciones implementadas en las mismas funciones del contrato o buscándose la mayor exploración de código posible. También incluye la estimación del uso de gas de las transacciones generadas. Contempla dos etapas, una primer etapa de preprocesamiento en la que se utiliza Slither [27] para extraer valores de interés a testear del contrato y posteriormente una etapa de campaña de *fuzzing* en la que se generan las transacciones aleatorias y se busca detectar violaciones de las propiedades o aserciones definidas. Es una herramienta performante, flexible y de gran grado de adopción por la comunidad (2300 stars y 292 forks[1]) que cuenta con un soporte sostenido por parte de sus autores además de una trayectoria de varios años en el ámbito. Como desventaja, no cuenta con una documentación detallada acerca de cómo funciona el algoritmo de fuzzing utilizado. [2]

[1] Consultado el 7 de septiembre de 2023.
[2] De los mismos autores, existe otra herramienta cross-platform de fuzzing paralelizado llamada medusa

- **Smartian** [25]: define una herramienta de testing que utiliza diversos oráculos para determinar si un test descubre *bugs* e incorpora *dataflow analysis* como método de feedback del algoritmo de fuzzing. Esto último lo diferencia de otras herramientas analizadas, ya que se trata de enfoque innovador de feedback: para cada variable, define una cadena de usos de la misma dentro de la ejecución de cada test, con la idea de que si un test descubre cadenas de uso nuevas, las mismas pueden suponer casos de uso distintos del estado interno del contrato y por lo tanto interesantes de explorar (incluso si no generan cobertura de nuevas líneas de código). Además, para el proceso de mutación de tests, utiliza *dynamic symbolic execution* y *random mutation* a nivel de secuencia (ya sea agregando, sacando o swapeando transacciones) y de transacción (alterando argumentos de la función del contrato ejecutada). En cuanto a la adopción, también es moderada (107 stars y 13 forks [1]) y el mantenimiento del mismo por sus autores pareciera ser acotado.

- **foundry** [26]: permite realizar *fuzzing* de tests parametrizados escritos por el usuario, fuzzeando dichos parámetros para intentar hacer fallar al test. Esto significa que la herramienta en sí no construye secuencias o transacciones a ejecutar si no que esa tarea recae en el usuario. En cuanto a los detalles de implementación y del algoritmo, no se encuentran documentados por lo que constituye una desventaja a la hora de extenderlo. A pesar de esto y que el fuzzing que implementa es acotado y simple, cuenta con una gran adopción por parte de la comunidad (6500 stars y 1000 forks [1]) por lo que en una primera instancia tampoco la desechamos.

En base a lo analizado, nos quedamos con las últimas cuatro herramientas mencionadas como alternativas ya sea por su adopción y flexibilidad (Echidna y Foundry) o por su innovación en el *feedback* del algoritmo de *fuzzing* (Smartian y sFuzz). En el gráfico 2.1, podemos ver la fluctuación de la popularidad de cada herramienta (medida en cantidad de stars en Github) desde su creación hasta la actualidad. Como podemos ver, tanto sFuzz como Smartian cuentan con una baja popularidad. Echidna, por otra parte, cuenta con una mayor trayectoria y adopción creciente sostenida, mientras que Foundry, creado hace relativamente poco, muestra un crecimiento exponencial de interés por parte de la comunidad llegando a sobrepasar a Echidna considerablemente. Sin embargo, esta diferencia de adopción probablemente se deba a que Foundry es un *framework* completo para compilar, testear y deployar contratos, por lo que tampoco podemos asegurar que su popularidad esté dada por sus capacidades de *fuzzing* de *smart contracts*.

Ya con estas herramientas preseleccionadas decidimos llevar a cabo una prueba empírica con un contrato simple para determinar la viabilidad de la compilación a partir del código fuente además de la calidad del output obtenido. En cuanto a Foundry, no logramos compilarlo con ningún sistema operativo (Linux, MacOS y Windows). En lo que respecta a sFuzz, si bien conseguimos que compile en Linux, a la hora de ejecutar la herramienta con un simple contrato, causa una excepción que no logramos solucionar. Por otro lado, pudimos compilar y ejecutar con resultados exitosos tanto Smartian como Echidna, obteniendo satisfactoriamente los tests generados por parte de ambos. Esto significa que debemos optar por alguna de estas dos opciones. Teniendo en cuenta los aspectos mencionados al comienzo de este capítulo, nos decidimos por Echidna ya que cuenta con una mayor adopción, mantenimiento y un aceptable grado de extensibilidad.

[28], que como al momento de realizarse la investigación se encontraba recién en fases tempranas de desarrollo, no fue incluida en el análisis.

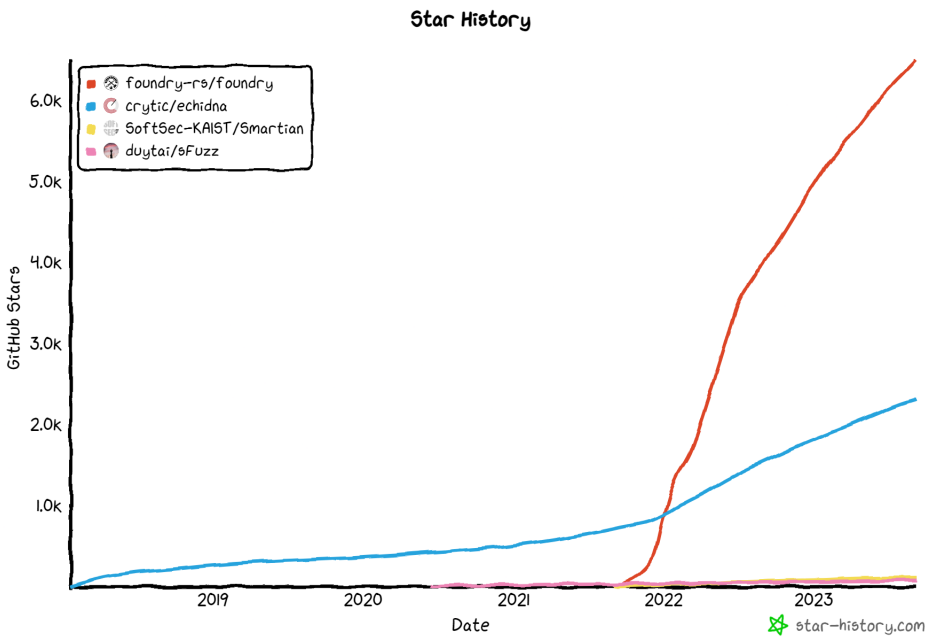

Fig. 2.1: Historial de stars en Github de las herramientas de fuzzing finalmente evaluadas.
Fuente: `star-history.com`

En el siguiente capítulo, nos dedicaremos a realizar un análisis detallado del funcionamiento actual de la herramienta elegida.

3. INVESTIGACIÓN SOBRE ECHIDNA

Una vez tomada la decisión de la herramienta a utilizar y teniendo en cuenta que nuestro objetivo es extender la misma, debemos poseer un profundo entendimiento del actual funcionamiento de la herramienta para así poder identificar los distintos puntos en los que el algoritmo podría ser mejorado. Dado que además el proyecto cuenta con una documentación muy dedicada a los usuarios de la misma y no tanto así a un desarrollador que busca extenderla o compilarla desde su código fuente, la información que poseemos acerca de la lógica de la herramienta es limitada. Esto significa que tenemos que llevar adelante un proceso extenso de análisis de la misma que nos permita completar los huecos en nuestro conocimiento de Echidna.

A grandes rasgos, podemos decir que Echidna lleva a cabo un proceso de fuzzing de *smart contracts* mediante la generación y ejecución de secuencias de transacciones que busca encontrar ejemplos que no satisfagan los *echidna tests* proveídos. Con respecto a esto último, Echidna cuenta con distintos modos de ejecución donde cada uno supone un tipo distinto de *echidna test*. Estos son:

- ***Property***: Permite al usuario definir propiedades que se deben cumplir en base a los contratos a testear. Llamamos propiedades a funciones booleanas que no cuentan con parámetros. Las mismas se suelen definir en un contrato que extiende a aquel que se desea testear (para evitar modificar el contrato original) y deben tener como prefijo *"echidna_"*. En este caso, cada *echidna test* se corresponde con una propiedad definida.

- ***Assertion***: Utiliza las aserciones definidas en el contrato a testear como *echidna tests*. Las mismas se definen utilizando la palabra reservada `assert`, nativa de Solidity para definir aserciones. Es decir, Echidna buscará hacer fallar a los *asserts* que se encuentren el código.

- ***Exploration***: Modo especial de ejecución pensado para poder correr benchmarks que no tiene ningún *echidna test* asociado para evitar *overheads*.

- ***Optimization***: El objetivo de esta configuración es lograr maximizar el valor de una función, que funcionará como un *echidna test*. En este caso, no se busca que el *echidna test* falle.

- ***Overflow*** [1]: Detecta integer overflows (Sólo disponible a partir de Solidity 0.8.0)

Una vez que se inicia una campaña de fuzzing, la misma se ejecutará hasta lograr hacer fallar todos los *echidna tests* definidos (dependiendo del modo que se haya elegido esto será o no posible) o hasta quedarse sin presupuesto de ejecución, que puede ser medido en cantidad máxima de secuencias generadas y/o en límite de tiempo. Ambas medidas son parte de los parámetros de la configuración, entre varias otras configuraciones existentes que mencionaremos de ser pertinente (estas pueden ser consultadas en [31]). Otro ejemplo de configuración es la variable *coverage* que permite al usuario indicar si se quiere calcular

[1] Este modo fue añadido luego de que se llevara a cabo el análisis de la herramienta y se implementaran las mejoras mencionadas más adelante

la cobertura de código de las secuencias ejecutadas y utilizarlo como *feedback* del fuzzer. También es posible definir una *whitelist* o *blacklist* de funciones de los contratos a testear.

En el momento en el que la campaña finaliza, la herramienta muestra al usuario el estado final de los *echidna tests* junto con otra información genérica de la ejecución. En la figura 3.2, podemos ver el output de Echidna corrido en modo *property* para un ejemplo simple de contrato que posee tres propiedades definidas y dos funciones públicas. El código del mismo junto con el *coverage* obtenido en la ejecución lo podemos observar en la figura 3.1 donde podemos ver que se logró cubrir la totalidad de los métodos públicos (las *echidna propiedades* no deben contarse como parte de la interfaz del contrato). También podemos ver en el output de la campaña que en caso de no lograr romperse un test, el mismo aparecerá como "PASSED"(como es el caso de *echidna_revert_always* y *echidna_alwaystrue*). Por el contrario, si se logra romper el test, el mismo se marcará como "FAILED"(como *echidna_sometimesfalse*) y Echidna nos proporcionará la secuencia de transacciones que logró hacerlo fallar.

/home/melissa/echidna/tests/solidity/basic/flags.sol

```
 1 | *r  | contract Test {
 2 |     |   event Flag(bool);
 3 |     |
 4 |     |   bool private flag0 = true;
 5 |     |   bool private flag1 = true;
 6 |     |
 7 |  *  |   function set0(int val) public returns (bool){
 8 |  *  |     if (val % 100 == 0)
 9 |  *  |       flag0 = false;
10 |     |   }
11 |     |
12 |  *  |   function set1(int val) public returns (bool){
13 |  *  |     if (val % 10 == 0 && !flag0)
14 |  *  |       flag1 = false;
15 |     |   }
16 |     |
17 |     |   function echidna_alwaystrue() public returns (bool){
18 |     |     return(true);
19 |     |   }
20 |     |
21 |     |   function echidna_revert_always() public returns (bool){
22 |     |     revert();
23 |     |   }
24 |     |
25 |     |   function echidna_sometimesfalse() public returns (bool){
26 |     |     emit Flag(flag0);
27 |     |     emit Flag(flag1);
28 |     |     return(flag1);
29 |     |   }
30 |     |
31 |     | }
32 |     |
```

Fig. 3.1: Ejemplo de archivo de coverage resultante de una campaña de fuzzing de Echidna.

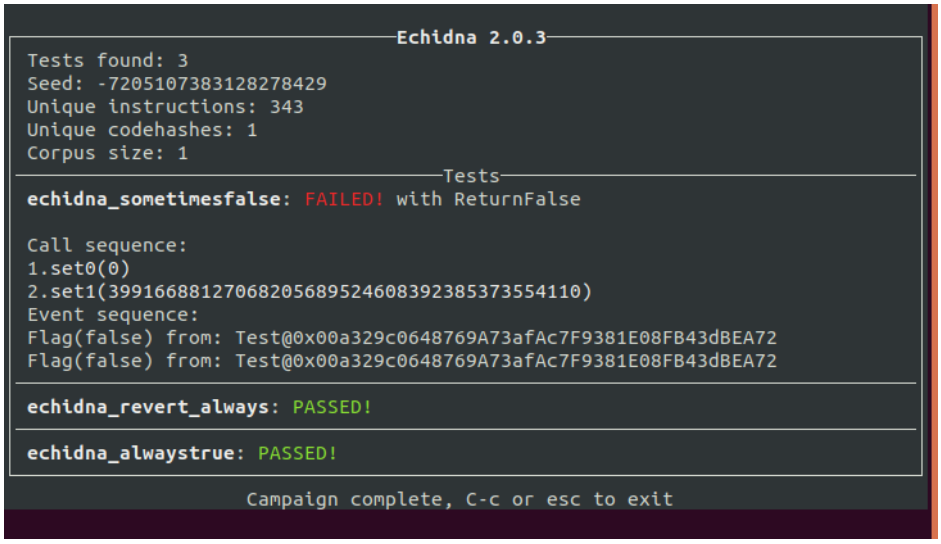

Fig. 3.2: Resultado de una campaña de fuzzing de Echidna.

3.1. Resultados del análisis en profundidad

A partir del código fuente, pudimos organizar la lógica del funcionamiento de Echidna en módulos conceptuales que describen distintas partes del comportamiento a alto nivel de la herramienta y que a su vez pueden contener submódulos. Estos mismos los presentamos en diagramas, donde la idea es ir desde conceptos más abstractos de alto nivel a conceptos cada vez más puntuales y concretos, de manera que podamos comprender a la herramienta en una metodología *top-down*.

Se deben tener en consideración las siguientes convenciones generales utilizadas en los diagramas que presentaremos a continuación:

- Los **óvalos de color blanco** representan nodos internos de ideas atómicas (o que no vale la pena ampliar).
- Los **óvalos de color amarillo** representan nodos terminales en la ejecución del concepto.
- Los **rectángulos blancos** son nodos internos que representan un submódulo que será explicado por otro diagrama.
- Las **flechas entrantes** ubicadas a la izquierda de los gráficos que no parten de ningún nodo representan el punto de inicio del diagrama.
- Las **etiquetas** que se pueden encontrar en ciertas flechas indican ya sea resultado del nodo origen, input del nodo destino o decisiones posibles en caso de que el nodo predecesor sea una condición (indicado con signo de pregunta).

3.1.1. *Run campaign*

En el más alto nivel, tenemos el concepto de ejecución de la campaña de *fuzzing* (*run campaign*), que podemos ver representado en la figura 3.3.

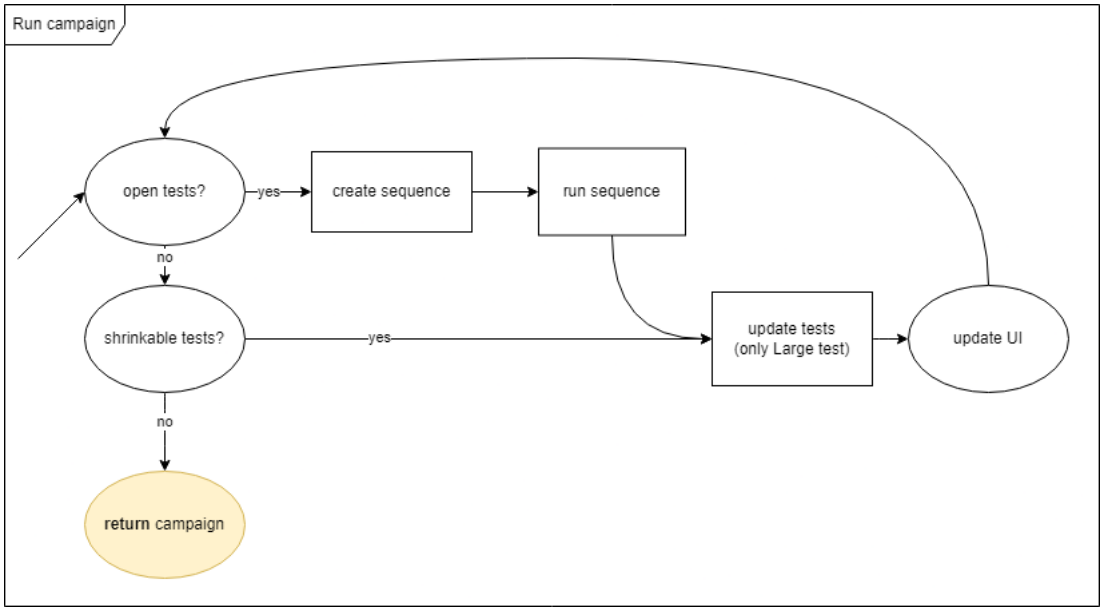

Fig. 3.3: Diagrama que representa la lógica a alto nivel del algoritmo que ejecuta una campaña de fuzzing en Echidna.

Como podemos ver, la idea general es que mientras todavía queden test sin fallar, se seguirán creando nuevas secuencias (sección 3.1.2) que luego serán ejecutadas (sección 3.1.6) para poder evaluar si se logró hacer fallar algún nuevo test (sección 3.1.7). Los cambios en el estado de los tests se van reflejando en la UI que se muestra por consola. Una vez que no quedan tests en estado *open* (es decir, que todos han fallado o se ha considerado que han pasado), se pasa a una etapa de *shrinking* que consiste en tratar de reducir el tamaño y simplificar las secuencias que han hecho fallar los distintos tests. Esta etapa también se encuentra limitada por un parámetro configurable (*shrinkLimit*) que busca fijar la cantidad de intentos de simplificación de tests. Por último, cuando no quedan tests por reducir o intentos disponibles de *shrinking*, la campaña finaliza.

Por otro lado, la totalidad de esta lógica corre en un *thread* que tiene configurado un *timeout*, configuración que mencionamos anteriormente, para que pasado ese tiempo se termine la ejecución (los *tests* y *coverage* quedarán en el último estado disponible).

Vale la pena mencionar que antes de ejecutarse la campaña se llevan a cabo ciertas tareas de preprocesamiento, de las cuales nos interesa destacar el uso de **Slither**, una herramienta de análisis estático desarrollada por los mismos creadores de Echidna. Si bien la herramienta permite la detección de vulnerabilidades y otros análisis, en este caso en particular se utiliza para la extracción de valores que se encuentren en el código y que pueden ser valores interesantes para utilizar a la hora de fuzzear los contratos. Todas estas constantes extraídas son utilizadas para nutrir un diccionario (**GenDict**) que, entre otras cosas, se utilizará para generar los distintos tipos de datos necesarios en las transacciones. También Slither extrae distinta metadata de los contratos que es después utilizada por Echidna en su ejecución.

3.1.2. *Create sequence*

Antes de entender cómo Echidna crea una secuencia de transacciones a ejecutar, primero recordaremos lo explicado en la sección de greybox-fuzzing, en donde mencionamos

la existencia de un conjunto semilla de casos de tests que luego será extendido a medida que se obtengan nuevos casos de test interesantes. En el contexto de Echidna, dichos tests consisten en secuencias de transacciones y a dicho conjunto semilla lo llamamos *corpus*. El conjunto semilla inicial es el *initial corpus* que en el caso de Echidna puede o no estar presente (dependiendo de si el usuario lo provee) y a medida que se ejecutan secuencias que obtienen nuevo *coverage*, se agregan al *corpus* definitivo.

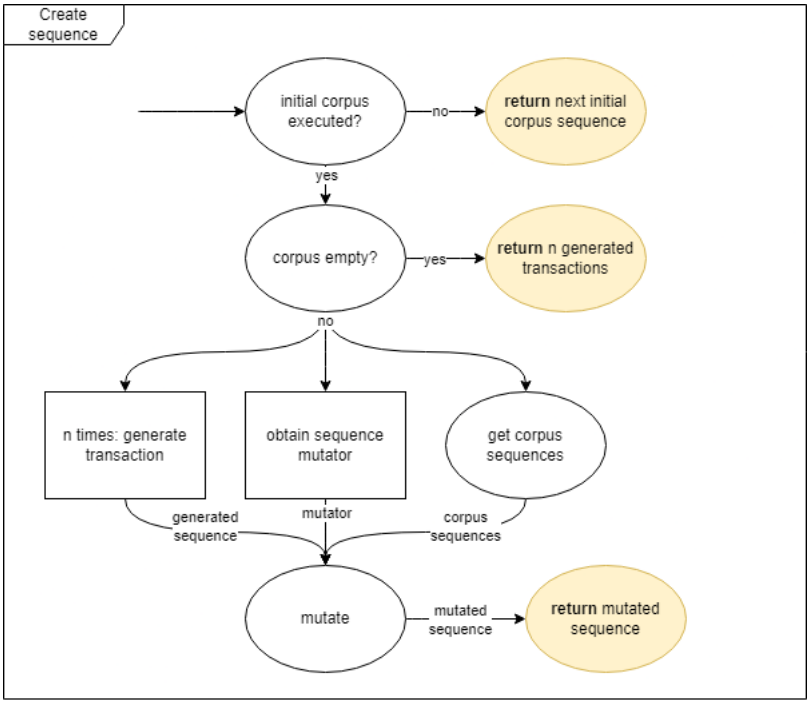

Fig. 3.4: Diagrama que representa la lógica a alto nivel del algoritmo que crear una secuencia de transacciones en Echidna.

Una vez entendido este concepto, pasemos a revisar el diagrama 3.4 de cómo se generan las secuencias en Echidna. En un primer lugar, en caso de constarse con secuencias en el *initial corpus*, la herramienta se asegurará de ejecutar primero dichas secuencias. Es decir, que si contamos con n secuencias en el initial corpus, las primeras n secuencias que obtendremos mediante este módulo serán las correspondientes a dicho conjunto. Una vez que estas secuencias hayan sido ejecutadas (y añadidas al corpus en caso de obtener nuevo coverage), las siguientes secuencias obtenidas buscarán depender del contenido del corpus. Lo que se buscará es obtener una nueva secuencia mutada a partir de secuencias del corpus, una nueva secuencia compuesta por n transacciones generadas y un *mutator* encargado de combinar dichas secuencias en una nueva. Esta secuencia obtenida a partir de la mutación será el resultado de este submódulo. Vale aclarar que en caso de no contarse con secuencias en el corpus, la secuencia a ejecutar estará puramente compuesta con transacciones generadas.

3.1.3. *Generate transaction*

En el anterior módulo, revisamos cómo se obtiene una nueva secuencia a ejecutar y mencionamos cómo es parte de este proceso la generación de una nueva transacción

para formar parte de dicha secuencia. Exploremos cuáles son los componentes de una transacción en Echidna y cómo los mismos son generados, como se ve en la figura 3.5:

- **sender address**: dirección de *wallet* o contrato que inicia la transacción. Echidna define tres valores *default* de *address* para ser usados con este fin pero permite mediante su configuración que sean definidos de manera *custom* por el usuario. Para definir este valor en la transacción generada, se obtiene un valor *random* de estos address definidos.

- **value**: corresponde al valor de ether enviado en la transacción. Para definirlo, se obtiene un valor random en un rango desde 0 al máximo configurado (por defecto, 100 ETH).

- **time delay**: es el tiempo expresado en milisegundos para avanzar el *timestamp* en relación al último *timestamp* ejecutado en la VM. Se utiliza un valor entre 0 y el máximo configurado.

- **block delay**: cantidad de bloques que se avanzarán con la transacción, lo que junto al último número de bloque percibido determinará el número de bloque de la transacción. También se utiliza un valor entre 0 y el máximo configurado correspondiente.

- **receiver contract address**: dirección del contrato que recibirá la transacción. Como una transacción ejecuta una función definida en alguno de los contratos por testear, se elige al azar uno de dichos contratos y se utiliza su *address*.

- **instantiated contract call**: esto hace referencia al método del contrato que se quiere ejecutar previamente elegido junto con la definición de los parámetros necesarios para ejecutarlo. Al objeto que engloba tanto el signature de la función como los parámetros de su llamada le llamamos `SolCall`.

Para generar la `SolCall`, se elige primero al azar una *signature* del contrato *receiver* elegido y luego se generan los parámetros que la misma requiere. Con este fin, utiliza el diccionario GenDict antes mencionado. El mismo, además de contener las constantes extraídas del código que antes explicamos, se va nutriendo de previas iteraciones de manera tal que las distintas llamadas a las *signatures* de secuencias anteriores que generaron nuevo *coverage* logran expandir el universo de SolCalls ya definidas, disponibles para su uso en nuevas transacciones en el GenDict. Como vemos en el diagrama 3.5, existe cierta probabilidad (***pSynthA***, definida en el mismo GenDict como configuración inicial) que determina si para la creación de la transacción se utilizan valores anteriores o se sintetizan nuevos valores. Esto se aplica tanto a la totalidad de la SolCall, como a cada parámetro de la misma en caso de sintetizarse.

Finalmente, una vez obtenida la SolCall, con cierta probabilidad (que depende del tipo del parámetro) uno de sus parámetros es mutado.

Con todos los valores generados, se devuelve la transacción instanciada.

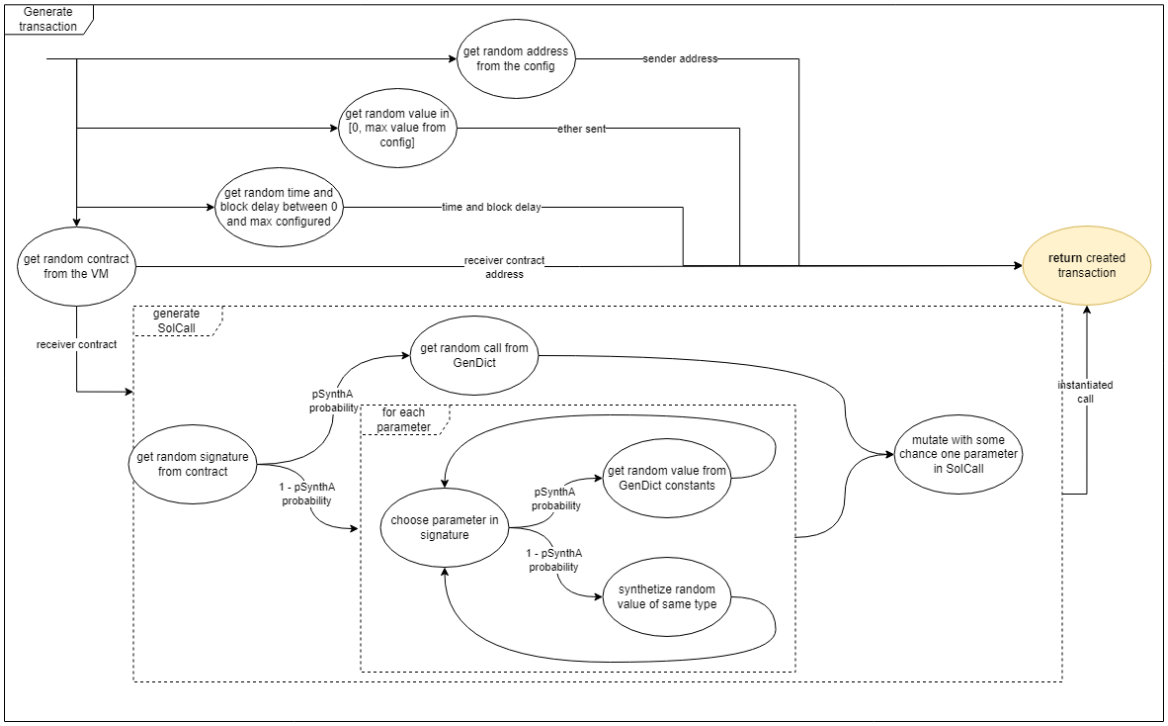

Fig. 3.5: Diagrama que representa la lógica a alto nivel del algoritmo que genera una nueva transacción para utilizar en la secuencia creada en Echidna.

3.1.4. *Sequence mutator*

Este módulo se encarga de obtener el *sequence mutator* a utilizar, que a partir de secuencias del corpus y una secuencia de transacciones generadas obtiene una nueva secuencia mutada. Como podemos ver en el diagrama 3.6, existen distintas estrategias de mutación que combinarán de manera distinta los elementos del corpus y las transacciones generadas. El *mutator* buscará componer una secuencia cuya longitud deberá coincidir con la definida por configuración (*seqLen*).

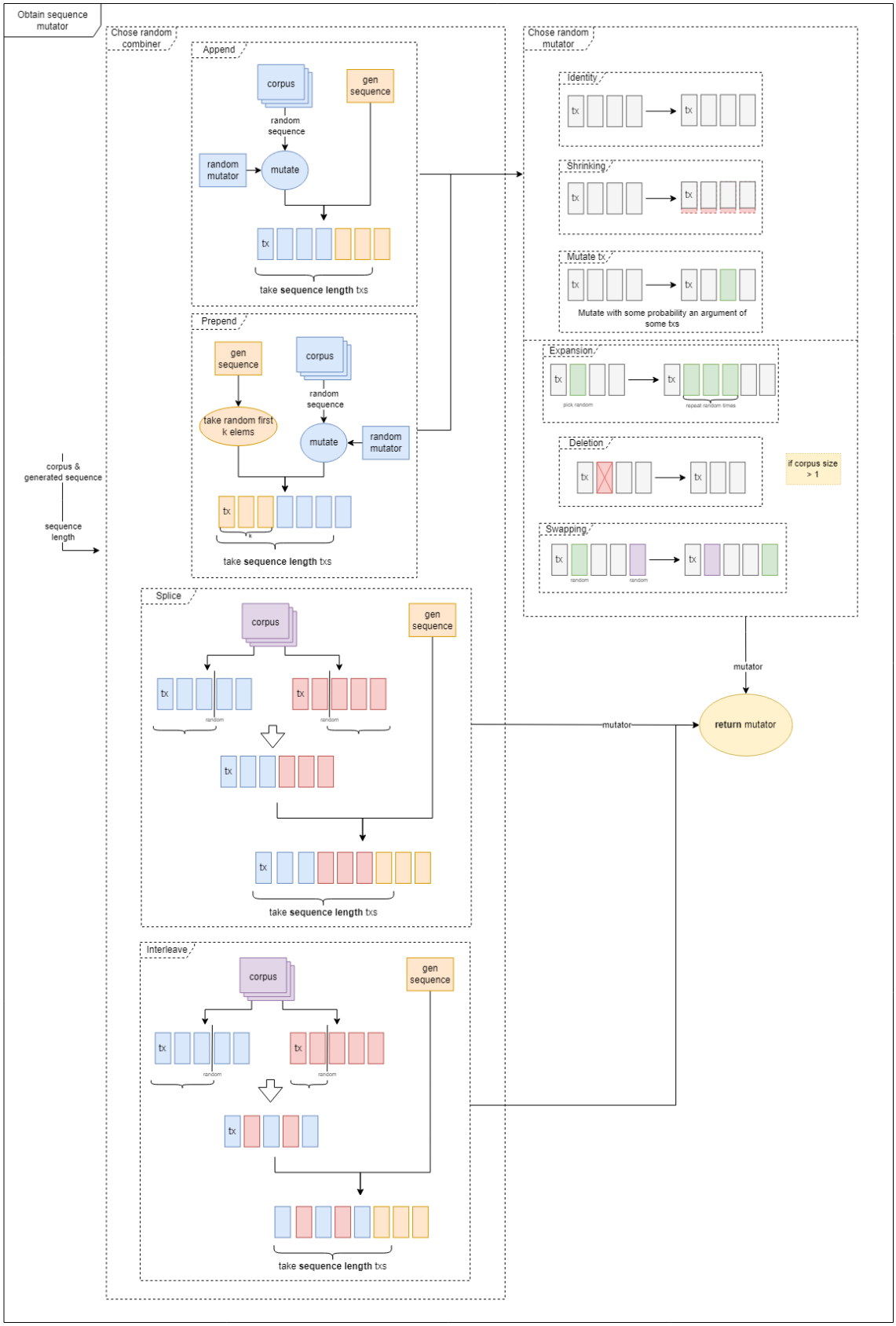

Fig. 3.6: Diagrama que representa la lógica a alto nivel del algoritmo que se encarga de mutar una secuencia en Echidna.

Un punto importante a mencionar que no se encuentra reflejado en el diagrama es la elección de las secuencias del corpus a utilizar. Esto constituye una parte importante en general de los algoritmos de *fuzzing* porque determina cómo se realiza el *feedback* y qué importancia se le dará a cada secuencia del corpus. La estrategia que utiliza Echidna es asignar energías en base a un número natural creciente donde a cada nueva secuencia que se agrega al corpus se le asigna una energía incrementada en 1 en comparación con la última secuencia añadida. Es decir, que las **nuevas secuencias añadidas tendrán más probabilidades de ser elegidas**, independientemente del nuevo *coverage* que hayan logrado. El número asignado a cada secuencia del corpus permanece estático, pero al agregarse nuevas secuencias con pesos mayores la probabilidad de la misma de ser elegida irá disminuyendo.

3.1.5. *Run sequence*

Una vez la nueva secuencia fue generada, la misma se debe ejecutar en la EVM simulada de Echidna. En el gráfico 3.7, podemos ver una vista a alto nivel del flujo de este proceso. Como se trata de una secuencia de transacciones, se debe llevar a cabo la ejecución de cada una de ellas respetando el orden en el que se encuentran en dicha secuencia y asegurándonos luego de cada ejecución de actualizar los *echidna tests* definidos, para poder verificar si hemos logrado hacer fallar alguno de ellos. Cada una de estas ejecuciones tendrá asociados: el resultado de dicha transacción (cuyos valores de retorno serán utilizados como constantes en el `GenDict`) y el gas utilizado (que permite actualizar la estimación de gas), a la vez que supondrá cambios en el estado de la VM mantenida, inherentes a la simulación de la ejecución de las transacciones. En caso de que estos cambios en la VM supongan nuevo *coverage*, esta secuencia ejecutada será añadida al *corpus*, con su peso correspondiente (calculado como mencionamos anteriormente).

Por último, si se encuentran nuevos contratos deployados en la VM, los *addresses* asociados también serán añadidos al `GenDict` como parte de las constantes de interés.

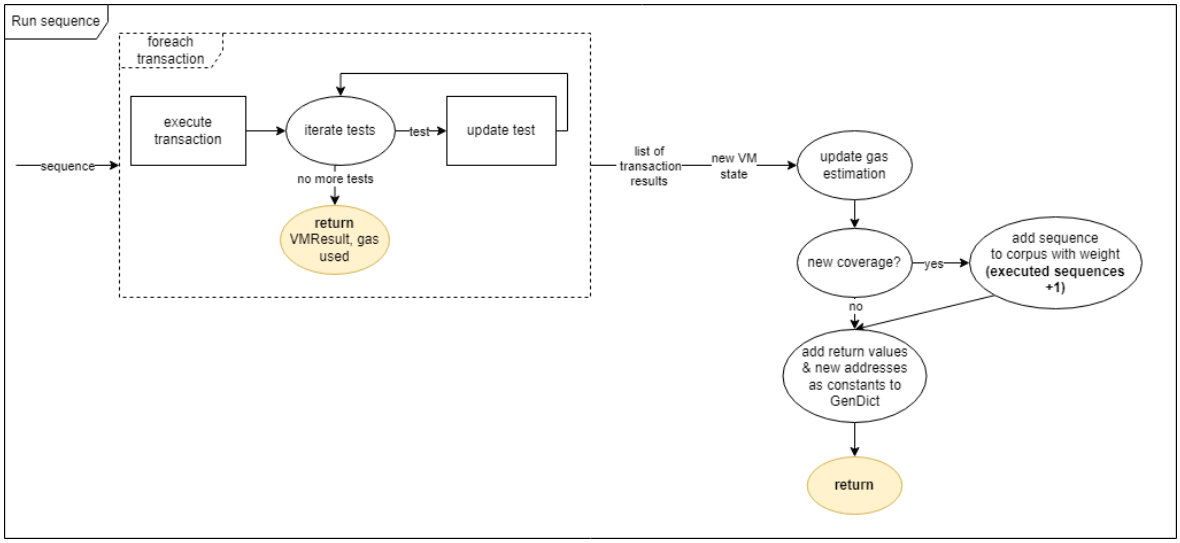

Fig. 3.7: Diagrama que representa la lógica a alto nivel del algoritmo que ejecuta en la EVM[2] simulada una secuencia de transacciones generadas.

3.1.6. *Execute transaction*

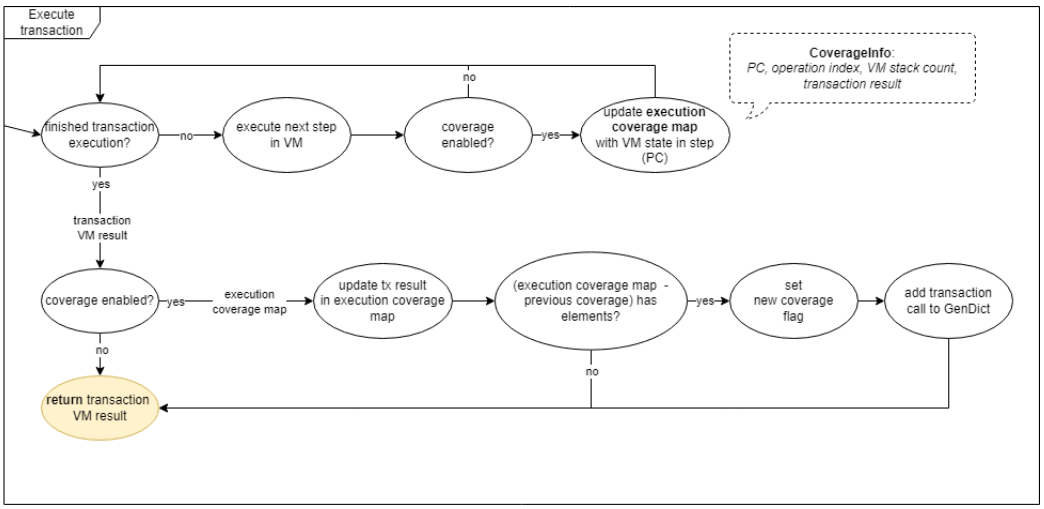

Fig. 3.8: Diagrama que representa la lógica a alto nivel del algoritmo que ejecuta en la EVM[2] simulada una transacción de una secuencia generada.

La lógica pertinente a la ejecución de una transacción se encuentra graficada en el diagrama 3.8. Se pueden distinguir dos flujos distintos dependiendo de si el cálculo y *feedback* de cobertura se encuentra o no habilitado en la configuración.

En caso de que el *coverage* no deba calcularse, el flujo de ejecución es simple: consiste en ejecutar cada step de la transacción hasta finalizar y retornar el resultado junto con la VM actualizada. Entendemos por *step* a cada operación a nivel *bytecode* de la EVM que se debe realizar para completar la transacción deseada.

Cuando el *coverage* se encuentra habilitado, ciertos pasos extra deben añadirse. Por un lado, luego de la ejecución de cada *step* se actualiza la información de *coverage* utilizando la metadata del estado en el que se encuentra la VM en ese momento. Al finalizarse la ejecución, dicha información recolectada sirve para compararla con el estado de *coverage* previo y, en caso de haberse obtenido nuevas líneas de cobertura, se añade la `SolCall` de la transacción al `GenDict` y se setea la variable que indica que hay nuevo *coverage*. Esta última es la que se consulta al finalizar la ejecución de la secuencia para determinar si se debe incorporar o no al corpus.

3.1.7. *Update test*

En este último submódulo, revisamos los flujos posibles de cómo los *echidna tests* existentes son actualizados luego de cada secuencia, como se encuentra representado en la figura 3.9. Estos flujos están fuertemente atados al estado en el que el test se encuentra. Los estados posibles de un *echidna test* son:

- **Open**: un test se encuentra en este estado si todavía no se ha encontrado una secuencia de transacciones que lo haga fallar y todavía no se ha alcanzado el límite de máximos intentos (*openAttempts*) para romper dicho test configurado. Esto significa que se deberá chequear si la ejecución de las transacciones de la secuencia hasta el momento lograron romper este test (esto se chequea en la VM pero los cambios

que se puedan dar en el estado de la misma con dicho chequeo no se persisten). En caso de que sí, pasará al estado correspondiente guardándose como *reproducer* las transacciones ejecutadas hasta el momento.

- **Passed**: un test en estado *passed* indica que no se ha encontrado una secuencia que lo haga fallar pero alcanzó el límite de intentos disponibles. En este caso, se asume que hacer fallar al test es irresoluble por lo que no se necesita hacer ningún tipo de acción.
- **Large**: en este caso, se ha encontrado una secuencia de transacciones que hizo fallar a dicho test pero todavía se intenta reducir el tamaño de la secuencia porque todavía le quedan intentos de *shrinking* (*shrinkingAttempts*). En vez de intentar encontrar otra secuencia que también lo haga fallar, los esfuerzos para este tipo de test se dedican a intentar simplificar el *reproducer* original. Esto significa que se simplificará ya sea cada transacción o la secuencia en sí mediante la deleción de una de sus transacciones, buscando que esta nueva secuencia no sacrifique el hecho de que el *reproducer* logre que el test falle (se chequea en una VM independiente). En caso de que se pierda esta propiedad, no se harán cambios en el *reproducer* y se reducirá la cantidad de intentos disponibles.
- **Solved**: si se ha encontrado una secuencia que hizo fallar al test y además se ha simplificado lo más posible la secuencia que lo reproduce (ya no hay más *shrinkingAttempts* o la secuencia es de una sola transacción que ya no se puede simplificar), se considera que ya no queda nada por hacer con dicho test.

Con la información detallada de cada estado acompañada con el diagrama de flujo proveído, se puede reconstruir la máquina de estados de un *echidna test* en todo su ciclo de vida.

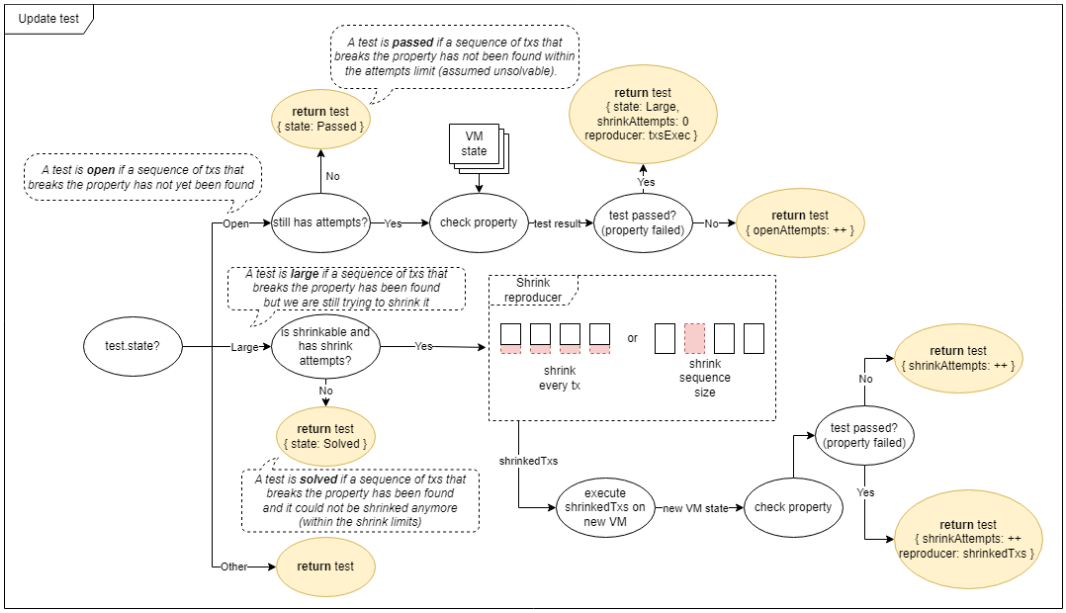

Fig. 3.9: Diagrama que representa la lógica que verifica luego de ejecutar una transacción si se logró hacer fallar algún test de Echidna.

En este capítulo pudimos entonces resumir los aspectos claves del algoritmo de Echidna, lo que nos permite plantearnos en qué puntos del mismo se pueden incorporar cambios con el objetivo de obtener una mejora en su funcionamiento. A su vez, provee una documentación intuitiva que permite a nuevos desarrolladores obtener un entendimiento más completo de Echidna de una manera mucho más simple y rápida, sin necesidad de revisar código.

4. EXTENSIÓN DE ECHIDNA

En este capítulo buscaremos determinar qué posibles mejoras podemos realizar en Echidna, teniendo en cuenta el funcionamiento de la herramienta analizado en el capítulo anterior. Una vez analizadas las diferentes opciones y elegida la mejora por realizar, describiremos los cambios llevados a cabo para implementarla.

Ya habiendo logrado tener un entendimiento profundo y completo de la herramienta elegida y considerando el *estado del arte* de las herramientas de *fuzzing* analizadas en el capítulo 2, surgieron diversas ideas de optimizaciones que creíamos podían aportar nuevo valor a Echidna. Las ideas consideradas fueron las siguientes:

- Agregar *built-in* oráculos a Echidna que permitan detectar distintos tipos de vulnerabilidades ya conocidas de *smart contracts*. Esto le permitiría a los usuarios contar con una base de propiedades simples y universales que testear sin tener que necesariamente definir properties o assertions custom, lo cual aceleraría el tiempo de adopción de la herramienta. Existen varias otras herramientas analizadas que cuentan con oráculos similares, como por ejemplo Smartian, que podrían servir como inspiración. Sin embargo, para poder implementar esta mejora, se requiere un mayor conocimiento acerca de cómo se manifiestan estas vulnerabilidades a nivel de ejecución de bytecode de la EVM además de la complejidad inherente de lograr generalizar las definiciones añadidas para que abarquen todas las posibles variantes de cada vulnerabilidad.

- Poder fijar una función o conjunto de funciones determinada de un contrato dado que siempre deba pertenecer a las secuencias de transacciones generadas y ejecutadas. Esto podría ser útil para casos en los que existe una de las funciones del contrato con mayor importancia o criticidad y que por lo tanto se desea que la misma sea testeada más exhaustivamente. En determinadas ocasiones en donde no se dispone de tanto tiempo para poder dedicarle a la campaña de *fuzzing*, esto podría ser especialmente ventajoso. Este cambio requeriría además agregar una validación extra post-mutaciones para garantizar que la función requerida no se vea suprimida por las mismas. Por ejemplo, dado un contrato A con funciones a, b y c, se podría querer testear más a fondo la función b (porque quizás es la más crítica o se sospecha que puede tener algún bug). En este caso, se podría fijar a la función b como requerida, de manera tal que las secuencias *[b,c,a,a,c]* y *[a,c,a,b,b]* son válidas pero la secuencia *[a,c,a,a,c]* no lo es. Este concepto puede extenderse ya sea a conjunto de funciones, donde el orden no importa, como a una subsecuencia de funciones que se quieren ejecutar consecutivamente.

- En la misma línea de restricción de las secuencias generadas, se podría determinar una relación de orden, ya sea parcial o total, entre un conjunto de las funciones a fuzzear. Este podría considerarse un caso más complejo de la idea anterior. Si bien permitiría expresar de esta forma ciertas precondiciones de cómo ejecutar las funciones de los contratos, podría estar interfiriendo en la capacidad del fuzzer de encontrar errores si no se cumplen dichas condiciones (estos casos suelen ser querer explotados por usuarios mal intencionados).

- Cambiar la herramienta para que utilice a alto nivel un algoritmo genético para obtener las secuencias de transacciones a fuzzear. Si bien podría suponer una mejora en los resultados obtenidos, es un cambio muy disruptivo para el estado actual de la herramienta y que en caso de desearse utilizar estas técnicas quizás sería mejor realizarlas desde cero o modificar un proyecto con una estructura más cercana a estas ideas.

- Basándonos en la idea de data flow analysis de Smartian, implementar un concepto similar de feedback en Echidna que pueda detectar los usos de las distintas variables y sus flujos asociados para poder ayudar al algoritmo de fuzzing. Esto requería un análisis estático y dinámico que permita definir cuáles son las variables definidas por el contrato y cuándo las mismas son usadas cuando se ejecutan las transacciones (esto implica ser capaz a partir del bytecode determinar las variables usadas, un conocimiento no trivial que se escapa a nuestra experiencia).

- Ser capaz de agregar condiciones que deben cumplirse para que las funciones definidas dentro de la *blacklist* o *whitelist* entren en efecto. Esto permitiría que una misma función pueda o no estar permitida en distintos momentos de la ejecución de la campaña dependiendo de las condiciones definidas, ampliándose así el poder de expresión del usuario para testear escenarios más complejos de setupear.

- Teniendo en cuenta las técnicas tradicionales de *fuzzing* y considerando que el *feedback* de Echidna actualmente es acotado, consideramos que se puede mejorar el resultado de la herramienta si se incorpora un método de *feedback* basado en *boosted greybox fuzzing* (sección 1.3.3).

Analizando las opciones dadas, decidimos en esta investigación llevar a cabo la última mejora propuesta. Esta decisión fue tomada teniendo en cuenta principalmente el posible impacto en la *performance* de la herramienta y la viabilidad del cambio a introducir así como también nuestro interés en analizar el verdadero impacto de un método dado de *feedback* en la capacidad de la herramienta de explorar nuevos caminos.

Recordemos que el feedback utilizado por Echidna se basa en otorgarle mayor energía a los nuevos inputs que se hayan agregado al corpus, sin importar qué tan poco frecuente resulta el camino hallado por el input. Si bien consideramos que esto se trata de una heurística que considera que nuevos caminos explorados tienen mejores oportunidades para explorar nuevos caminos, entendemos que esto no garantiza que un camino poco frecuente descubierto no termine siendo ignorado en el proceso. Por ejemplo, si un input genera un determinado camino poco frecuente y difícil de alcanzar al principio de la ejecución y posteriormente se siguen añadiendo seeds que exploran otros caminos (más simples y por tanto fácil de encontrar), estos últimos tendrán más posibilidades para generar nuevos inputs dejando relegado al camino inicial independientemente de si el mismo exploraba un camino difícil de alcanzar. Por otro lado, no encontramos ninguna otra herramienta del *estado del arte* que utilice una heurística similar ni garantías sobre la efectividad de la misma. Esto nos llevó a cuestionarnos este tipo de *feedback* y plantearnos si otras técnicas tradicionales más utilizadas y comprobadas podrían mejorar la efectividad de Echidna, como lo es *boosted greybox fuzzing*. Creemos que este método de asignar energías puede resultar en una más rápida y quizás mejor exploración del árbol de cómputo pudiendo lograr un mejor *coverage* en líneas de código que requieren múltiples condiciones para poder ser alcanzadas.

Como también existe la posibilidad de que en este contexto de aplicación a smart contracts el feedback utilizado no influya de manera tangible en el resultado final, también decidimos implementar una versión de la herramienta que le otorgue la misma energía a todos los elementos del corpus, constituyéndose así una técnica de *greybox fuzzing* tradicional.

Es decir, contaremos con tres versiones de Echidna que buscaremos comparar: la original, una con pesos equivalentes del corpus (que llamaremos *random*) y una con energías proporcionales a las frecuencias de los caminos explorados (que llamaremos *EchidnaAFL*). Con esto buscaremos además tener una mayor comprensión acerca de la influencia del *feedback* utilizado en el contexto de *fuzzing* de *smart contracts*.

Exploremos en qué consiste la implementación de cada una de las versiones añadidas y los cambios necesarios para llevarlas a cabo.

4.1. Implementación

4.1.1. EchidnaAFL

La idea de la implementación de **EchidnaAFL** se basa en extrapolar el algoritmo de *feedback* de la herramienta AFL Fast [12] para testeo de funciones al contexto de secuencias de funciones de *smart contracts*. AFL Fast implementa un algoritmo de *boosted greybox fuzzing* en donde la energía de cada *input* semilla es inversamente proporcional a la frecuencia de exploración de ese camino por todos los inputs (visto en 1.3.3). Es decir, aquellos inputs cuyos caminos de ejecución en el código hayan sido poco explorados y por tanto poco frecuentes, tendrán una mayor energía y por tanto una mayor probabilidad de ser elegidos al momento de mutar un nuevo input.

Ahora bien, en el caso de AFL Fast, un input es el conjunto de parámetros que toma la única función siendo testeada y en este contexto se entiende como camino al único recorrido en el árbol de cómputo resultante de ejecutar la función evaluada en dicho conjunto de parámetros. Sin embargo, en nuestro contexto de aplicación el input es una secuencia de transacciones con sus respectivos parámetros por lo que debemos poder definir qué consideramos el camino de la secuencia en este contexto para así poder implementar el algoritmo análogo con esta nueva definición de input y camino.

En particular, a la hora de definir cómo representar el camino de una secuencia, consideramos distintas opciones.

Como primera opción, podemos pensar a un camino como la sucesión de instrucciones que se ejecutan en cada transacción de la secuencia, tomadas secuencialmente en el mismo orden que se encuentran en la misma y sin segmentar por transacción. En este contexto, representaremos las instrucciones ejecutadas como los *program counters* (***PCs***) de cada *step* a nivel del *bytecode* ejecutado en la EVM. Un ejemplo de cómo se vería esta representación podemos verlo en el gráfico 4.1, donde mostrarmos para una secuencia ejemplo cómo sería su camino respectivo con esta opción bajo la leyenda *Opción 1*. La desventaja de esta representación es que cualquier cambio en el orden o la longitud de una secuencia resultará en otra representación de camino incluso si cubren el mismo código. Es decir, que es poco probable que dos secuencias coincidan en su camino y por tanto la magnitud de la frecuencia de cada elemento del corpus tenderá a ser baja y a la vez la cardinalidad de cantidad de caminos ejecutados será muy alta sin que esto impacte en los elementos guardados en el corpus que sólo se rigen por la cobertura o no de nuevo código. Esto podría

resultar en que las frecuencias terminen no influenciando en gran medida a la energía de los elementos del corpus.

Otra opción de representación es utilizar como camino al conjunto no ordenado de PCs cubierto por las distintas transacciones que componen una secuencia dada. Podemos ver la representación de esta opción en el diagrama 4.1, bajo el título *Opción 2*. Esto disminuiría la cardinalidad del universo de caminos posibles y por tanto aumentaría la probabilidad de que dos secuencias compartan representación de camino (obteniéndose así valores mayores de frecuencia para los elementos del corpus). En este caso, si dos secuencias cambian el orden de sus transacciones pero exploran el mismo código entonces compartirán la misma representación de camino. Sin embargo, siguen existiendo casos de secuencias ejecutadas cuyos caminos no serán representados por ningún elemento del corpus y su ejecución no contribuirá al *feedback*. Para entender mejor por qué sucede esto, consideremos un corpus con dos elementos con sus respectivos caminos: {{1,2,3}, {3,4,5}}. Ambas secuencias se encuentran en el corpus ya que añadieron nuevo *coverage*. Ahora consideremos que ejecutamos otra secuencia cuyo camino es {2,3,5}. Si bien no añade *coverage* y por tanto no será añadido al corpus, tampoco comparte camino con ninguno de los actuales elementos por lo que no actualizará la frecuencia de ninguno de ellos.

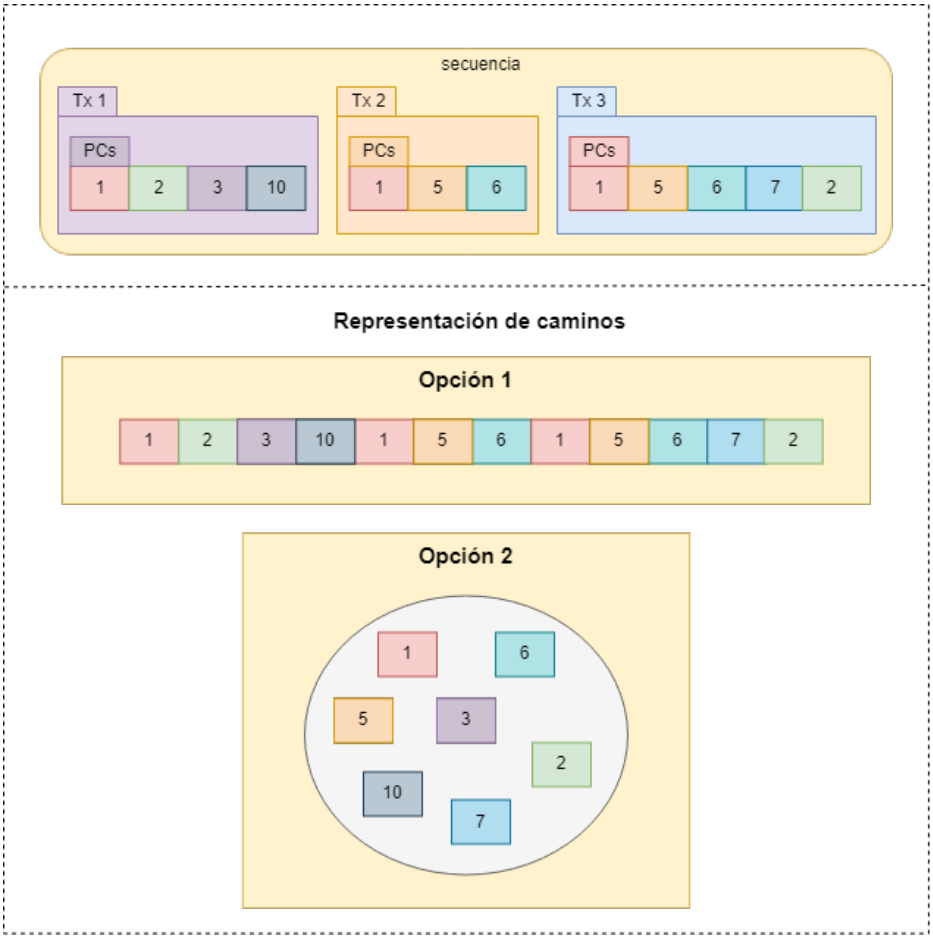

Fig. 4.1: Diagrama que representa gráficamente dada una secuencia de transacciones las dos opciones de representación de camino consideradas.

Para ambas opciones, podemos ver que existen una gran cantidad de secuencias con caminos no representados en el corpus. Como esto supone que existan secuencias que no aporten nueva información al *feedback* de la herramienta, una solución a esto podría ser cambiar la condición de inclusión en el corpus para que en vez de agregarse una secuencia al mismo si se detecta nuevo *coverage*, se añada si la secuencia representa un nuevo camino no explorado previamente. Si bien esto soluciona el problema de existencia de caminos que no se encuentran asociados a ningún elemento del corpus, en el caso de la primera opción la cardinalidad del corpus se verá incrementada considerablemente sin que esto necesariamente implique un mejor *feedback* y además, la probabilidad de que dos secuencias compartan el mismo camino se mantiene baja. Para el segundo caso, aumentará moderadamente la cantidad de elementos del corpus. El problema con esta solución es que se aleja de la definición tradicional del algoritmo de *fuzzing* y en particular la implementada por AFL Fast. Es por esto que decidimos no cambiar la condición de inclusión al corpus para mantener la cantidad de cambios introducidos en Echidna al mínimo y poder evaluar la efectividad de las modificaciones lo más independientemente posible.

Ahora bien, entre la primera y la segunda opción de representación camino, optamos por la segunda ya que suponía un *feedback* más apropiado con menos desventajas (menos ejecuciones que no contribuyen al algoritmo).

Vale la pena mencionar que también consideramos guardar información de frecuencias por transacción en vez de por secuencia para que influya en el *feedback* utilizando una suma sopesada de la frecuencia de cada una de sus transacciones. Esto incluso si se quiere se podría combinar con la definición ya mencionada por secuencia para calcular la energía utilizando una combinación lineal de ambas opciones. Si bien esto constituye una idea interesante, no se podía considerar de forma aislada sin primero llevar a cabo una interpretación más directa del concepto de AFL Fast. Es por esto que no profundizaremos en ella dentro del marco de esta tesis ya que supone una aplicación más compleja. Sin embargo, la tendremos en cuenta para futuros trabajos porque no carece de valor.

Una vez determinada la representación de camino elegida, debemos realizar los cambios pertinentes en Echidna. Por un lado, necesitamos una estructura global que contenga las frecuencias para los distintos caminos y que se actualice con cada nueva ejecución de una secuencia. Además, para actualizarla debemos registrar los PCs cubiertos por la secuencia actual a medida que ejecutamos las transacciones que la componen. Por otro lado, debemos añadir a los elementos del corpus el camino que lo representa para así poder recuperar la frecuencia correspondiente. Y por último, tenemos que actualizar la manera en la que se obtienen elementos del corpus para que se utilice el nuevo cálculo de la energía y se elija en base a ella.

Para lograr llevar un registro del camino de la secuencia actual, definimos el siguiente tipo que representa el camino entendido como el conjunto de PCs cubiertos por contrato (representado con su *bytecode*):

```
type SequenceCoverage = Map ByteString (Set PC)
```

Una instancia de este tipo es guardada en el estado global de la campaña (llamada `currentSequenceCoverage` en el `CampaignState`) y es actualizada con el PC de cada *step* ejecutado de cada transacción de la actual secuencia. Al finalizarse la ejecución de la secuencia, este `currentSequenceCoverage` se utiliza para actualizar la frecuencia correspondiente y se resetea el valor en el estado global para que en la próxima secuencia empiece vacía.

En cuanto a las frecuencias, agregamos al estado global de la campaña una instancia `corpusCoverageFrequences` del siguiente tipo:

```
type CorpusCoverageFrequences = Map Hash Int
```

La misma es actualizada utilizando el hash del `SequenceCoverage` de la secuencia actual e incrementándose la frecuencia asociada al mismo. En caso de que el hash no haya existido previamente, se insertará en el mapa con un valor de 1.

Los elementos del corpus, por otra parte, se definen de la siguiente forma:

```
-- Corpus original
type Corpus = Set (Int, [Tx])

-- Corpus modificado
type Corpus = Set (Hash, [Tx])
```

De esta manera, cuando una secuencia cubre nuevo *coverage* y debe ser añadida al corpus, se utiliza el hash de su `SequenceCoverage` para guardarlo junto con la secuencia de transacciones correspondiente en vez del anterior valor incremental utilizado como energía. Decidimos utilizar un hash en vez del `SequenceCoverage` directamente para optimizar el uso de la memoria de los cambios introducidos, ya que el conjunto de PCs ejecutados por una secuencia podría ser considerablemente grande. Si bien existe la posibilidad de que haya colisiones en base a la implementación de la función de hash utilizada, la probabilidad es ínfima y en caso de que suceda no debería afectar en demasía el funcionamiento del algoritmo (se superpondrían las frecuencias de las secuencias colisionadas).

Teniendo en cuenta este último cambio, a la hora de tomar elementos del corpus ya no se puede utilizar directamente el peso guardado como energía si no que debemos calcular la misma utilizando el hash registrado para obtener la frecuencia guardada en `corpusCoverageFrequences` y realizar el cálculo correspondiente. El cálculo de las energías se implementa de la siguiente forma en `Corpus.hs`:

```
computeEnergies :: CorpusCoverageFrequences ->Corpus ->[([Tx], Rational)]
computeEnergies corpusCovFreq = map ((hash, txs) ->
    (txs, 1 / fromIntegral (findWithDefault 0 hash corpusCovFreq))) . Set.toDescList
```

Un punto importante a tener en cuenta es que los cambios y estructuras adicionales introducidas suponen un *overhead* tanto espacial como temporal en el funcionamiento de la herramienta, ya que añadimos estructuras que son mantenidas en memoria y el nuevo cálculo de energías cuando se obtienen elementos del corpus.

Con estos cambios mencionados, logramos implementar la versión *EchidnaAFL* (la totalidad de las modificaciones puede ser consultada en [29]).

4.1.2. Random

Como mencionamos anteriormente, la idea detrás de la implementación de esta versión *random* es poder utilizarla como punto base de comparación con las otras dos versiones de Echidna ya que es la asignación de energías más básica que se puede tener.

Para poder llevar a cabo la implementación, debemos cambiar los pesos de los elementos del corpus para que en lugar de ser incrementales sean constantes para todas las secuencias. Manteniendo la definición de corpus de la versión original (en donde cada secuencia tiene asignado un entero que representa su peso), asignamos a todos los elementos que se añaden al mismo un peso con valor 1. Esto garantiza que a la hora de elegirse secuencias del corpus, todas tendrán la misma probabilidad de ser elegidas.

Debido a la simplicidad de esta propuesta, no se genera ningún *overhead* de implementación en comparación con la herramienta original.

En conclusión, tenemos tres versiones de Echidna cada una con una estrategia distinta de cálculo de energía para la elección de elementos del corpus. Como ayuda visual, podemos ver en la figura 4.2 una representación gráfica de cómo cada versión almacena los elementos del corpus y cómo se le asocia su energía correspondiente.

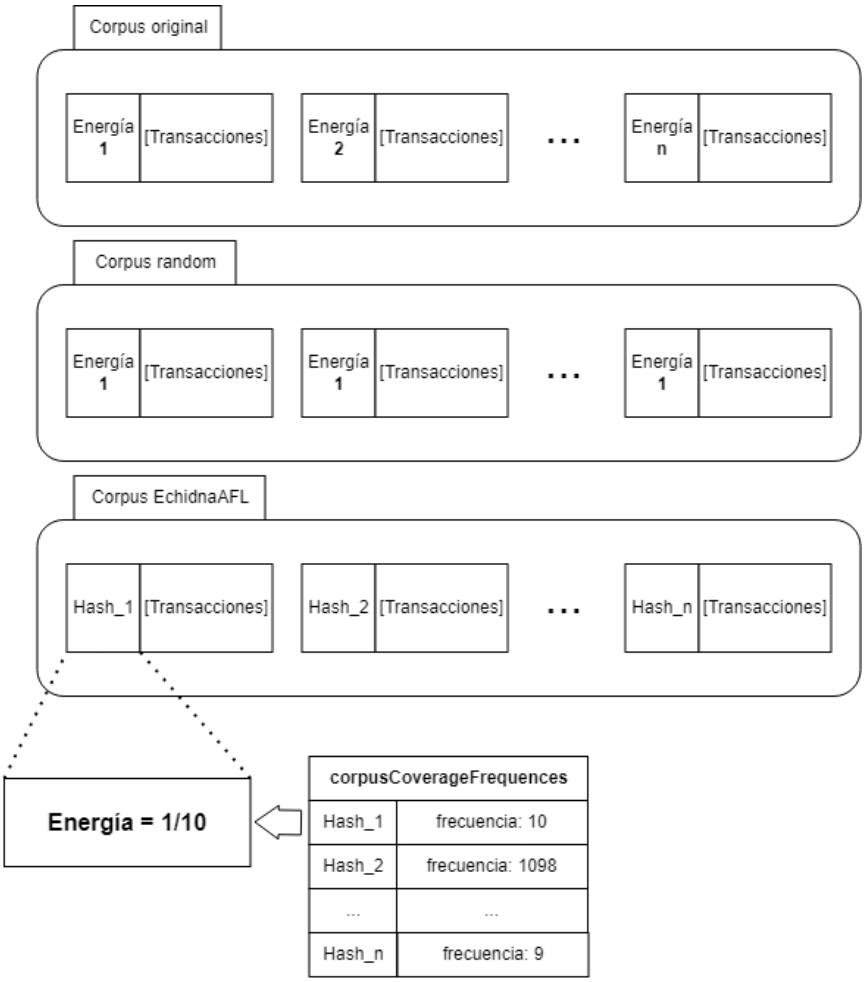

Fig. 4.2: Diagrama que representa gráficamente el almacenamiento de los corpus de cada versión de Echidna implementadas (original, EchidnaAFL y random).

En el capítulo siguiente llevaremos a cabo una experimentación con el fin de comparar estas versiones y poder responder nuestras preguntas de investigación.

5. EVALUACIÓN

La introducción de las distintas versiones de Echidna detalladas anteriormente incita a cuestionarse cómo las mismas se relacionan entre sí y qué conclusiones seremos capaces de obtener acerca de la influencia de la elección de corpus a partir de estas comparaciones.

Esto supone no sólo la definición de qué preguntas deseamos abordar sino también la definición de un marco de experimentación que nos permita poder respondernos dichas preguntas, con una diversidad de casos de prueba que nos otorgue la confianza necesaria para poder asegurar si una versión es superior a otra y en qué condiciones.

5.1. Preguntas de investigación

Las siguientes preguntas son aquellas que nos surgieron y que buscaremos abordar en nuestra investigación:

1. ***¿Podemos afirmar que EchidnaAFL supone una mejora en comparación con la versión original? ¿Bajo qué premisas de uso esto sucede?***
2. ***¿Cuál es el overhead de tiempo asociado a EchidnaAFL?***
3. ***El método de elección de corpus propuesto por Echidna, ¿es una buena heurística para mejorar la exploración de código?***
4. ***¿Influye de manera relevante la elección de elementos del corpus en Echidna?***
5. ***¿Resulta aplicable al caso de fuzzing de smart contracts en el contexto de Echidna la técnica de feedback de AFL Fast?***
6. ***¿Se trata de una buena adaptación del feedback de AFL Fast la representación de camino elegida?***

5.2. Casos de estudio

Para poder realizar una comparación lo más justa posible entre las distintas versiones a analizar, decidimos elegir tres distintos *benchmarks* de *smart contracts*. Cada uno constituirá un caso de estudio distinto, en donde se contará con distintas características que los distinguen entre sí. Esto nos proveerá un marco variado de análisis que buscará contestarnos las preguntas de investigación planteadas; por ejemplo, bajo qué condiciones de contratos una versión se considera mejor que otra.

5.2.1. Setup general

Todos los *benchmarks* que describimos más adelante contaron con una misma configuración inicial y fueron ejecutados en las mismas condiciones de *hardware*. Además, todos los casos de estudio tuvieron que ser adaptados para ser soportados por Echidna (y las versiones de Solidity que Echidna requiere).

En lo que compete al *hardware* utilizado, las especificaciones de la computadora en donde se llevaron a cabo las pruebas son las siguientes:

- **RAM**: 16 GB
- **CPU**: Intel Core™ i7-7700 CPU @ 3.60GHz × 8
- **GPU**: Mesa Intel® HD Graphics 630 (KBL GT2)
- **OS**: Ubuntu 22.04.1 LTS 64-bit

Además, la PC mencionada fue dedicada exclusivamente a la ejecución de pruebas para evitar que otros procesos intermitentes afecten los resultados.

En cuanto al método de ejecución de los distintos *benchmarks*, seguimos el siguiente lineamiento:

- Para todas las experimentaciones, cada versión y test de prueba fue ejecutado diez veces, obteniéndose el promedio de todas las iteraciones para analizar el resultado.
- Utilizamos $testLimit = 2{,}000{,}000$ para las campañas de *fuzzing* ejecutadas, lo que significa que cada una de ellas genera 2 millones de secuencias para testear. La razón por la que utilizamos esta configuración y no un límite de tiempo es que, dado el *overhead* que trae aparejada la versión EchidnaAFL, queríamos que todas las versiones estuviesen en igualdad de condiciones en lo que respecta a la capacidad de exploración del algoritmo de *fuzzing* correspondiente en la misma cantidad de secuencias generadas en base al *feedback*. En caso de encontrar que EchidnaAFL performa mejor en estas condiciones, se puede analizar qué mejoras se pueden llevar a cabo para reducir aún más su *overhead*. Por otro lado, elegimos el número necesario para que, en el hardware disponible, cada iteración de campaña de fuzzing dure aproximadamente 1 hora. Esto fue así ya que en este punto es donde observamos un estancamiento en el crecimiento de cobertura de código utilizando como muestra una breve experimentación con un caso de test simple. Para que el límite de tiempo no interfiriese, definimos $timeLimit = 36{,}000$, que equivale a 10 horas.
- Para $seqLen$, consideramos los valores 10, 100 y 200 para determinar si el cambio del tamaño de las secuencias generadas era un factor determinante. Estos fueron elegidos teniendo en cuenta que buscábamos contar con un rango de valores que fuese desde el mínimo de transacciones posible que todavía permitiese realizar los suficientes cambios de estado para representar casos de uso interesantes hasta secuencias de gran cantidad de transacciones, utilizando 100 y 200 para representar este grupo. No utilizamos $seqLen = 1$ (que sería equivalente a sólo una transacción) ya que en muchos casos se requiere la ejecución de múltiples métodos para lograr un cambio de estado en las variables del contrato, que a su vez permiten mayor exploración de código.
- Decidimos desactivar la funcionalidad de *shrinking* (es decir, utilizamos $shrinkLimit = 1$) ya que la misma no cambia la exploración alcanzada ni los elementos del corpus pero sí supone más tiempo de cómputo en la búsqueda de lograr secuencias más acotadas.

- Como puntos de referencia para la comparación de resultados, utilizamos además de la cantidad de *echidna tests* que fallan en función de la cantidad de secuencias creadas, la variación de los *coverage points* a través de las secuencias creadas. Estos son una medida de la cantidad de líneas cubiertas en los contratos por testear (a mayor valor de *coverage points*, mayor cantidad de líneas abarcadas). La razón por la que consideramos esto es ya que el número de cantidad de *echidna tests* rotos suele ser un valor chico por lo que su variación implica un gran porcentaje de cambio en los resultados. En cambio, los *coverage points* cuentan con una mayor granularidad y suponen una representación más fiel de la exploración de código obtenida en un momento dado ya que la cobertura de una nueva línea no implica necesariamente lograr romper un nuevo test.

Para el resto de las configuraciones disponibles en Echidna, se dejaron determinados los valores *default*. Vistas ya las configuraciones generales utilizadas para todos los casos de estudio, pasaremos a caracterizar las particularidades de cada *benchmark* utilizado.

5.2.2. Contratos autogenerados: *Maze Benchmark*

Se trata de un conjunto de contratos generados automáticamente (obtenidos por la herramienta Daedaluzz [32]) que tienen como característica la abundancia de "*ifs*"nesteados con condiciones aritméticas que dependen de múltiples variables y del estado del contrato. Esto hace que las funciones no dependan sólo de las variables de la transacción sino también de ejecuciones anteriores que modifiquen el estado del contrato.

Este tipo de contratos nos pemite testear a las distintas versiones en un contexto en el que existen muchos puntos del código difíciles de alcanzar debido a la dificultad de las sucesivas condiciones por lo que nos da una idea de la capacidad de exploración de cada una de ellas. No obstante, al tratarse de contratos generados artificialmente, no necesariamente refleja un caso de uso o una aplicación realista de la herramienta.

El *benchmark* elegido se encuentra compuesto por cuatro contratos autogenerados (*maze* 1, 2, 3 y 4). En este caso, para utilizarlo con Echidna cambiamos todos las *assertions* por *flags* que comienzan siendo verdaderas y para luego ser seteadas en falso en cada lugar designado. Por cada una de estas *flags* agregamos una *echidna property* correspondiente, de manera de poder correr Echidna en modo *property*.

5.2.3. Contratos básicos: *Smart-Pulse Benchmark*

Como el caso de estudio anterior consistía de contratos complejos pero artificiales, decidimos agregar otro *benckmark* con características opuestas; es decir, de casos de uso realistas pero simples. Estos fueron extraídos de los *benchmarks* presentes en el repositorio de la herramienta Smart Pulse [33]. Los contratos elegidos fueron: *Auction*, *Crowfunding* y *RockPaperScissors*. El primero busca representar la interfaz que una subasta online podría llegar a tener, en donde se puede pujar y retirar fondos invertidos; el segundo, implementa una financiación colectiva con un tiempo límite en el que se pueden realizar donaciones; mientras que el último, simula un juego de piedra, papel o tijera. Para otorgar una noción de su simplicidad, el promedio de líneas de los contratos elegidos ronda las 67 líneas.

La elección de dichos contratos se basó en descartar aquellos que no fuimos capaces de compilar y adaptar a Echidna (entre otras razones, algunos dependían de RPCs[1] que si bien

[1] https://en.wikipedia.org/wiki/Remote_procedure_call

se encuentran soportados por la herramienta, su uso puede afectar considerablemente los tiempos de ejecución) y modificar los restantes para el uso de la herramienta. En este caso, debimos alterar los métodos constructores de los contratos para que los mismos no tomaran parámetros ya que esto no se encuentra soportado por Echidna. Para ello, seteamos valores *default* razonables dentro del caso de uso para cada uno de los parámetros eliminados.

Este *benchmark* fue ejecutado utilizando el modo *exploration* de Echidna, modo en el cual no se busca romper *echidna tests* sino obtener el mayor *coverage* posible dentro de las restricciones de secuencia y tiempo configuradas.

5.2.4. Contratos de aplicación real: *Uniswap Benchmark*

Por último, nos interesaba contar con un conjunto de contratos de casos de uso reales, para poder testear a las distintas versiones de la herramienta en un ejemplo más sofisticado. Para ello, utilizamos los contratos que se pueden encontrar en Uniswap v3[34], en particular aquellos que ya disponían de una versión adaptada para ser utilizada por Echidna. Estos son: `BitMath`, `FullMath`, `LowGasSafeMath`, `Oracle`, `SqrtPriceMath`, `SwapMath`, `Tick`, `TickBitmap`, `TickMath`, `TickOverflowSafety` y `UnsafeMath`. Como sus nombres lo indican, se encuentran más orientados a poder testear las lógicas de operaciones aritméticas bajo distintos contextos. Todos ellos buscan testear casos interesantes de sus respectivas librerías, que se encuentran disponibles para su uso genérico por parte de la comunidad.

En este caso, como la adaptación ya estaba realizada, sólo debimos correr los *benchmarks* con el modo correcto, *assertion*.

5.3. Análisis y resultados

Teniendo en cuenta las diversas preguntas de investigación que deseamos explorar, antes que nada debemos definir un eje de investigación que nos permita dirigir nuestro análisis. Con esto en mente, decidimos abordar en primer lugar la pregunta de si podemos o no afirmar que EchidnaAFL supone una mejora con respecto a su versión original, ya que entendemos que se trata de un cuestionamiento inherente a la acción de implementar una modificación a una herramienta. Este será entonces nuestro primer eje de análisis.

5.3.1. ¿Podemos afirmar que EchidnaAFL supone una mejora en comparación con la versión original?

En primer lugar, presentamos los resultados obtenidos en el *Maze Benchmark* en la comparación de las dos versiones a analizar en la tabla 5.1. En la misma, vemos un resumen de qué versión fue la ganadora teniendo en cuenta los coverage points obtenidos para las distintas combinaciones de *seqLen* y *maze*. Además, incluimos un porcentaje de cuántas veces cada algoritmo resultó el ganador por *seqLen* o por *maze*.

Como primera observación, podemos decir que no parece haber ninguna versión que se destaque por sobre la otra de una manera contundente; de hecho, los casos en los que gana una versión o la otra están divididos exactamente a la mitad en los distintos escenarios considerados. Sí podemos notar una tendencia por *seqLen*, en la cual a mayor longitud de secuencias generadas hay un porcentaje creciente de victorias para *EchidnaAFL*.

Ahora bien, si volcamos nuestra atención sobre las tendencias que se pueden ver por contrato, podemos decir que pareciera existir una diferencia de qué versión se destaca dependiendo el *maze* ya que por ejemplo tanto para el *Maze 1* como el *Maze 2*, hay una

versión que gana la totalidad de las veces. Sin embargo, esto no es tan evidente en los otros dos *mazes*. Teniendo en cuenta esto, decidimos analizar el código de los contratos en los que notamos una predilección por una versión específica, en búsqueda de alguna característica que las diferencie. No obstante, no pudimos encontrar ninguna referente a la distribución de la profundidad de las condiciones anidadas o longitud de código utilizado. Estas tendencias entonces pueden o estar ligadas a características no distinguibles a simple vista o dado que las mismas son en base a tres casos, puede no existir una causalidad ligada a estas.

	Maze 1	*Maze 2*	*Maze 3*	*Maze 4*	*% afl*	*% original*
SeqLen 10	afl	original	original	original	25	75
SeqLen 100	afl	original	afl	original	50	50
SeqLen 200	afl	original	afl	afl	75	25
% afl	100	0	66	33		
% original	0	100	33	66		

Tab. 5.1: Tabla comparativa sintetizadora de los resultados obtenidos para las versiones original y EchidnaAFL, para todas las combinaciones posibles de seqLen y mazes

Una vez hecho este primer análisis, queremos profundizar y expandir sobre los resultados obtenidos y las tendencias observadas. Para ello, graficaremos los valores de *coverage points* y de *tests* alcanzados observados, agrupando tanto por *seqLen* como por contrato *maze*. Nuestro objetivo es ser capaces de analizar la magnitud de las diferencias observadas, tanto entre las versiones a comparar como, para una misma versión, entre las distintas combinaciones de los parámetros variados.

En las figuras 5.1 y 5.2, vemos los valores de *coverage points* y *tests* alcanzados respectivamente, agrupados en este caso por *seqLen* de cada ejecución. A primera vista, podemos decir que entre las distintas versiones para cada una de las ejecuciones la diferencia observada tanto de *coverage points* como de *tests* descubiertos es muy pequeña (de hecho, sólo existe una diferencia promedio del 0,5 % de coverage points entre las dos versiones). Vale aclarar que además tanto la cantidad de *tests* como los *coverage points* obtenidos en cada caso no son los máximos obtenibles, por lo que todavía quedaban líneas que cubrir y *tests* por romper. La conjunción de estas observaciones pareciera indicar que las diferencias observadas de victorias de una versión por sobre la otra no son un fuerte indicador de dominancia de alguna estrategia, sino que más bien parece mostrar que ambas versiones se encuentran al mismo nivel en cuanto a exploración de código y explotación de *tests*. Esto refuerza lo observado en la tabla 5.1 en la equivalente cantidad de victorias de cada versión. Esto pareciera indicar que el ganador en cada caso está más relacionado a la aleatoriedad intrínseca de generación de *inputs* y por tanto de exploración de caminos más que a la superioridad de un algoritmo por sobre otro.

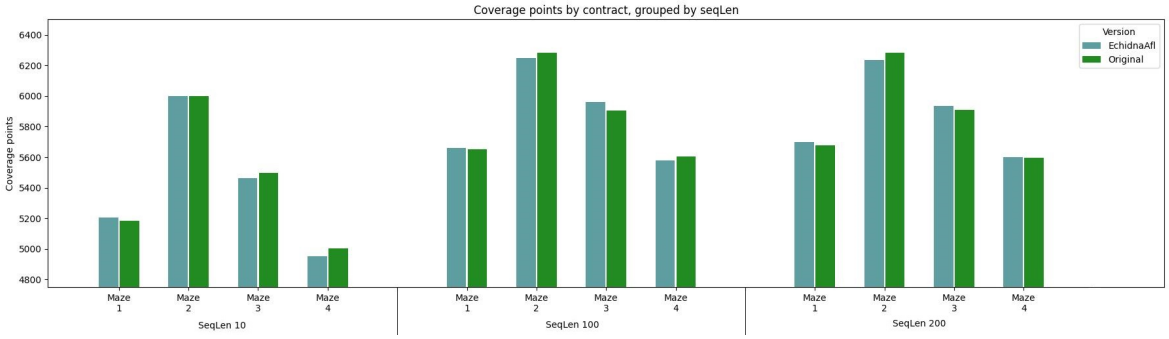

Fig. 5.1: Gráfico de coverage points por contrato del benchmark Maze agrupado por seqLen del promedio de las ejecuciones correspondientes, para las versiones original y EchidnaAFL

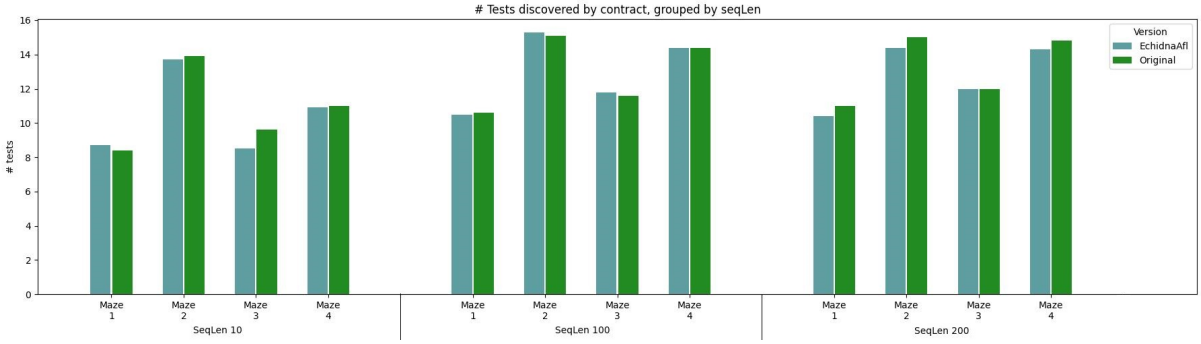

Fig. 5.2: Gráfico de cantidad de tests rotos por contrato del benchmark Maze agrupado por seqLen del promedio de las ejecuciones correspondientes, para las versiones original y EchidnaAFL

Podemos seguir viendo en la figura 5.1 que a mayor *seqLen*, *EchidnaAFL* resulta victorioso más veces lo cual podría significar la existencia de una tendencia. Sin embargo, la magnitud de la diferencia en *coverage points* sigue siendo no significativa (es menor que la diferencia obtenida por ejemplo entre distintas ejecuciones del mismo contrato con distintas *seqLen*) y al mismo tiempo al no contar con ejemplos de mayor *seqLen*, no podemos confirmar dicha tendencia.

Por otro lado, en el gráfico 5.2, resaltan los valores de la diferencia entre los resultados obtenidos para ambas versiones, que siempre resultan por debajo de 1. Esto quiere decir que, en promedio, las versiones no logran diferenciarse lo suficiente entre sí para suponer el descubrimiento de 1 test o más entre ellos. Sí resulta curioso que la versión original resulte ganadora en un mayor porcentaje en comparación con los resultados obtenidos según cobertura de líneas. Nuestra intuición es que esta diferencia se encuentra relacionada con la naturaleza de la exploración de código de cada versión. La versión original le otorga más energía a los últimos *inputs* explorados por lo que podría tender a dedicar más esfuerzo en recorrer un conjunto de condiciones anidadas adquiriendo cada vez más profundidad. Este tipo de exploración es similar a la estrategia DFS (*Depth-First Search*) de búsqueda de árboles o grafos. En cambio, *EchidnaAFL* al asignar energía en base a la frecuencias de los caminos recorridos por los inputs, distribuye más el esfuerzo dedicado a todas las condiciones ya que a medida que se va explorando más una rama, su frecuencia aumenta y

por tanto su energía baja permitiendo que se sigan explorando otros caminos. Este enfoque se asemeja más a una estrategia BFS (*Breadth-First Search*) de exploración de estructuras de tipo árbol. Estos dos acercamientos distintos pueden ser la causa de la diferencia en resultados si consideramos *coverage points* versus *echidna tests*, ya que ya que en este tipo de contratos los *flags* que logran romper los *echidna tests* definidos se encuentran al final de cada conjunto de condiciones anidadas.

En cuanto a la comparación entre gráficos, también queremos mencionar que no parece existir una correlación entre las victorias de las versiones en base a *coverage points* y en base a *tests* fallidos. Esto quiere decir que existen casos donde un mayor *coverage* no implica una mayor cantidad de *tests* alcanzados, lo cual tiene cierto sentido ya que debido a la naturaleza de los contratos que presentan muchas condiciones anidadas, la cobertura de nuevas líneas no supone un nuevo test quebrantado si no se alcanza la condición más profunda.

Por último, en lo que respecta al gráfico de barras 5.1, notamos que las muestras de *seqLen = 10* se encuentran por debajo de las observadas para *seqLen* mayores. Esto nos lleva a explorar mejor esta relación entre distintos valores de *seqLen*, por lo que en las figuras 5.3 y 5.4 podemos ver los mismos resultados pero esta vez agrupados por *maze*.

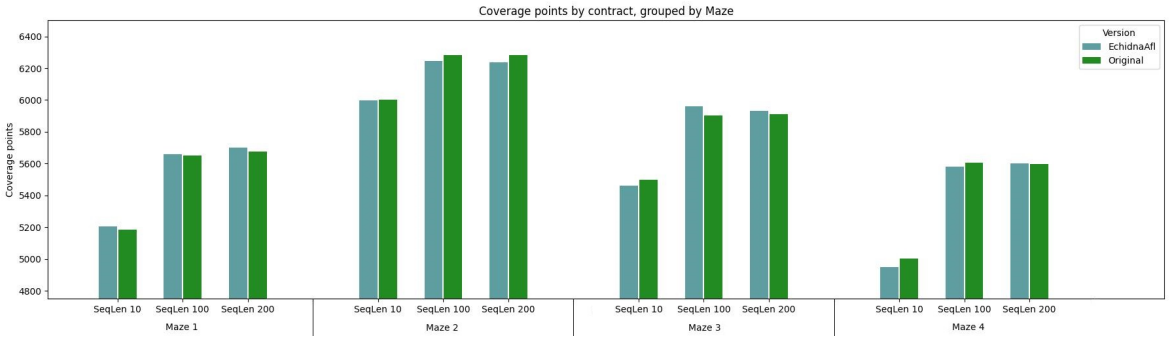

Fig. 5.3: Gráfico de coverage points agrupados por contrato del benchmark Maze para las distintas seqLen de los promedios de las ejecuciones correspondientes, para las versiones original y EchidnaAFL.

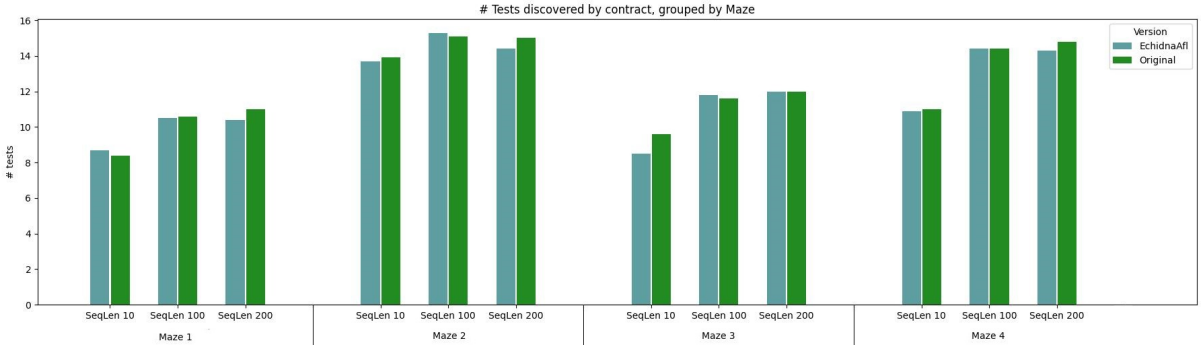

Fig. 5.4: Gráfico de cantidad de tests rotos agrupados por contrato del benchmark Maze para las distintas seqLen de los promedios de las ejecuciones correspondientes, para las versiones original y EchidnaAFL.

Si bien en el gráfico 5.3 podemos ver la tendencia de victorias de versión por *maze*

mencionada anteriormente, esta representación nos permite ver que las diferencias que consolidan dichas victorias no parecen ser significantes. Esto podría explicar por qué no fuimos capaces de reconocer características en cada *maze* que expliquen esta predisposición. Esto nos permite por el momento decidir que no existe una tendencia concluyente de victorias de cada versión en determinados contratos, en lo que a este *benchmark* respecta.

Por otro lado, en la figura 5.3, podemos ratificar que a mayor *seqLen* parece haber un mejor *coverage* de los contratos ya que se puede ver este comportamiento ascendente en todos los mazes. Sin embargo, si bien la diferencia entre los resultados de *seqLen = 10* y *seqLen = 100* son significativos, no es así para las muestras de *seqLen = 100* y *seqLen = 200*. Esto parece indicar que esta mejora observada a partir de la longitud permitida de las secuencias se estanca y el valor óptimo podría estar en algún lugar entre 100 y 200. También es posible que el mismo se encuentre ligado al tipo de contrato que se analiza y los cambios de variables necesarios para llegar a un estado global que permita la exploración de nuevo código.

Para visualizar mejor cómo afecta el *seqLen* elegido no sólo el resultado obtenido sino también el crecimiento de cobertura obtenida a lo largo de toda la campaña de exploración, graficamos en la figura 5.5 los *coverage points* en función de la cantidad de secuencias creadas a lo largo de la campaña para las ejecuciones de la versión *EchidnaAFL* de distintas *seqLen* del contrato *Maze 1*.

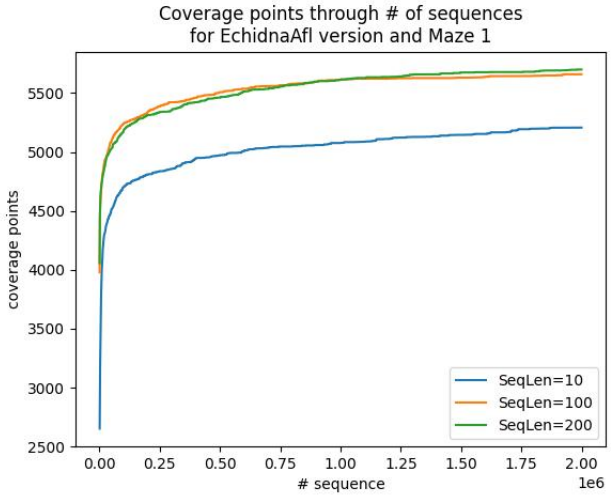

Fig. 5.5: Gráfico de comparación de coverage points en función de la cantidad de secuencias creadas para las distintas seqLen de los promedios de las ejecuciones de la versión EchidnaAFL y el Maze 1.

Podemos ver que si bien la curva de *seqLen = 10* se encuentra por debajo de las otras dos, todas poseen la misma curvatura con un punto de inflexión en su crecimiento que sucede en el mismo momento. Las curvas de *seqLen = 100* y *seqLen = 200* están mayormente solapadas por lo que se confirma que la diferencia de este parámetro entre 100 y 200 no resulta en una mejora en la exploración de código. Más allá de esto, que el caso de *seqLen = 10* posea la misma curvatura de gran crecimiento inicial seguido por un estancamiento en su incremento nos indica que la diferencia en los resultados obtenidos está dada por el valor de *seqLen* que supone una barrera en la cobertura que se puede obtener. Esto puede ser así debido a lo mencionado sobre los cambios necesarios del

estado global de contrato para alcanzar ciertas condiciones más profundamente anidadas. Parecería entonces que nuestra intuición se ve reforzada por las curvas observadas en el gráfico.

A continuación analizaremos los resultados obtenidos para los *benchmark SmartPulse* y *Uniswap*. Estos resultados tienen la particularidad de que el 90 % de los casos resultó en empate entre las dos versiones analizadas. En los pocos casos en los que esto no sucedió (de los cuales, de 4 casos, en 3 resultó ganador *EchidnaAFL*), de todas formas la diferencia de *coverage points* observada no fue mayor a 3,2 %. En promedio, observamos un 0,3 % y 0,19 % de diferencia en *coverage points*, para SmartPulse y Uniswap respectivamente. Esto tiene dos explicaciones aparejadas.

Por un lado, la naturaleza de los contratos de estos dos *benchmarks* particulares facilita que no existan muchas divergencias en las líneas de código exploradas. En el caso de *SmartPulse*, la simplicidad de los contratos y sus métodos provocó que en la mayor parte de los casos se alcanzara a cubrir la totalidad de las líneas, no permitiéndose que exista una diferencia en los resultados de las dos versiones. En el único caso en el que se observó una diferencia fue para el contrato *Crowdfunding* con *seqLen = 10*, resultados que se pueden observar en el gráfico 5.6. En este vemos que no existe mucha distancia entre ambas curvas a lo largo de toda la ejecución de la campaña, y que si bien al finalizar esta EchidnaAFL resulta ganador, sólo lo hace con 10 puntos de *coverage*. En cuanto al *benchmark* de *Uniswap*, pudimos observar que los contratos que forman parte del mismo no poseen estado (es decir, no existen variables del contrato). Por lo tanto, la cantidad de elementos de la secuencia no va a lograr una mejor exploración más allá de que se puedan probar las funciones una mayor cantidad de veces. Por otro lado, al tratarse de contratos que utilizan una librería disponible para ser usada en contratos deployados en la *mainnet*, la misma no debería presentar casos de bug, que son justamente lo que buscan encontrar los *asserts* definidos dentro de los contratos. En lo referente a los *coverage points*, esto significa que las líneas no cubiertas suelen estar ligadas a código que sólo puede ser alcanzado de encontrarse algún bug. El resto de las líneas, son cubiertas en su totalidad. La excepción de esto es el contrato *TickOverflowSafety* en donde sí vemos diferencias entre los resultados obtenidos para los tres valores evaluados de longitud de secuencias.

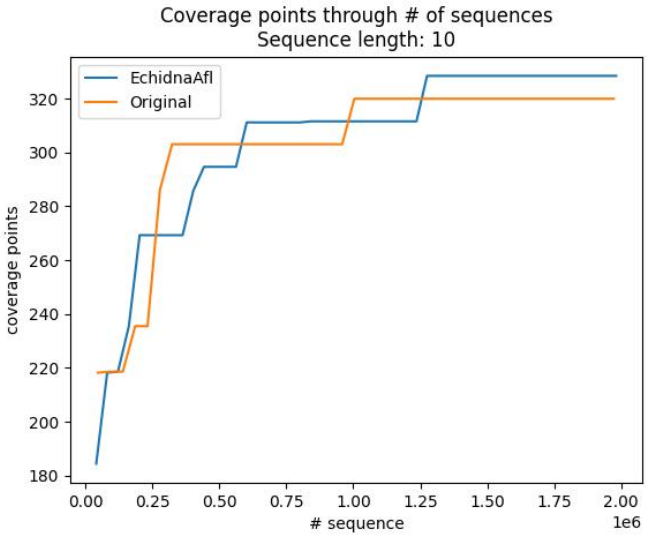

Fig. 5.6: Gráfico de comparación de promedio de coverage points en función de la cantidad de secuencias creadas para el contrato Crowdfunding del benchmark SmartPulse para un valor de seqLen de 10.

Por otro lado, más allá de la naturaleza de los contratos evaluados, el hecho de que en los casos que se presentaron diferencias la misma siga siendo de valores no significativos parece seguir validando que no existe una estrategia de elección de corpus que resulte superior a la otra según los casos de estudio analizados.

Volviendo a nuestra pregunta de investigación inicial y teniendo en cuenta los resultados obtenidos y el análisis realizado para los tres *benchmarks* considerados, podemos afirmar que la versión *EchidnaAFL* no parece suponer una mejora en performance ni exploración de código con respecto a la versión original de la herramienta. La explicación de estos resultados, sin embargo, puede estar dada por dos razones. La primera es que el método que utiliza actualmente Echidna de elección de corpus resulta una buena heurística de *greybox fuzzing*, que resulta igual de competente a la de AFL en este caso de aplicación. La otra posible explicación es que no podemos encontrar una diferencia en la *performance* de ambas versiones porque la elección de corpus para la generación de nuevos *inputs* no resulta determinante a la exploración de la campaña de *fuzzing* de contratos de Ethereum. Esto nos lleva a nuestro segundo eje de análisis que buscará responder si la elección de corpus influye en los resultados obtenidos por la herramienta. En caso de que la respuesta sea afirmativa, podremos además concluir que la estrategia actual de Echidna resulta una buena heurística.

5.3.2. ¿Influye de manera relevante la elección de elementos del corpus en Echidna?

Para poder empezar a responder esta pregunta, utilizaremos la versión *random* mencionada en la sección 4.1.2 como punto de referencia de una elección de corpus que no hace uso de ningún *feedback* y por tanto representa una versión en donde la estrategia de elección de corpus no es relevante al algoritmo. Esto nos permite tener un punto de comparación de manera tal que si *random* obtiene resultados que divergen de los obtenidos para las otras versiones podremos afirmar que la elección de corpus sí está influyendo en

la efectividad del algoritmo de Echidna. En cambio, si se obtienen resultados similares para la versión *random*, esto nos indicará que la elección del corpus no afecta de manera significativa a los resultados de la campaña de *fuzzing* de la herramienta.

Para el *benchmark Maze*, presentamos los resultados en forma gráfica utilizando el mismo formato que en el análisis anterior, para poder nuevamente estudiar posibles tendencias en los mismos. En el gráfico 5.7, podemos ver graficadas los *coverage points* obtenidos por parte de las tres versiones para cada ejecución, agrupadas por igual *seqLen*.

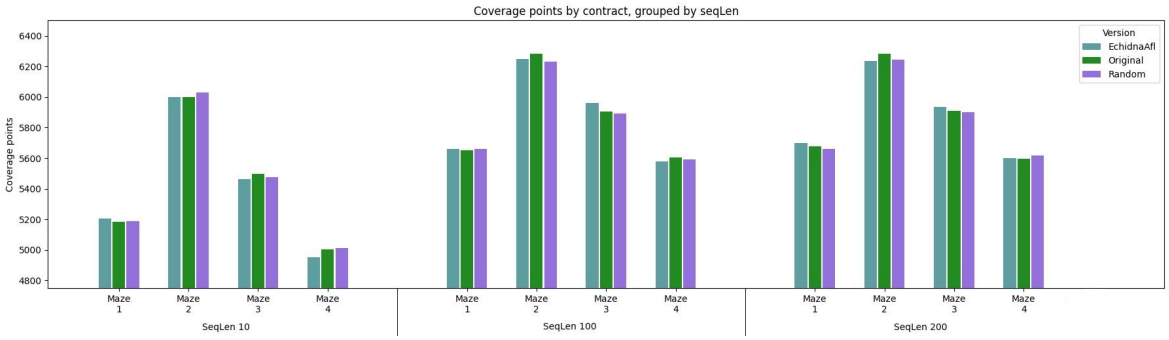

Fig. 5.7: Gráfico de coverage points por contrato del benchmark Maze agrupado por seqLen del promedio de las ejecuciones correspondientes, para las versiones original, random y EchidnaAFL.

Como podemos ver, la versión *random*, representada por el color violeta, no resulta significativamente mejor o peor que las otras dos versiones en ninguno de los 12 escenarios considerados y las diferencias de *coverage points* se siguen manteniendo en el mismo rango. Esto parece indicar que la elección de corpus efectivamente no está influenciando significativamente los resultados obtenidos. Para comparar mejor las versiones, en el gráfico 5.8 podemos ver la distribución de las posiciones obtenidas por cada versión al compararlas entre sí por *coverage points*. En base a este, vale la pena notar que la proporción de posiciones obtenidas por *random* siempre representa un tercio de la totalidad de casos considerados. Si bien hay una pequeña diferencia en la segunda y tercera posición entre original y EchidnaAFL, en líneas generales las tres estrategias obtienen similares resultados, lo cual refuerza nuestra hipótesis de que la elección de corpus utilizando *feedback* no interfiere en la *performance* de la herramienta.

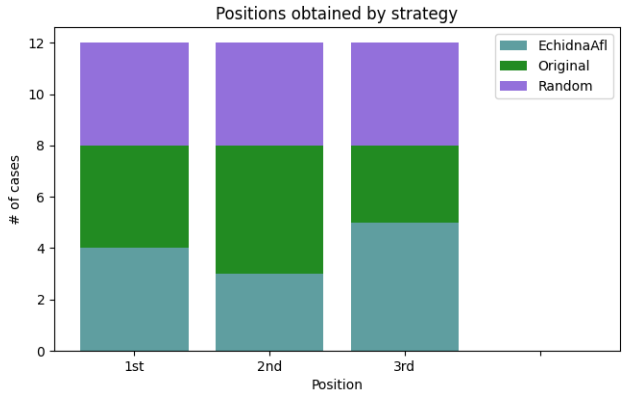

Fig. 5.8: Gráfico de cantidad de casos del benchmark Maze en los que se obtuvo cada posición por promedio de coverage points para las versiones EchidnaAFL, original y random.

Volviendo al gráfico 5.7, podemos ver que se sigue manteniendo que a mayor *seqLen* observamos un mayor valor de *coverage points*. Sin embargo, más allá de las tendencias mencionadas en el análisis anterior que se mantienen, tampoco podemos ver marcada ninguna nueva tendencia por *seqLen* en lo referente a los resultados de *random*.

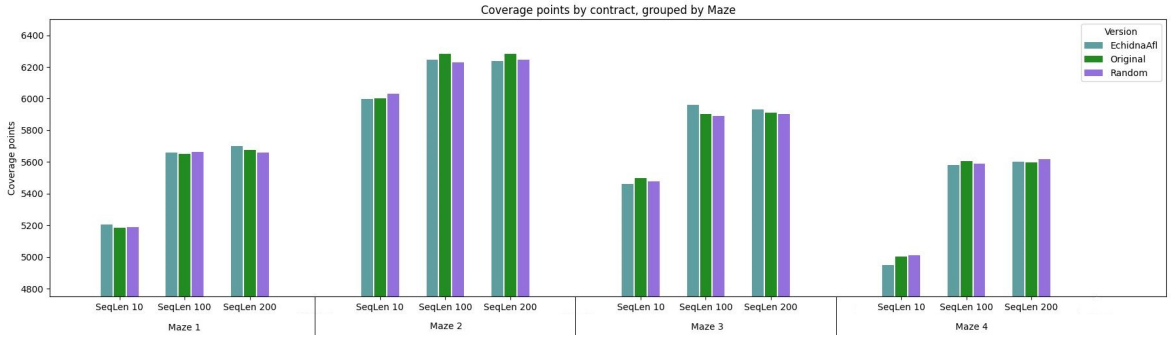

Fig. 5.9: Gráfico de coverage points agrupados por contrato del benchmark Maze para las distintas seqLen de los promedios de las ejecuciones correspondientes, para las versiones original, random y EchidnaAFL.

En cuanto a las tendencias por *maze*, para ser capaces de analizarlo, en el gráfico 5.9 podemos ver la comparación de resultados agrupada por contrato. Podemos notar que tampoco parece existir una tendencia por *maze* para *random*, al contrario de lo que veíamos a primera vista en la comparación entre EchidnaAFL y la original. Esto refuerza además lo concluido en el apartado anterior que no existe una característica determinada de los *mazes* que logre una mejor *performance* para una versión específica.

En lo que respecta a los *benchmarks* SmartPulse y Uniswap, los resultados obtenidos para *random* resultan en comparación similares a los vistos anteriormente. Es decir, en el 90 % de los casos las tres versiones empatan en *coverage points* obtenidos. En los casos restantes, la diferencia de *coverage points* pasa de ser menor a 3,5 % en el análisis anterior a ser 3,75 % en la comparación entre *random* y el mejor de las otras dos versiones, manteniéndose siempre *random* igual o por debajo. Si bien este constituye un cambio en los resultados, la magnitud del mismo no determina que *random* sea una estrategia inferior a

las otras dos versiones.

Para concluir, teniendo los resultados obtenidos y el análisis llevado a cabo, podemos decir que la elección de corpus asignando energías no equitativas no influye significativamente en la *performance* de la herramienta. Volviendo sobre nuestro análisis anterior, esto significa que tampoco podemos afirmar que la estrategia de la versión original constituya una buena heurística ya que se encuentra a la par de *random*.

Otra posible interpretación de los resultados equitativos obtenidos es que la adaptación del algoritmo de *AFL Fast* aplicada a Echidna no resulta apropiada para el contexto dado y por tanto no necesariamente la elección de corpus no influye si no que la calidad de las estrategias consideradas resulta al mismo nivel que la de *random*. Esto quiere decir que existe todavía la posibilidad de que una nueva interpretación consiga otros resultados. En cuanto a esto último, una opción es que la interpretación de camino considerada (explicada en la sección 4.1.1) no sea la adecuada en el contexto de *smart contracts* debido a la existencia de una dependencia extra inherente del estado global para la obtención de una buena exploración de código. Otra opción es que la estrategia de *AFL Fast* en sí misma no sea aplicable en este contexto y que por tanto deban aplicarse otros métodos de *feedback* diseñados más específicamente para tener en cuenta las particularidades de este uso.

Por otro lado, para poder validar las hipótesis analizadas en esta experimentación, se debería llevar a cabo una investigación más exhaustiva considerando otro conjunto de *benchmarks* que resulte más complejo y más diverso.

5.3.3. ¿Cuál es el *overhead* de tiempo asociado a EchidnaAFL?

Si bien hemos realizado los diversos análisis utilizando como punto de referencia la cantidad de secuencias creadas (como mencionamos en la sección 5.2.1), no queremos dejar de cuantificar cómo se ve impactado el tiempo total de la campaña de *fuzzing* mediante las estructuras adicionales necesarias para proveer *feedback* a la estrategia, como es el caso de *EchidnaAFL*.

En los gráficos 5.10, 5.11 y 5.12, podemos ver graficados los porcentajes de *overhead* de tiempo de *EchidnaAFL* por sobre la versión original asociados a los *benchmarks* Maze, SmartPulse y Uniswap, respectivamente.

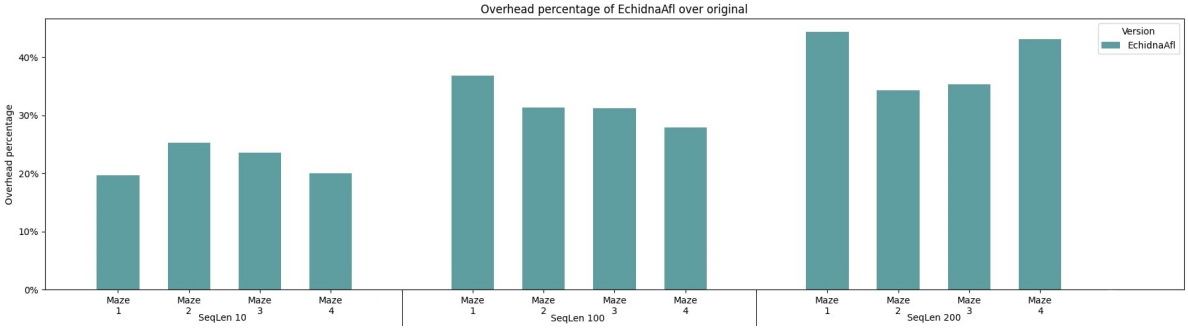

Fig. 5.10: Gráfico de porcentaje promedio de overhead de tiempo observado en EchidnaAFL por sobre la versión original, para las distintas combinaciones de maze y seqLen del benchmark Maze.

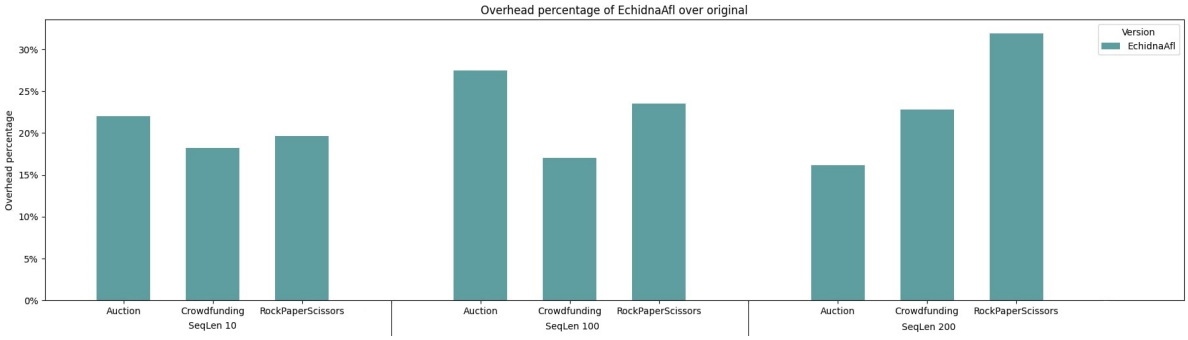

Fig. 5.11: Gráfico de porcentaje promedio de overhead de tiempo observado en EchidnaAFL por sobre la versión original, para las distintas seqLen y contratos del benchmark SmartPulse.

Como podemos ver, en los dos primeros casos observamos un *overhead* que varía entre un 15 % y un 45 %, lo cual constituye magnitudes que indican una degradación en *performance* no despreciable de la herramienta. En el caso del *benchmark* Maze, el promedio ronda el 31 % de *overhead* y para el *benchmark* SmartPulse es de 22 %. Esta diferencia de magnitudes entre los distintos *benchmarks* también es apreciable en los gráficos mencionados. Esto puede estar relacionado con la estructura `currentSequenceCoverage` que se mantiene en el estado global de la campaña y que almacena el conjunto de PCs ejecutado hasta el momento en la secuencia actual. Dado que el universo de posibles PCs en el caso de los *mazes* es mayor que en el caso de SmartPulse (ya que posee más líneas de código), tiene sentido que esta estructura entonces ocupe más espacio y por tanto genere más *overhead*.

Por otro lado, en el gráfico de *overhead* del *benchmark* Uniswap (figura 5.12), podemos ver la presencia de porcentajes mucho más elevados a la vez que se nota una mayor desviación estándar. El promedio en este caso es de 78 %, mucho mayor al de los otros, y existe mucha distancia entre el valor mínimo que es de 1.6 % y el valor máximo, que ronda el valor de 260 %. La razón detrás de esta diferencia con respecto a los otros *benchmarks* no parece estar relacionada con la longitud de los contratos, ya que se encuentran por debajo de las observadas para los *mazes*. Sin embargo, los tiempos observados para la versión original también se encuentran por encima de los tiempos registrados para los otros *benchmarks*, lo cual nos habla de una complejidad temporal mayor que afecta a todas las versiones que puede ser parte de la razón de esta diferencia notable en el *overhead*.

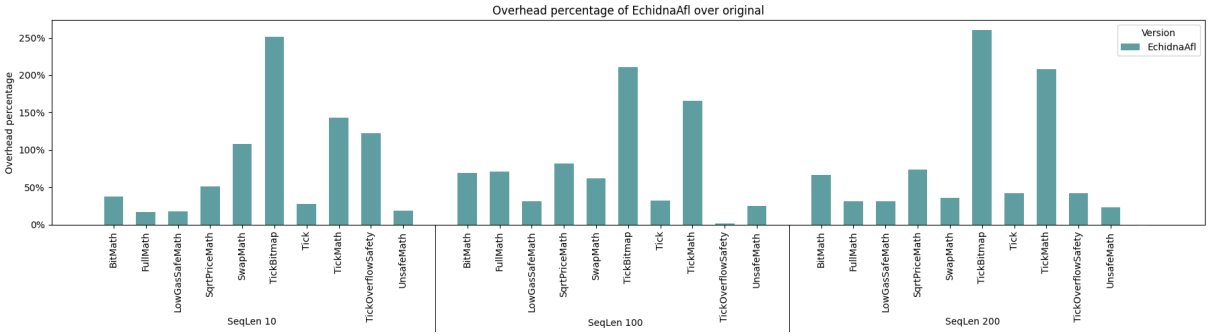

Fig. 5.12: Gráfico de porcentaje promedio de overhead de tiempo observado en EchidnaAFL por sobre la versión original, para las distintas seqLen y contratos del benchmark Uniswap.

En otro orden de análisis, en el gráfico 5.10 vemos marcado un aumento en el porcentaje de *overhead* al aumentar el *seqLen* considerado, mientras que lo mismo no se ve reflejado para los otros dos casos, en las figuras 5.11 y 5.12. Teniendo en cuenta que en el caso de los *mazes*, al aumentar el *seqLen* aumentaban los *coverage points* y por tanto el conjunto de PCs cubiertos, tiene sentido que también veamos un incremento en el *overhead* observado. Esta relación entre *coverage points* y *seqLen* no se mantiene necesariamente para los *benchmarks* de SmartPulse y Uniswap ya que en muchos casos se obtiene una cobertura máxima para todos los *seqLen* considerados, por lo que resulta coherente que tampoco exista dicha relación entre *overhead* y *seqLen*.

Como conclusión, el *overhead* de los cambios introducidos en *EchidnaAFL* resulta significativo y por tanto se podría intentar, de ser necesario, disminuir la diferencia de tiempos de ejecución percibida. Un ejemplo de mejora sería no almacenar frecuencias de secuencias que no se encuentran representadas en el corpus. Como *EchidnaAFL* no resultó constituir una mejora sustancial por sobre la versión original, estas optimizaciones no fueron llevadas a cabo.

6. CONCLUSIONES Y TRABAJO FUTURO

A lo largo de esta tesis, hemos llevado a cabo una implementación de distintas versiones de la herramienta Echidna y hemos realizado un análisis acerca su *performance*, buscando entender cómo afecta la elección del corpus a la efectividad de la misma. Este análisis nos ha llevado a concluir que nuestra versión EchidnaAFL, que utiliza una estrategia de elección de elementos de corpus similar a *AFLFast*, no resulta superior a la versión original ya que obtiene resultados similares tanto en *echidna tests* rotos como en exploración de código obtenida. Además, debido a la necesidad de estructuras adicionales para la implementación de esta versión, la misma trae aparejado un *overhead* de tiempo significativo.

Por otro lado, hemos sido capaces de corroborar que la elección de elementos del corpus con energías no uniformes no pareciera interferir en la efectividad de la herramienta, ya que tanto la versión original como EchidnaAFL resultan igual de performantes que la versión *random* de energías uniformes.

Estos resultados nos generan incógnitas acerca de qué tan aplicable resulta la técnica de *feedback* de AFL Fast al contexto de *fuzzing* de *smart contracts*, ya que pone en juego la presencia de un estado global que permite o no la exploración de escenarios deseados. Por otro lado, también incita dudas acerca de la calidad de adaptación de la representación de caminos elegida, ya que no existía una traducción directa en este contexto de uso.

En base a esto, se abre una arista de investigación posible sobre la representación de caminos usada y la adaptación de la técnica de AFL Fast, ya que existe la posibilidad de que la versión no haya resultado efectiva por una adaptación incorrecta. Como posible trabajo futuro, consideramos que se podría añadir información acerca de la ejecución de cada transacción, con los caminos recorridos por estas y la frecuencia de los mismos. Esto resulta una aplicación más cercana al *fuzzing* tradicional, ya que podemos ver cada transacción como la ejecución de una función en sí misma. Utilizando esta información en conjunto con el *feedback* a nivel de secuencia, creemos que se puede lograr una combinación de ambos aspectos que le otorga importancia tanto al *coverage* por transacción como a los cambios de estado global que se obtienen a partir de la secuencia completa. Esto podría lograrse mediante una combinación lineal de ambos *feedback* que permita calibrar los pesos de cada uno para lograr una *performance* óptima. Otra posible modificación sería cambiar la condición utilizada para añadir elementos al corpus, que en vez de evaluarse si se han logrado nuevas líneas cubiertas, se tenga en cuenta si se ha logrado un nuevo camino de secuencia. Esto permitiría que exista una representación uno a uno entre la estructura `corpusCoverageFrequences` y los elementos existentes en el corpus, lo cual a su vez permitiría mayores elementos de *feedback*.

Más allá de los posibles trabajos futuros, teniendo en cuenta los resultados obtenidos y el análisis realizado sobre los mismos en el transcurso de esta tesis, podemos concluir que la técnica que pareciera ser mejor basada en su efectividad y por sobre todo su tiempo de ejecución es la versión original de Echidna. Vale la pena mencionar que la versión *random* resulta igual de performante y que si bien en varios casos tiene un mejor tiempo de ejecución, esto no es consistente y por lo tanto no tiene sentido reemplazar la estrategia original.

Si bien estas conclusiones tienen sentido en el marco de esta tesis y los *benchmarks* utilizados, creemos que debería llevarse a cabo un análisis más profundo utilizando *bench-*

marks más diversos y extensos para lograr tener confianza en las tendencias observadas. Por ejemplo, en el *benchmark* Maze, habíamos observado una mejora en la *performance* de *EchidnaAFL* a medida que crecía el *seqLen* considerado. En caso de que esta tendencia se siga corroborando con otros *benchmarks*, tendría sentido realizar un estudio de la relación de *seqLen* con la efectividad de EchidnaAFL, para poder así determinar si esta mejora se sigue estancando con valores mucho mayores de *seqLen* a la vez que determinar el valor de *seqLen* óptimo para esta estrategia. Por otro lado, teniendo en cuenta que los benchmarks utilizados son ya sea casos de uso artificiales, muy sencillos o *stateless*, tiene sentido realizar un análisis más profundo a partir de la creación o utilización de un benchmark complejo y de uso real al mismo tiempo.

Como conclusión del trabajo realizado, creemos que existen múltiples oportunidades de continuar con la investigación sobre Echidna y cómo lograr mejorar su *performance* en base a su técnica de *feedback*. Para facilitar esta tarea, creemos que la disponibilización de la documentación que realizamos del algoritmo de Echidna puede resultar de utilidad para futuros investigadores y desarrolladores mismos de la herramienta.

Contrato	*seqLen*	*Coverage points: Original*	*Coverage points: EchidnaAFL*	*Coverage poi*
BitMath	10	530	530	53
FullMath	10	695	695	69
LowGasSafeMath	10	603	603	60
SqrtPriceMath	10	2978	2978	297
SwapMath	10	1520	1520	152
Tick	10	285	285	28
TickBitmap	10	1106	1106	110
TickMath	10	1307	1307	130
TickOverflowSafety	10	1688	1733	166
UnsafeMath	10	151	151	15
BitMath	100	530	530	53
FullMath	100	695	695	69
LowGasSafeMath	100	603	603	60
SqrtPriceMath	100	2978	2978	297
SwapMath	100	1520	1520	152
Tick	100	285	285	28
TickBitmap	100	1106	1106	110
TickMath	100	1307	1307	130
TickOverflowSafety	100	1728	1756	171
UnsafeMath	100	151	151	15
BitMath	200	530	530	53
FullMath	200	695	695	69
LowGasSafeMath	200	603	603	60
SqrtPriceMath	200	2978	2978	297
SwapMath	200	1520	1520	152
Tick	200	285	285	28
TickBitmap	200	1106	1106	110
TickMath	200	1307	1307	130
TickOverflowSafety	200	1734	1711	171
UnsafeMath	200	151	151	15

Tab. 6.1: *Tabla comparativa de los resultados obtenidos de coverage points promedio para las versiones original, EchidnaAFL y random para los contratos del benchmark Uniswap, para todos los valores de seqLen utilizados.*

Bibliografía

[1] Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform (2013). https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf

[2] Ethereum Virtual Machine https://ethereum.org/es/developers/docs/evm/

[3] Etherscan, visitado en Febrero de 2023 https://etherscan.io/chart/address

[4] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur and H. -N. Lee, Ëthereum Smart Contract Analysis Tools: A Systematic Review,ïn IEEE Access, vol. 10, pp. 57037-57062, 2022, doi: 10.1109/ACCESS.2022.3169902 https://ieeexplore.ieee.org/document/9762279

[5] Coinmarketcap, visitado en Febrero de 2023 https://coinmarketcap.com/

[6] Mehar, Izhar & Shier, Charles & Giambattista, Alana & Gong, Elgar & Fletcher, Gabrielle & Sanayhie, Ryan & Kim, Henry & Laskowski, Marek. (2019). Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. Journal of Cases on Information Technology. 21. 19-32. 10.4018/JCIT.2019010102

[7] David Ingram & Jason Abbruzzese (March 29, 2022), Hackers steal more than $600 million from maker of Axie Infinity https://www.nbcnews.com/tech/tech-news/hackers-steal-600-million-maker-axie-infinity-rcna22031

[8] Solidity documentation. https://docs.soliditylang.org/en/v0.8.19/ (2023)

[9] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, ‘‘Echidna: Effective, usable, and fast fuzzing for smart contracts,’’ in Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA), Jul. 2020, pp. 557{560, doi: 10.1145/3395363.3404366. https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf

[10] H. Liang, X. Pei, X. Jia, W. Shen and J. Zhang, "Fuzzing: State of the Art,ïn IEEE Transactions on Reliability, vol. 67, no. 3, pp. 1199-1218, Sept. 2018, doi: 10.1109/TR.2018.2834476. https://wcventure.github.io/FuzzingPaper/Paper/TRel18_Fuzzing.pdf

[11] Boosted greybox fuzzing
https://www.fuzzingbook.org/html/GreyboxFuzzer.html

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 1032{1043.
https://mboehme.github.io/paper/CCS16.pdf

[13] Bitcoin whitepaper
https://bitcoin.org/bitcoin.pdf

[14] Ethereum documentation
https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

[15] https://www.pixelcrayons.com/blog/pros-and-cons-of-blockchain-technology/

[16] Wang, Q., Li, R., Wang, Q., Chen, S. (2021). Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. arXiv preprint arXiv:2105.07447.

[17] https://ethereum.org/en/developers/docs/smart-contracts/upgrading/

[18] Control flow graph definition https://ics.uci.edu/ lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf

[19] Kushwaha, S. S., Joshi, S., Singh, D., Kaur, M., Lee, H. N. (2022). Ethereum smart contract analysis tools: A systematic review. IEEE Access, 10, 57037-57062.

[20] Fu, Ying Ren, Meng Ma, Fuchen Jiang, Yu Shi, Heyuan Sun, Jiaguang. (2019). EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine.

[21] B. Jiang, Y. Liu and W. K. Chan, ÇontractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection,"2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), Montpellier, France, 2018, pp. 259-269, doi: 10.1145/3238147.3238177.

[22] Nassirzadeh, Behkish Sun, Huaiying Banescu, Sebastian Ganesh, Vijay. (2021). Gas Gauge: A Security Analysis Tool for Smart Contract Out-of-Gas Vulnerabilities.

[23] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: an efficient adaptive fuzzer for Solidity smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20). Association for Computing Machinery, New York, NY, USA, 778{788.
https://doi.org/10.1145/3377811.3380334

[24] Yuhe Huang, Bo Jiang, and W. K. Chan. 2021. EOSFuzzer: Fuzzing EOSIO Smart Contracts for Vulnerability Detection. In Proceedings of the 12th Asia-Pacific Symposium on Internetware (Internetware '20). Association for Computing Machinery, New York, NY, USA, 99{109. https://doi.org/10.1145/3457913.3457920

[25] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce and S. K. Cha, "SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses,"2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), Melbourne, Australia, 2021, pp. 227-239, doi: 10.1109/ASE51524.2021.9678888.

[26] Foundry documentation y Github https://book.getfoundry.sh/ https://github.com/foundry-rs/book

[27] Josselin Feist, Gustavo Greico, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19). IEEE Press, 8{15. https://doi.org/10.1109/WETSEB.2019.00008

[28] Github del proyecto Medusa https://github.com/crytic/medusa/

[29] Github de cambios realizados a Echidna version EchidnaAFL https://github.com/ddrenteria/echidna/tree/feat/add-feature-updated

[30] Github de cambios realizados a Echidna version Random https://github.com/ddrenteria/echidna/tree/random_baseline

[31] Echidna config https://github.com/crytic/echidna/wiki/Config

[32] Github del benchmark Maze https://github.com/Consensys/daedaluzz

[33] Github del benchmark SmartPulse https://github.com/utopia-group/SmartPulseTool/tree/master/benchmarks/evalBenchmarks

[34] Github del benchmark Uniswap https://github.com/Uniswap/v3-core/tree/main/contracts/test