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

Generador de Abstracciones de Comportamiento para Contratos Inteligentes mediante Fuzzing

Tesis de Licenciatura en Ciencias de la Computación

Ian Grinspan

Director: Javier Godoy

Codirector: Diego Garbervetsky

Buenos Aires, 2025

GENERADOR DE ABSTRACCIONES DE COMPORTAMIENTO PARA CONTRATOS INTELIGENTES MEDIANTE FUZZING

Los contratos inteligentes son programas inmutables desplegados en la blockchain, utilizados para gestionar activos digitales y automatizar acuerdos sin intermediarios. Dada la inmutabilidad de su código una vez desplegado y el manejo de recursos de alto valor que emplean, la identificación temprana de errores y vulnerabilidades es crucial para evitar pérdidas económicas y fallos de seguridad.

En este trabajo se presenta un enfoque para la generación automática de abstracciones por predicados, aplicada a contratos inteligentes escritos en Solidity para la red Ethereum. A diferencia de herramientas previas que emplean análisis estático, esta propuesta se basa en técnicas de análisis dinámico mediante *fuzzing*, utilizando la herramienta de código abierto Echidna.

El prototipo desarrollado permite explorar el comportamiento de los contratos generando abstracciones que reflejan el estado del contrato y las precondiciones necesarias para la habilitación de sus funciones. Este enfoque permite identificar estados de ejecución no triviales y condiciones que podrían pasar desapercibidas en un análisis manual.

Se analizan las ventajas y limitaciones de la herramienta propuesta, comparándola con enfoques previos que emplean analizadores estáticos, y se discuten posibles extensiones para mejorar la eficiencia y precisión del análisis.

Palabras claves: Validación de software, Análisis dinámico, Fuzzing, Echidna, Blockchain, Contratos inteligentes, Ethereum, Solidity, Abstracciones por predicados.

BEHAVIOR ABSTRACTION GENERATOR FOR SMART CONTRACTS USING A FUZZER

Smart contracts are immutable programs deployed on the blockchain, used to manage digital assets and automate agreements without intermediaries. Given the immutability of their code once deployed and the high-value resources they handle, early identification of errors and vulnerabilities is crucial to prevent economic losses and security breaches.

This work presents an approach for the automatic generation of predicate-based abstractions applied to smart contracts written in Solidity for the Ethereum network. Unlike previous tools that rely on static analysis, this proposal is based on dynamic analysis techniques through fuzzing, using the open-source tool Echidna.

The developed prototype allows for exploring contract behavior by generating abstractions that reflect the contract's state and the preconditions required to enable its functions. This approach helps identify non-trivial execution states and conditions that might be overlooked in manual analysis.

The advantages and limitations of the proposed tool are analyzed, comparing it with previous approaches that use static analyzers, and potential extensions are discussed to improve the efficiency and accuracy of the analysis.

Keywords: Software validation, Dynamic analysis, Fuzzing, Echidna, Blockchain, Smart contracts, Ethereum, Solidity, Predicate-based abstractions.

AGRADECIMIENTOS

Quiero agradecer, en primer lugar, a los jurados Juan Pablo Galeotti y Gustavo Grieco por su tiempo.

A Javi y a Diego, por la oportunidad, la predisposición, la buena onda y por siempre hacerme sentir valorado.

A Abi y a Andy, por dejarme transitar toda la carrera a su lado.

A la UBA, por ampliar mi visión del mundo y por el orgullo de ser parte de su historia y a todos los trabajadores de la universidad pública, por su esfuerzo y dedicación.

A mi vieja, Geral, por ser lo más grande que hay. A Juan y a Ricky, por cuidarme desde cerca y desde lejos.

A Anto y a la LBB, que son la familia que elegí.

Índice general

1.. Introducción 1

2.. Preliminares 3
- 2.1. Blockchain 3
  - 2.1.1. Ejemplo de contrato inteligente en Solidity 4
- 2.2. EPAs 6
- 2.3. Abstracciones por predicados para contratos inteligentes 9
- 2.4. Echidna 11

3.. Diseño de la herramienta 15
- 3.1. Modo EPA 15
- 3.2. Modo States 17
- 3.3. Implementación 20
  - 3.3.1. Archivo de configuración 20
  - 3.3.2. Ejecución 22
- 3.4. Integración con VeriSol 23
- 3.5. Optimizaciones 24
  - 3.5.1. Manejo de funciones con precondición `true` 24
  - 3.5.2. Combinación de funciones con las mismas precondiciones 24
  - 3.5.3. Descarte de estados inalcanzables 25
- 3.6. Consideraciones 27
  - 3.6.1. Diferencia de versiones 27
  - 3.6.2. Separar en múltiples contratos 27
  - 3.6.3. Asserts internos 28
  - 3.6.4. Fuzzing en el constructor 28

4.. Evaluación 31
- 4.1. Introducción 31
- 4.2. Metodología de evaluación 32
- 4.3. Resultados 33
  - 4.3.1. Viabilidad de la generación de abstracciones mediante fuzzing 33
  - 4.3.2. Utilidad y aplicación de las abstracciones generadas 42
  - 4.3.3. Comparación de cobertura con VeriSol 45
  - 4.3.4. Comparación de tiempos con VeriSol 51
- 4.4. Discusión 53

5.. Trabajo relacionado 55

6.. Trabajo futuro 57

7.. Conclusiones 61

1. INTRODUCCIÓN

El presente trabajo se enfoca en el análisis del comportamiento de contratos inteligentes escritos en Solidity para la red de Ethereum, mediante la construcción de abstracciones que modelan su funcionamiento. Con el crecimiento exponencial del uso de la tecnología blockchain, los contratos inteligentes han adquirido un rol fundamental en el uso de activos digitales y la creación de aplicaciones descentralizadas, entre otras cosas. Sin embargo, la complejidad de estos programas, sumada a su inmutabilidad y ejecución en entornos descentralizados, plantea desafíos significativos en términos de seguridad. Detectar errores o vulnerabilidades en los contratos antes de su despliegue es crítico para prevenir pérdidas económicas y mejorar la confianza en los sistemas basados en blockchain.

Este contexto ha dado lugar al desarrollo de herramientas que buscan analizar contratos inteligentes mediante distintas técnicas, principalmente de análisis estático mediante verificación formal y *model checking*. Estas metodologías intentan garantizar que los contratos cumplan con propiedades específicas mediante un análisis exhaustivo de todas sus posibles ejecuciones. Sin embargo, generalmente requieren de intervención humana en alguna de sus etapas. Es común complementar el proceso con auditorías de seguridad manuales, en las que expertos revisan el código en busca de vulnerabilidades que podrían pasar desapercibidas en un análisis automatizado. A la vez, existen herramientas que buscan crear modelos abstractos que capturen el comportamiento de los contratos, facilitando su interpretación y evaluación tanto para desarrolladores como para auditores. No obstante, cada enfoque presenta limitaciones que motivan la búsqueda de otro tipo de técnicas.

En este trabajo se presentan dos contribuciones principales. La primera es el desarrollo de una herramienta que, a partir del código fuente de contratos inteligentes escritos en Solidity, genera automáticamente abstracciones por predicado utilizando un análisis dinámico mediante campañas de *fuzzing*. Esta técnica permite explorar estados y transiciones alcanzables de un contrato, proporcionando una visión general de su comportamiento y facilitando la detección de errores o inconsistencias. La segunda contribución es la integración parcial con una herramienta previamente desarrollada que utiliza análisis estático, lo que abre la posibilidad de combinar ambos enfoques en futuros desarrollos para lograr un análisis más exhaustivo.

Para validar la efectividad de la herramienta, se realizarán experimentos sobre un conjunto de *benchmarks*, evaluando tanto la cobertura alcanzada como la calidad de las abstracciones generadas. Para ello, se compararán las abstracciones obtenidas con abstracciones ya existentes para dichos *benchmarks* y se analizará el impacto de diversas configuraciones y optimizaciones en la ejecución. En particular, se busca responder preguntas clave, como la viabilidad de generar abstracciones dinámicas a partir de contratos inteligentes y la utilidad práctica de dichas abstracciones para detectar errores.

En síntesis, este trabajo busca aportar una nueva perspectiva al análisis de contratos inteligentes, combinando técnicas de *fuzzing* y abstracción de estados con herramientas existentes y proponiendo un enfoque que puede sentar las bases para investigaciones futuras.

2. PRELIMINARES

2.1. Blockchain

A grandes rasgos, una blockchain [37] es una estructura formada por una secuencia de bloques que funciona como un libro contable accesible a todos los usuarios de una red. Estos usuarios pueden realizar transacciones entre sí y la información de estas transacciones es almacenada en los bloques de la blockchain y se encuentra disponible públicamente.

Una característica fundamental de la blockchain es la descentralización, en el sentido en que no hay una autoridad central, sino que la información se distribuye entre todos los nodos de la red y las acciones se verifican gracias a mecanismos de consenso. Otra característica importante es la inmutabilidad: una vez que se realiza una transacción, no se puede deshacer.

En una blockchain, la cuenta de cada usuario está asociada a una cantidad específica de criptomoneda (como Bitcoin o Ether). En la mayoría de las blockchains, esta cantidad se calcula a partir del historial completo de transacciones registradas en la cadena de bloques. Este cálculo, que puede realizar cualquier usuario debido a la naturaleza pública de la blockchain, se lleva a cabo recorriendo las transacciones desde el bloque más reciente hasta el inicial.

Este recorrido es posible porque cada bloque contiene un puntero que referencia al bloque anterior, permitiendo visitar todos los bloques de forma secuencial hasta llegar al bloque inicial, llamado *bloque génesis*. La figura 2.1 ilustra esta estructura y su funcionamiento.

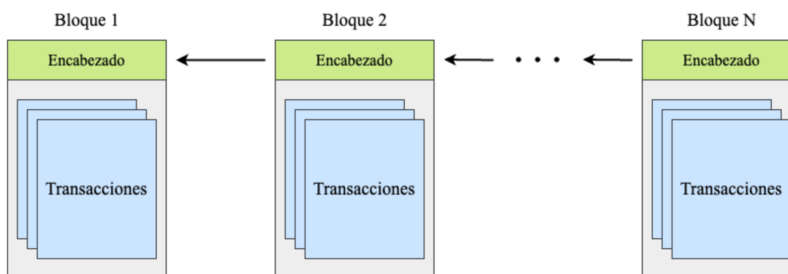

Fig. 2.1: Diagrama de una blockchain

Ethereum [19] va más allá de la infraestructura estándar de las blockchains tradicionales, implementando un modelo basado en cuentas que se diferencia del clásico modelo basado en transacciones de Bitcoin[33]. Esto hace que exista la noción intuitiva de un estado global de la blockchain, que se puede pensar como una estructura que incluye el saldo y los datos de cada cuenta de la red. Los participantes pueden modificar este estado interactuando con contratos inteligentes.

En Ethereum existen dos tipos de cuentas, las cuentas externas y las cuentas de contrato. Las cuentas externas son las cuentas que pertenecen a los usuarios de la red y pueden enviar y recibir fondos. Son las encargadas de iniciar las transacciones y cada propietario tiene acceso a la clave privada de la cuenta, la cual utiliza para firmar transacciones. Por otro lado, las cuentas de contrato son cuentas que están asociadas con contratos inteligen-

tes y están controladas por el código de los mismos.

Un contrato inteligente es un programa que se almacena en una dirección de la blockchain de la misma manera que una cuenta externa. Por lo tanto, los contratos inteligentes son cuentas que pueden recibir y transferir fondos y cada uno tiene su propio balance. Al igual que las transacciones entre cuentas en la blockchain no se pueden revertir, un programa implementado en un contrato inteligente de la red Ethereum no se puede modificar.

Una vez que se crea un contrato inteligente, el mismo es desplegado por un usuario a la red, que cuenta con una máquina virtual proporcionada por Ethereum (la *Ethereum Virtual Machine* ó EVM)[5] que se encarga de efectivamente ejecutar las transacciones entre cuentas y actualizar el estado interno de la red.

Estos contratos suelen ser escritos en el lenguaje de programación Solidity [14] [15], desarrollado específicamente para Ethereum, que es compilado por el compilador solc[13] y traducido a bytecode listo para ejecutar por la EVM.

Las funciones de los contratos inteligentes se invocan desde cuentas (direcciones en la blockchain) que disponen de fondos. Para ejecutarse, el contrato inteligente consume una cantidad de gas (es decir, una cantidad de la criptomoneda) que quien invoca la función debe proveer. El costo en gas de ejecutar una transacción dependerá de las acciones específicas que realizadas por el contrato inteligente y la carga de la red en ese momento.

2.1.1. Ejemplo de contrato inteligente en Solidity

En esta sección se analizará en detalle un ejemplo de contrato inteligente escrito en lenguaje Solidity.

En el listing 2.1 se puede ver el contrato *Auction*. El mismo representa un sistema que permite a los usuarios participar en una subasta digital de manera transparente y segura. El contrato gestiona todo el proceso de una subasta, desde su inicio hasta su conclusión, incluyendo el manejo de pujas y la distribución final de fondos.

```
contract Auction {
    uint public auctionStart;
    uint public biddingTime;
    address payable public beneficiary;

    bool public ended = false;
    address payable public highestBidder = payable(address(0x0));
    uint public highestBid = 0;
    mapping(address => uint) public pendingReturns;
    uint public pendingReturnsCount = 0;

    constructor(uint _auctionStart, uint _biddingTime, address payable
        _beneficiary) {
        auctionStart = _auctionStart;
        biddingTime = _biddingTime;
        beneficiary = _beneficiary;
    }

    function bid() public payable {
        require(block.number >= auctionStart, "Auction has not started");
        uint end = auctionStart + biddingTime;
        require(block.number < end && !ended, "Auction has ended");
        require(msg.value > highestBid, "Bid not high enough");
```

```
        if (highestBidder != address(0x0) && pendingReturns[highestBidder] == 0) {
            pendingReturnsCount += 1;
        }
        pendingReturns[highestBidder] += highestBid;
        highestBidder = payable(msg.sender);
        highestBid = msg.value;
    }

    function withdraw() public {
        require(pendingReturns[msg.sender] != 0 && pendingReturnsCount > 0, "No
            funds to withdraw");

        uint amount = pendingReturns[msg.sender];
        pendingReturns[msg.sender] = 0;
        pendingReturnsCount -= 1;

        payable(msg.sender).transfer(amount);
    }

    function auctionEnd() public {
        uint end = auctionStart + biddingTime;
        require(block.number > end && !ended, "Auction cannot be ended yet");

        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

Listing 2.1: Contrato Auction

Al desplegar el contrato, se establece el marco temporal de la subasta. Esto se logra mediante las variables `auctionStart`, que marca el inicio de la subasta, y `biddingTime`, que determina su duración medida en cantidad de bloques. Además, se designa un beneficiario, cuya dirección se almacena en la variable `beneficiary`, y que recibirá el pago de la puja ganadora al concluir la subasta. Toda esta configuración está incluida en el constructor del contrato y se recibe en forma de argumentos. El constructor es público, es decir que cualquier usuario puede desplegar su propia instancia del contrato Auction con sus parámetros determinados, siempre y cuando tenga acceso al código.

Durante el transcurso de la subasta, el contrato mantiene un registro del estado actual. La variable `ended` indica si la subasta ha concluido, mientras que `highestBidder` y `highestBid` almacenan la dirección y el valor de la oferta más alta en cada momento. Para manejar las devoluciones a los participantes no ganadores, se utiliza un *mapping* llamado `pendingReturns`, junto con un contador `pendingReturnsCount`.

Además del constructor, el contrato cuenta con 3 funciones públicas que pueden ser llamadas por cualquier usuario y determinan el comportamiento del programa. La función `bid()` es una función de tipo `payable` que permite a los usuarios realizar ofertas. Cada vez que se llama a esta función, el contrato verifica que la subasta aún esté en curso y que la nueva oferta supere a la actual más alta. Si estas condiciones se cumplen, la mayor oferta anterior se añade a `pendingReturns` para su posible devolución, y se actualizan `highestBidder` y `highestBid` con los nuevos valores. La keyword `payable` es crucial, ya

que permite a la función recibir Ether junto con la llamada a la función, y la cantidad de Ether enviada está especificada en `msg.value`.

Para los postores que han sido superados, el contrato ofrece la función `withdraw()`. Esta función permite a los usuarios retirar sus ofertas anteriores que han sido superadas. El contrato verifica que el usuario tenga fondos para retirar y, en caso afirmativo, transfiere los fondos y actualiza el estado del contrato para reflejar el retiro. La transferencia de fondos se realiza mediante la función `transfer`, que es la manera más sencilla de enviar Ether en Solidity y tiene su propio mecanismo de seguridad ante fallas en la transferencia.

La conclusión de la subasta se maneja mediante la función `auctionEnd()`. Esta función solo puede ejecutarse una vez que el tiempo de la subasta ha expirado, lo cual se verifica comparando `block.number` con la suma de `auctionStart` y `biddingTime`. Cuando se llama con éxito, marca la subasta como terminada y transfiere la oferta más alta al beneficiario designado al inicio.

A lo largo de su ciclo de vida, el contrato puede encontrarse en diversos estados. Inicialmente, dependiendo del valor con el cual es inicializada la variable `auctionStart`, la subasta puede estar o bien en un estado de espera o bien en curso. Luego, cuando la subasta está en curso, puede aceptar ofertas de los participantes y retiros de dinero de postores cuya puja ha sido superada. Una vez que el tiempo de la subasta expira, entra en un estado en el que ya no se aceptan más ofertas, pero aún pueden realizarse retiros de fondos. En este caso, se puede pensar a la subasta como finalizada.

2.2. EPAs

El interés central de este trabajo es utilizar abstracciones acerca del comportamiento de programas para validar que cumplan con su comportamiento esperado. Para eso se suelen crear modelos que luego serán usados para guiar a los desarrolladores, usuarios o auditores a entender el programa y a detectar posibles vulnerabilidades.

Una manera de obtener modelos abstractos de comportamiento es utilizando abstracciones por predicado. Este enfoque consiste en definir un conjunto de predicados y agrupar los estados concretos del programa de acuerdo con la validez de estos predicados. Es decir, cada estado abstracto representa un conjunto de estados concretos que da la misma valuación a todos los predicados.

En este contexto, se introducen las *enabledness-preserving abstractions* [20], que son máquinas de estados que describen el comportamiento de la implementación de un programa introduciendo abstracción a la misma de varias maneras distintas:

- Por un lado, se ignoran los estados alcanzados mientras se ejecuta un método; el foco se pone en los estados del programa antes y después de ejecutar un método.
- Por otro lado, los valores de la estructura de datos subyacente son abstraídos; nos centramos en qué operaciones están permitidas (y cuáles no) cuando tenemos esos valores. De esta manera, los estados de una EPA se agrupan según qué conjunto de operaciones están habilitadas en cada estado.
- Por último, también se ignoran los valores de retorno y los argumentos de las funciones. Lo único que va a importar es si existen valores de los parámetros tales que la ejecución del método hace que la implementación pase de un estado abstracto a otro.

Formalmente, una clase C es una estructura $\langle M, F, R, inv, init \rangle$ donde $M = \{m_1, ..., m_k\}$ es un conjunto finito de métodos públicos, F es un conjunto de implementaciones de esos métodos, R es un conjunto de precondiciones, inv es el invariante de la clase e $init$ denota las condiciones iniciales, dadas por el constructor. Los conjuntos F y R están indexados por M, es decir, para cada método m_i hay una implementación f_i y una precondición r_i.

Dada una clase C y dos instancias de la misma, c_1 y c_2, se dice que c_1 y c_2 son equivalentes (*enabledness equivalent*, $\equiv_e$) si para todo método $m \in M$, c_1 satisface R_m si y sólo si c_2 satisface R_m.

Cuando separamos a las instancias usando $\equiv_e$ obtenemos un conjunto de estados abstractos, en el que cada uno está mapeado a un conjunto (único) de métodos permitidos. Cada estado abstracto agrupa todas las instancias que comparten el mismo conjunto de métodos permitidos, y puede ser caracterizado por un predicado de estados.

A partir de una clase, se puede definir una EPA como $\langle \Sigma, S, S_0, \delta \rangle$, donde Σ es un conjunto de etiquetas de transición, S es un conjunto de estados abstractos, $S_0 \subseteq S$ es el conjunto de estados iniciales y δ es una función de transición de estados, que establece cómo se conectan los estados por medio de etiquetas. Estos elementos deben cumplir las siguientes condiciones:

- El conjunto Σ es igual al conjunto de métodos públicos de la clase C (M).

- El conjunto S es el conjunto de todos los posibles subconjuntos de M. Cada elemento de S es un conjunto de métodos permitidos, dados por alguna instancia concreta c para la cual además se cumple $inv(c)$. Esto quiere decir que en S están los estados abstractos ya mencionados.

- Los elementos de $S_0 \subseteq S$ son conjuntos de métodos permitidos por alguna instancia c de la clase en la que se satisface tanto el invariante de estado como la condición inicial.

- La función de transición entre estados abstractos δ se define de la siguiente manera: para cada elemento A de S (cada conjunto de métodos permitidos / estado abstracto) y para cada método público a de Σ:

  - Si $a \notin A$ (a no está permitido en el estado abstracto A), entonces

  $$\delta(A, a) = \emptyset. \tag{2.1}$$

  - En cambio, si $a \in A$, definimos a la función de transición como

  $$\delta(A, a) = \{\, B \mid \exists c.\ inv(c) \land pred_A(c) \land \exists p.\ R_a(c, p) \land pred_B(F_a(c, p)) \,\}, \tag{2.2}$$

  donde c es una instancia de la clase, $pred_A$ y $pred_B$ son los predicados que caracterizan a los estados abstractos A y B respectivamente y p es un conjunto de parámetros para el método a. Esta ecuación nos dice que δ va desde A hacia B por medio de la transición a si existe una instancia c del contrato (que cumple con el predicado de A) tal que luego de ejecutar la función a con parámetros p, resulta en una instancia que cumple con el predicado de B.

Vale la pena señalar que las EPAs pueden resultar en una sobreaproximación del comportamiento del programa, ya que en el resultado se pueden incluir secuencias que no

son cadenas de invocación permitidas por el programa. Esto se debe a que las EPAs no tienen en cuenta el orden en el que se invocan los métodos, es decir que no representan necesariamente un flujo posible del programa.

Un ejemplo de este comportamiento es el siguiente. Supongamos que tenemos un programa en el que contamos con los estados abstractos A, B, C y D y las funciones f1(), f2() y f3(). En este programa, un posible flujo de ejecución es partir del estado A y pasar por el estado B para luego llegar al C y terminar en el D, luego de ejecutar las funciones f1(), f2() y, en dos ocasiones consecutivas, f3(). Además, otro posible camino que puede tomar el programa es partir desde A y llegar hasta C tras ejecutar la función f1(), finalizando allí su ejecución. La cuestión surge de que, analizando la EPA, el lector podría pensar que existe un flujo de ejecución desde el estado A hacia D pasando únicamente por el estado C, tras ejecutar las funciones f1() y f3(), cuando esto no puede suceder en el contrato.

Esto se puede ver en la figura 2.2.

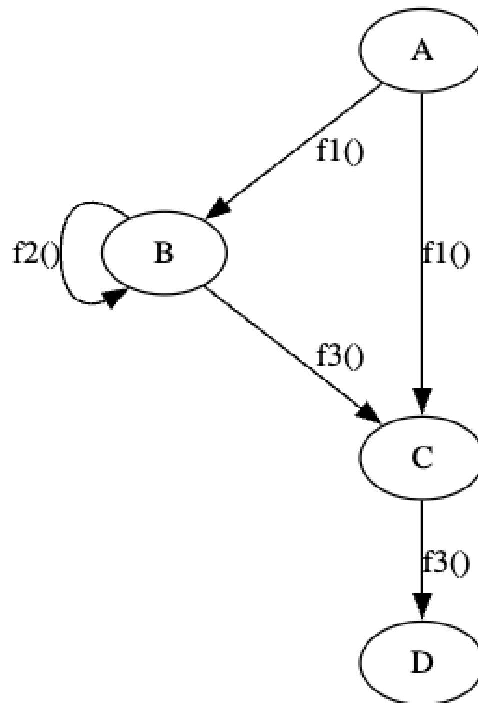

Fig. 2.2: Ejemplo de sobreaproximación en una EPA

El ejemplo presentado en la figura 2.2 es un caso real, extraído del contrato *Crowdfunding*, uno de los evaluados en este trabajo. Este caso puede ser observado en la figura 2.3. Es importante aclarar que, exceptuando el nodo etiquetado con la palabra `init` (agregado por conveniencia para representar el estado previo a la ejecución del constructor), cada nodo está etiquetado únicamente con las funciones que están habilitadas en ese estado abstracto. En la figura, se observa que, desde el estado inicial, al ejecutar el constructor, se puede alcanzar el estado en el que sólo se puede ejecutar la función `t()`, y desde allí, al invocar esa función, se llegaría al estado en el que también está permitida la función `Claim()`. Sin embargo, este flujo de ejecución nunca puede ocurrir en el contrato real, ya que para poder ejecutar `Claim()`, previamente alguien debe haber realizado una donación (es decir, haber llamado a la función `Donate()`), dado que esta es una de las precondiciones

de la función `Claim()`.

El flujo de ejecución que sí es posible, y que puede deducirse correctamente de la abstracción, es aquel que inicia en el estado inicial, pasa por el estado en el que está permitida la función `Donate()`, luego por el estado que sólo permite `t()` y finalmente llega al estado que permite `Claim()`, ejecutando en el camino el constructor y las funciones `Donate()`, `t()` y `t()`, respectivamente.

En conclusión, las EPAs no deben interpretarse como representaciones de secuencias específicas de invocación de funciones, sino como modelos de la capacidad del contrato para alcanzar distintos estados y transicionar entre ellos.

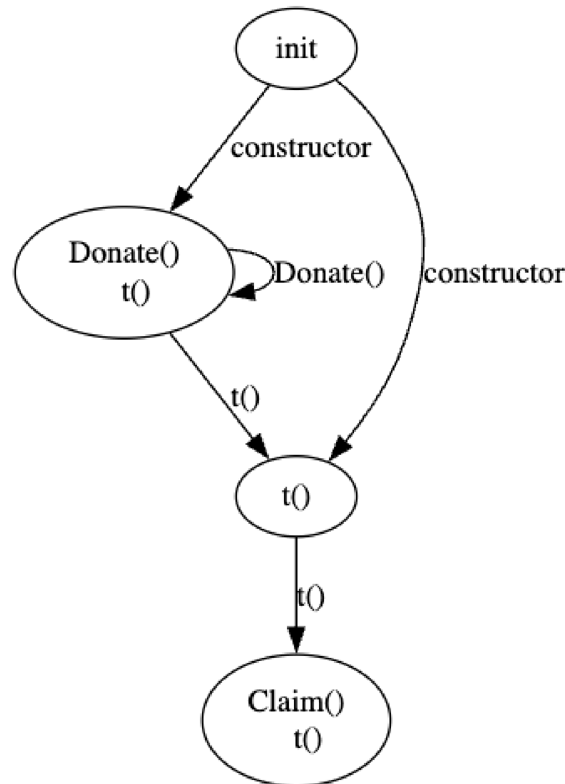

Fig. 2.3: Ejemplo de sobreaproximación en una EPA

2.3. Abstracciones por predicados para contratos inteligentes

Las EPAs constituyen un caso particular de abstracciones por predicados. Esta clase de abstracciones ha sido previamente aplicada al análisis de contratos inteligentes en [29], donde se evaluó su utilidad para validar la correspondencia entre contratos inteligentes y sus requerimientos.

Además de las EPAs, este trabajo introduce otro tipo de abstracción por predicados, basada en el estado del contrato en distintos momentos de su ejecución.

Para generar este tipo de abstracciones, a las que llamaremos abstracciones de tipo States, se deciden de forma manual los diferentes estados del programa y se pueden introducir predicados arbitrarios para particionar con mayor detalle el conjunto de estados abstractos. Cuanto más predicados se introduzcan, en general más fino será el análisis y

la abstracción.

Aquí los estados no representan funciones permitidas y no permitidas, sino que tienen un significado más cercano al entendimiento humano, definido por el programador en base a los predicados elegidos. De esta manera, se podría hacer más fácil para un desarrollador interpretar la abstracción y así entender en mayor detalle el posible comportamiento del contrato.

Un ejemplo destacado que ilustra la utilidad de las abstracciones basadas en estados para comprender el comportamiento de un contrato inteligente es el contrato *Auction*, presentado previamente en la sección 2.1.1.

La abstracción basada en estados generada para este contrato se encuentra en la figura 2.4.

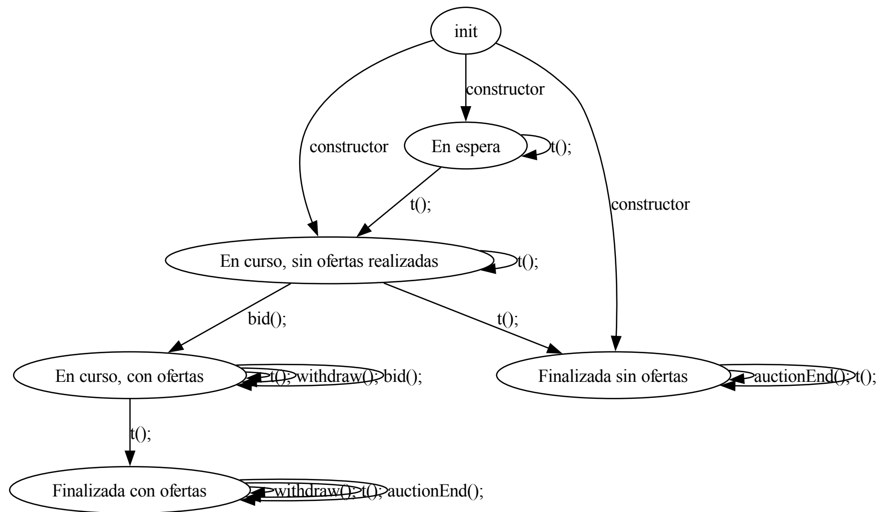

Fig. 2.4: Abstracción States para el contrato Auction

Es importante destacar que la abstracción incorpora una función denominada `t`, introducida en trabajos previos para modelar aspectos temporales de los contratos inteligentes. Esta función resulta especialmente útil en contratos donde el estado y el comportamiento dependen del valor de `block.number`, ya que permite simular su avance de manera controlada en contextos distintos a los de una ejecución del contrato en la red real. Para lograrlo, se introduce una variable adicional denominada `blockNumber`, cuyo propósito es mantener un contador interno del número de bloque. La función `t()` incrementa esta variable en una unidad cada vez que se ejecuta, permitiendo emular el paso del tiempo dentro del contrato sin depender de la red subyacente. En el caso particular de la subasta, se utiliza porque la duración de la misma está definida por la cantidad de bloques añadidos a partir de un punto inicial.

Gracias al uso de estados y predicados formales, los cuales se etiquetan en lenguaje natural para facilitar su comprensión humana, el comportamiento del contrato se podría volver más fácil de interpretar y explicar simplemente observando la abstracción.

Por ejemplo, si se desea aumentar la granularidad de la abstracción para distinguir un estado en el que la subasta ha finalizado, pero aún hay postores con pujas superadas que

no han retirado, se podría incluir un predicado adicional que determine si existen pujas sin reclamar.

En este caso, el estado en el que la subasta ha terminado y se realizaron pujas ("*Finalizada con ofertas*") se dividiría en dos: uno en el que hay pujas por retirar y otro en el que todas las pujas han sido reclamadas. Este predicado es fácil de implementar gracias a la variable `pendingReturnsCount`. Para lograr la nueva abstracción, se duplica el predicado formal que representa a la subasta finalizada con ofertas y se agrega la condición `pendingReturnsCount == 0` en un caso y `!(pendingReturnsCount == 0)` en el otro.

Entonces, a modo de ejemplo, el nuevo predicado que distingue el caso en el que la subasta ha finalizado y aún existen pujas por retirar es el presentado en 2.2.

```
highestBidder != address(0x0) && time >= auctionStart && time >
    (auctionStart + biddingTime) && !(pendingReturnsCount == 0)
```

Listing 2.2: Refinamiento - Finalizada con ofertas (con retiros pendientes)

El resultado de esta refinación se muestra en la figura 2.5.

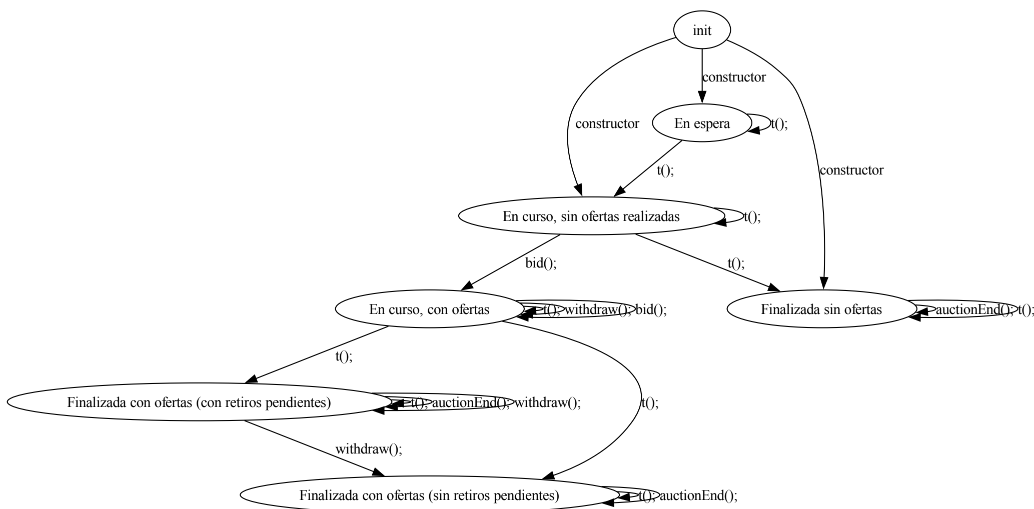

Fig. 2.5: Abstracción States para el contrato Auction con refinamiento

Este ejemplo evidencia la utilidad de las abstracciones basadas en estados, así como su capacidad para explorar y representar distintos escenarios posibles en la ejecución del contrato inteligente.

2.4. Echidna

El *fuzzing* [6] es una técnica de detección de vulnerabilidades de programas que consiste en ir generando automáticamente entradas aleatorias para las funciones del programa que se quiere analizar, hasta que alguna de esas entradas haga que el programa termine con un error o tenga un comportamiento inesperado.

Generalmente se parte de una semilla (conjunto inicial de inputs) y a partir de ella se van construyendo nuevas instancias de entrada.

El propósito principal de un *fuzzer* es generar aserciones “violentadas”, identificar buffer overflows, memory leaks, y otros errores similares. Adicionalmente, si se conoce alguna propiedad específica que debe cumplir el sistema bajo prueba, el *fuzzing* puede enfocarse en tratar de falsificar dicha propiedad.

Existen distintos tipos de *fuzzing*, cuya clasificación se puede basar en distintos criterios, como el nivel de conocimiento del sistema bajo prueba (*blackbox*, *whitebox* [27] y *greybox fuzzing* [1][7]), el método de generación de inputs (basado en mutaciones, en generación [32] o en gramáticas [26]) o el uso de retroalimentación.

En el marco de los contratos inteligentes, Echidna [4] es una de las herramientas de *fuzzing* más utilizadas. Esta herramienta usa campañas de *grammar-based fuzzing* basadas en la estructura del contrato para falsificar predicados predefinidos por el usuario (*property based testing*) o lograr violaciones de aserciones (*assertion testing*). En adición, como monitoriza la ejecución del contrato mientras realiza las pruebas y puede utilizar la cobertura de código para guiar la generación de inputs, se puede considerar que es una herramienta de *greybox fuzzing*.

A modo de ejemplo, un usuario podría colocar la siguiente función en su contrato y Echidna se encargaría de ejecutar funciones del contrato observando si este predicado es falso en algún momento:

```
function echidna_check_balance() public returns (bool) {
    return(address(this).balance >= 20);
}
```

Listing 2.3: Query Echidna

Al iniciar una campaña de *fuzzing*, la herramienta utiliza el analizador estático Slither [24] [11] para extraer información crítica del código, lo que le permite comprender la estructura y las propiedades del contrato. A partir de esta información, crea una serie de transacciones que se ejecutan de manera secuencial. Si alguna de estas transacciones logra violar una propiedad o provocar un fallo en un assert, la herramienta registra este comportamiento. Una vez completada una secuencia de transacciones, Echidna restaura el estado del contrato a su condición original y procede a generar una nueva secuencia, repitiendo este proceso hasta llegar a alguna condición de corte predeterminada.

Para moldear la campaña de *fuzzing* según las necesidades de cada usuario, se pueden utilizar más de 30 parámetros de configuración, entre los que se destacan `testMode`, `testLimit`, `balance`, `seqLen` y `workers`. Estos parámetros tienen valores determinados por los desarrolladores de Echidna por defecto, pero le ofrecen mucha flexibilidad a la herramienta y cada usuario puede configurarlos a su conveniencia. A continuación, se detallan los parámetros usados en este trabajo:

- **testMode**: en nuestro caso usamos el modo “assertion”, que detecta fallas en los asserts del contrato.

- **testLimit**: es la cantidad de transacciones totales que se generan durante la campaña de *fuzzing*.

- **seqLen**: es la longitud de las secuencias de llamadas a las funciones del contrato que se generan en la campaña de *fuzzing*.

- **balanceContract**: es la cantidad de ether que se le asigna al contrato al desplegarlo.

- **workers**: es la cantidad de procesos que se ejecutan, con el objetivo de realizar transacciones en paralelo.

En cuanto al `testMode`, Echidna usa el modo “property” por defecto, en el que se definen funciones especiales con el prefijo `echidna_` que se ejecutan al final de cada transacción para chequear si se cumplen ciertas propiedades. Un ejemplo de estas funciones especiales se puede ver en el listing 2.3. El problema que surge con este modo es que estas funciones no permiten recibir parámetros, lo que impide que se puedan evaluar propiedades que dependan de los argumentos de las funciones. Además, este modo revierte la transacción al estado inicial si detecta que una de las funciones especiales fue falsificada. En la mayoría de los casos se recomienda usar “assertion testing” cuando se requieren argumentos y cuando se esperan cambios en el estado del contrato en cada transacción. Es por estos motivos que en este trabajo se utilizó el modo “assertion”.

La ejecución que hace Echidna de los contratos es por medio de hevm, una implementación de la EVM específicamente para *testing*, que es utilizada por Echidna como motor de ejecución. Tiene la gran ventaja sobre la EVM original de que no es necesario gastar criptomonedas para realizar las ejecuciones de los contratos.

Al incluir los predicados que deseamos probar dentro del contrato en cuestión, obtenemos acceso total a su estado interno. Esta práctica, en este contexto, se denomina *internal testing*. Para implementar esta estrategia, es fundamental contar con el código fuente del contrato. Sin acceso a este, aún sería posible realizar pruebas externas, pero sin la capacidad de interactuar con las variables internas, lo que incrementaría la complejidad en la configuración de las campañas de *fuzzing*.

3. DISEÑO DE LA HERRAMIENTA

El aporte principal de este trabajo consiste en el diseño e implementación de una herramienta capaz de construir abstracciones por predicados para contratos inteligentes, de manera automática, usando Echidna.

Afortunadamente, gracias a trabajos anteriores[34], contamos con una herramienta similar que genera las abstracciones usando una herramienta de verificación formal (VeriSol [17]) en vez de un *fuzzer* (Echidna). Es común que a la hora de poner a prueba el comportamiento de programas, se combinen ambos métodos de *testing*, ya que cada uno tiene sus ventajas para encontrar ciertos tipos de errores específicos. Por ende, como aporte adicional, el trabajo se propone combinar ambas herramientas, VeriSol y Echidna, en un mismo programa capaz de generar las abstracciones tanto utilizando una herramienta como la otra, y hasta en algunos casos, corriendo ambas de forma conjunta en una misma ejecución.

Antes de comenzar a hablar de la implementación concreta del programa, es necesario mencionar que las abstracciones generadas pueden ser de dos tipos, dependiendo de las necesidades del usuario. Un tipo es EPA y el otro es States.

3.1. Modo EPA

Como ya se mencionó en la sección 2.2, cada nodo de la abstracción representa un conjunto de estados concretos del programa en los que ciertas funciones están habilitadas y ciertas otras funciones no. Las transiciones entre estados, es decir los ejes del grafo, representan la posibilidad de pasar desde un estado concreto hacia otro ejecutando una función.

Por ejemplo, en la figura 3.1 se ve la situación en la que estamos en un estado en el que se puede ejecutar las funciones f1 y f2, pero no la f3 y luego de ejecutar la función f1 podemos pasar a un estado en el que se permiten las 3 funciones.

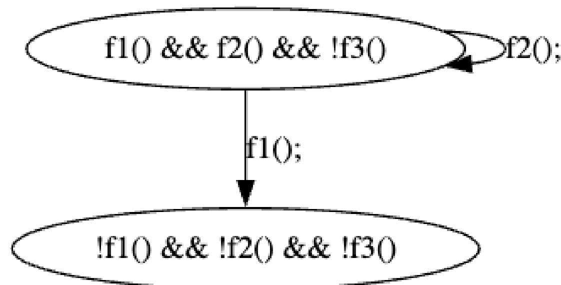

Fig. 3.1: Ejemplo de transición en EPA

En este caso, Solidity cuenta con las cláusulas `require`, que son de gran utilidad para determinar, en cada función, qué condiciones deben cumplirse para que la misma pueda ejecutarse. Se puede pensar en las cláusulas `require` como precondiciones de las funciones, aunque es importante destacar que su cumplimiento no garantiza necesariamente que la función se ejecutará correctamente. Es decir, si bien funcionan como una primera barrera

de validación, no constituyen formalmente las precondiciones de la función en un sentido estricto.

Un ejemplo común de uso de estas cláusulas es la necesidad de restringir la ejecución de una determinada función al dueño del contrato (es decir, quien lo desplegó). Para esto, en la primera línea de la función, se agrega un `require(msg.sender == owner)`. El uso de estas cláusulas puede verse en el contrato de ejemplo ya presentado en la figura 2.1.

La *keyword* `require` fue introducida en Solidity en la versión 0.4.10, por lo que es posible encontrar contratos desarrollados previamente en los que la misma funcionalidad se lograba utilizando, en lugar de `require(condition)`, la construcción `if(!condition) revert()`.

Uniendo esto con nuestra idea de abstracción basada en "*enabledness*", cada estado de la abstracción tendrá su precondición, que corresponde a la unión de todas las precondiciones de las funciones habilitadas en ese estado.

Es decir, suponiendo que se cuenta con la abstracción para un contrato que tiene tres funciones y se está analizando un estado en el cual están habilitadas únicamente las funciones `f1` y `f2`, entonces la precondición de ese estado será el predicado `precondiciones(f1) && precondiciones(f2) && !precondiciones(f3)`.

Un aspecto a considerar es que la cantidad de posibles estados abstractos para un contrato con n funciones es 2^n. Esto puede hacer que la técnica sea intratable o excesivamente lenta, afectando la escalabilidad. En la práctica, sin embargo, la cantidad de estados abstractos factibles suele ser considerablemente menor. En la sección 3.5 se presentan múltiples técnicas para reducir esta cantidad sin perder estados valiosos.

Para construir estas abstracciones tal como fueron descritas, se utilizan funciones del siguiente tipo.

```
// Transitions
function from_i_to_j_by_k() payable public{
    require(precondition_state_i);
    function_k();
    assert(!preconditions_state_j);
}
```

Listing 3.1: Función de consulta (*transitions*)

Estas funciones, ocasionalmente llamadas *queries*, junto con las funciones del contrato, van a estar siendo ejecutadas permanentemente por la herramienta, y si en algún momento se falsifica el assert (i.e. `assert(false)`), eso quiere decir que `preconditions_state_j` es `true`, por lo que partiendo del estado i, ejecutando la función k, se puede llegar al estado j.

De esta manera se van a ir construyendo las abstracciones.

En particular, para definir los posibles estados iniciales, se construyen funciones de la forma presentada en 3.2 y en el contrato se eliminan todas las funciones excepto el constructor.

De esta manera, se va a ir ejecutando el constructor con diferentes parámetros y se van a ir chequeando las funciones de este estilo para determinar si algún assert falla. En ese caso, si el assert dentro de la función `constructor_to_i()` falla, podremos afirmar que luego de ejecutar el constructor podemos llegar al estado de índice i.

```
// Init (a donde se llega desde el constructor)
function constructor_to_i() payable public {
    assert(!precondition_state_i)
}
```

Listing 3.2: Función de consulta (*init*)

3.2. Modo States

El otro tipo de abstracciones con las que se va a trabajar son las abstracciones de tipo States, presentadas en la sección 2.3.

Como se mencionó previamente, para generar estas abstracciones, las *queries* no son obligatoriamente creadas con las cláusulas require de las funciones del contrato, sino que se utilizan predicados arbitrarios que el programador puede definir para particionar el conjunto de estados abstractos de la manera que considere más conveniente.

Por ejemplo, en la figura 3.3 se ve la diferencia entre la abstracción de un contrato en modo EPA y en modo States. El contrato es el siguiente y representa una partida de piedra, papel o tijera.

```
contract RockPaperScissors {
    address payable player1;
    address payable player2;
    address payable owner;

    int p1Choice = -1;
    int p2Choice = -1;
    uint[3][3] public payoffMatrix;

    constructor(address payable _p1, address payable _p2, address payable
        _owner) {
        player1 = _p1;
        player2 = _p2;
        owner = _owner;

        //Rock - 0, Paper - 1, Scissors - 2
        payoffMatrix[0][0] = 0;
        payoffMatrix[0][1] = 2;
        payoffMatrix[0][2] = 1;
        payoffMatrix[1][0] = 1;
        payoffMatrix[1][1] = 0;
        payoffMatrix[1][2] = 2;
        payoffMatrix[2][0] = 2;
        payoffMatrix[2][1] = 1;
        payoffMatrix[2][2] = 0;
    }

    function choicePlayer1(uint choice) public {
        require(msg.sender == player1);
        require(p1Choice == -1);
        p1Choice = int(choice % 3);
    }
```

```
    function choicePlayer2(uint choice) public {
        require(msg.sender == player2);
        require(p2Choice == -1);
        p2Choice = int(choice % 3);
    }

    function determineWinner() public {
        require(p1Choice != -1 && p2Choice != -1);
        uint winner = payoffMatrix[uint(p1Choice)][uint(p2Choice)];
        if(winner == 1) {
            player1.transfer(address(this).balance);
        }
        else if(winner == 2) {
            player2.transfer(address(this).balance);
        }
        else {
            owner.transfer(address(this).balance);
        }
    }
}
```

Listing 3.3: Contrato RockPaperScissors

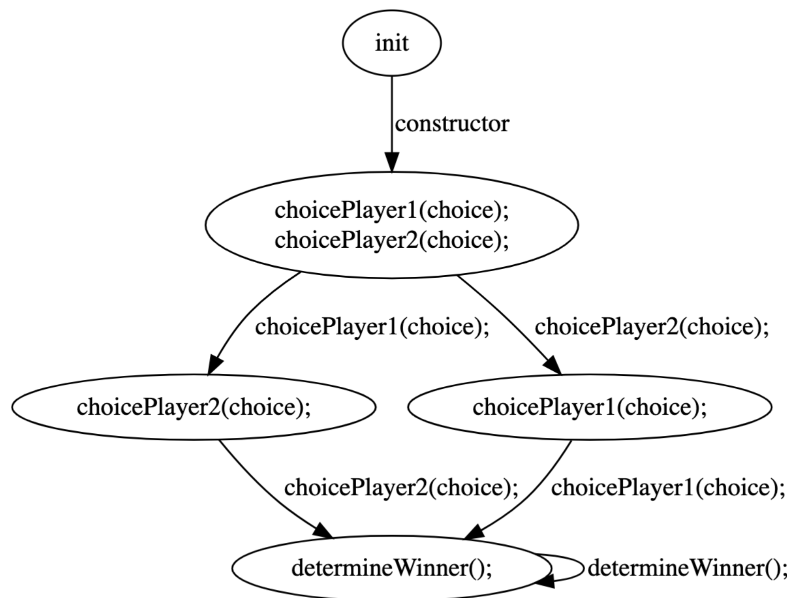

Fig. 3.2: EPA para el contrato RockPaperScissors

Podemos ver que, en la abstracción en modo States, tenemos mucha más información sobre el contrato gracias a los siguientes predicados extra, que nos sirven para particionar el estado que antes estaba representado por la función `determineWinner()` en 3 estados distintos:

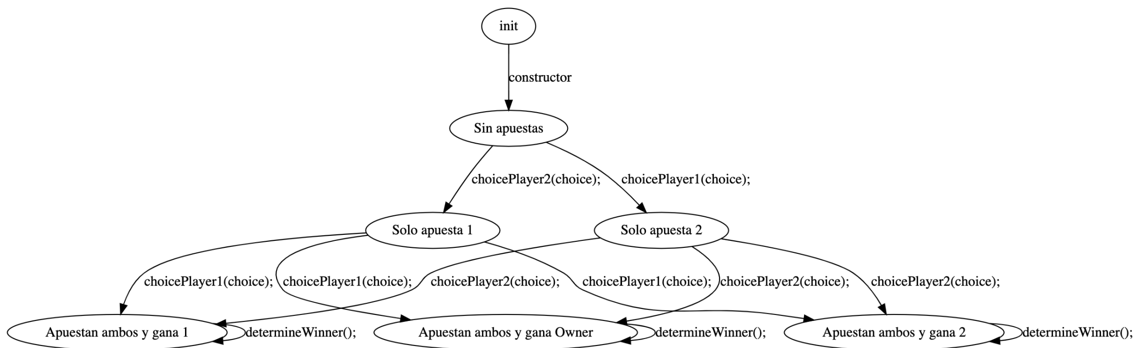

Fig. 3.3: Abstracción States para el contrato RockPaperScissors

- `payoffMatrix[uint(p1Choice)][uint(p2Choice)] == 1 // gana player1`
- `payoffMatrix[uint(p1Choice)][uint(p2Choice)] == 2 // gana player2`
- `payoffMatrix[uint(p1Choice)][uint(p2Choice)] == 0 // empate`

Los estados, además de poder ser definidos por un humano, pueden ser extraídos directamente del código del contrato, siempre y cuando el mismo cuente con una variable que se encargue de especificar en qué estado se encuentra el contrato en cada momento. Esto haría posible una capa más de automatización en la herramienta.

Por ejemplo, el contrato RoomThermostat tiene un `enum StateType` { `Created, InUse` } y una variable `StateType public State`, cuyo valor va cambiando a lo largo de la ejecución del contrato dependiendo de qué funciones son ejecutadas. Esos estados son usados para generar la abstracción de tipo States que se puede ver en la figura 3.4. Un ejemplo de *query* para este contrato, que representa la transición entre el estado `Created` y el estado `InUse` por medio de la función `StartThermostat()` se presenta en el listing 3.4.

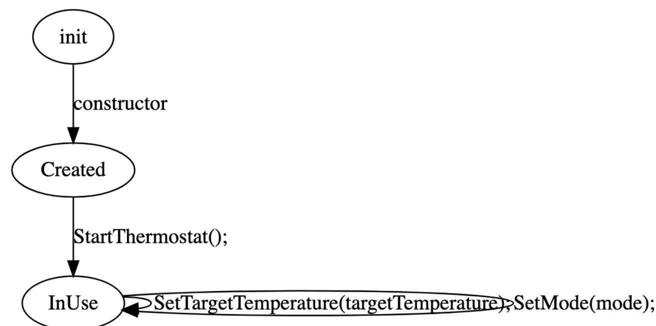

Fig. 3.4: Abstracción States para el contrato RoomThermostat

```
function from_Create_to_InUse_by_StartThermostat() payable public {
    require(State == StateType.Created);
    StartThermostat();
    assert(!(State == StateType.InUse));
}
```

Listing 3.4: Función de consulta utilizando variables de estado

3.3. Implementación

La herramienta opera de la siguiente manera: recibe como entrada un contrato escrito en Solidity, acompañado de un archivo de configuración y de un conjunto de parámetros adicionales que serán presentados posteriormente, y genera como salida una abstracción que representa el comportamiento del contrato. Durante su ejecución, crea automáticamente las consultas mostradas en 3.1 y 3.2, ejecuta Echidna para intentar falsificarlas, recoge los resultados generados, los organiza y, a partir de ellos, construye la abstracción, que en términos prácticos es un multigrafo con ejes dirigidos y posibles ciclos.

Al ya contar con una herramienta muy similar desarrollada en Python, se decidió desarrollar esta herramienta en ese lenguaje para aprovechar la posibilidad de reutilizar código y de integrar ambas herramientas para que funcionen en conjunto y puedan ampliar las capacidades de cada herramienta por separado. Al combinar las herramientas se pretende lograr un mejor resultado que usando sólo una.

3.3.1. Archivo de configuración

Antes de adentrarse en el proceso de ejecución de la herramienta y en otros detalles de implementación, es fundamental comprender en detalle la estructura y el propósito del archivo de configuración que recibe como entrada.

Este archivo tiene como objetivo que el programa comprenda la estructura del contrato inteligente y de esta manera pueda generar las consultas adecuadas, respetando los nombres de las funciones, los parámetros, el nombre del contrato, las precondiciones de cada función y demás. Para facilitar al programa la lectura del archivo, el mismo también debe estar escrito en Python.

Específicamente, un archivo de configuración contiene las siguientes variables:

- El nombre del archivo en el que se encuentra el contrato inteligente a analizar.
- El nombre del contrato en cuestión.
- Las funciones a analizar, junto con sus precondiciones. Estas precondiciones se dividen en 2 grupos: las precondiciones de estado, que indican en qué condiciones debe estar el contrato para que la función se pueda ejecutar y las precondiciones de función, que están orientadas a los parámetros de la función y reflejan qué restricciones deben cumplir los argumentos (por ejemplo, una restricción común es que quien realiza la transacción sea el propietario del contrato).
- El tipo y nombre de los parámetros de cada función.

Adicionalmente, para generar la abstracción de tipo States, en el archivo de configuración se deben especificar los nombres de los estados que se quieren observar acompañados con un identificador y los predicados que definen a esos estados, es decir, las condiciones necesarias para que el contrato esté en cada estado.

Estos elementos deben ser proporcionados por el usuario y deben corresponderse exactamente con los contenidos del contrato inteligente. Este proceso podría ser automatizado utilizando tanto un analizador (o *parser*) de código Solidity como por medio de expresiones regulares en cualquier otro lenguaje.

En el listing 3.5 se puede ver, como caso representativo, un archivo de configuración para el contrato SimpleAuction.

```
# Nombre del archivo que contiene al contrato.
fileName = "SimpleAuction.sol"

# Nommbre del contrato.
contractName = "SimpleAuction"

# Funciones a analizar.
functions = [
    "bid();",
    "withdraw();",
    "auctionEnd();",
    "t();"
]

# Precondiciones de las funciones.
statePreconditions = [
    "time <= (auctionStart + biddingTime)",
    "pendingReturnsCount > 0",
    "!ended && time >= (auctionStart + biddingTime)",
    "true"
]

# Precondiciones de los argumentos de las funciones.
functionPreconditions = [
    "msg.value > highestBid",
    "true",
    "true",
    "true"
]

# Parametros de las funciones.
functionVariables = "address refundee"

# Identificadores de estados para abstraccion de tipo States.
statesModeState = [[1,0,0,0], [0,2,0,0], [0,0,3,0], [0,0,0,4]]

# Nombres de los estados para abstraccion de tipo States.
statesNamesModeState = [
    "En curso, sin ofertas realizadas",
    "Finalizada sin ofertas",
    "En curso, con ofertas",
    "Finalizada con ofertas"
]

# Predicados de los estados para abstraccion de tipo States.
statePreconditionsModeState = [
    "!ended && highestBidder == address(0x0) && pendingReturnsCount == 0",
    "ended && highestBidder == address(0x0) && pendingReturnsCount == 0",
    "!ended && highestBidder != address(0x0)",
    "ended && highestBidder != address(0x0)",
]
```

Listing 3.5: Archivo de configuración para el contrato SimpleAuction

3.3.2. Ejecución

En esta sección se presentan en detalle todos los parámetros de entrada que recibe el programa, explicando su propósito y la manera en que influyen en la ejecución. Asimismo, se describen las distintas etapas que sigue el programa, desde la lectura de los datos de entrada hasta la generación de los resultados.

Los parámetros utilizados son los siguientes:

1. **Contract config**

   El archivo de configuración necesario para ejecutar el programa, como se mencionó previamente. Notar que desde este archivo se referencia al contrato que se quiere analizar.

2. **Modo**: e (epa) o s (states).

   Indica el tipo de abstracción que quiere generar el usuario.

3. **Tool**: echidna o verisol.

   Como se ha descrito previamente, el programa permite generar las abstracciones tanto utilizando un *fuzzer* (Echidna) como un verificador formal (VeriSol). Es por esto que se le da la posibilidad al usuario de elegir qué herramienta utilizar. De todas maneras, el enfoque principal de esta investigación es el uso de Echidna.

   En caso de elegir Echidna, como se mencionó en la sección 2.4, el testMode será siempre “assertion”.

4. **Test limit (cuando tool = Echidna)**: La cantidad total de transacciones a ejecutar en la campaña de *fuzzing* de Echidna. Cuanto mayor sea su valor, más tardará la ejecución.

5. **Parámetros extra (cuando tool = VeriSol)**: txbound y timeout.

   El txbound (*transaction bound*) es un parámetro que se utiliza para limitar la cantidad de transacciones o *loop unrollings* máximos que utiliza el verificador al buscar asserts que fallen. *Loop unrolling* es una técnica muy común utilizada en el contexto de ejecución simbólica y de verificación formal que consiste en “desenrollar” iteraciones de un ciclo para que sean analizadas secuencialmente. Por otro lado, el timeout es un parámetro que se usa para asegurarnos de que VeriSol no se quede ejecutando durante un tiempo excesivo. Específicamente, indica cuánto tiempo se debe estar analizando cada consulta, en segundos. Por defecto, `txbound=8` y `timeout=600`.

6. **Flags de optimización**. El programa, por defecto, realiza 3 optimizaciones, que vamos a llamar reduce, reduce_equal y reduce_true. El usuario tiene la posibilidad de desactivar algunas de estas optimizaciones, que son explicadas en profundidad en la sección 3.5, utilizando las siguientes *flags*.

   a) -rs: desactiva el reduce.
   b) -rt: desactiva el reduce_true.
   c) -re: desactiva el reduce_equal.
   d) -rte: desactiva el reduce_true y el reduce_equal.
   e) -ra: desactiva todas las optimizaciones.

Por ejemplo, dos posibles llamadas al programa podrían ser:

```
$ python3 main.py RockPaperScissorsConfig e echidna test_limit=100000 -rs
$ python3 main.py CrowdfundingConfig s verisol txbound=4 time_out=30
```

Listing 3.6: Posibles ejecuciones del programa

El primer comando va a generar una abstracción en modo EPA (parámetro e) usando la herramienta Echidna (parámetro echidna) con un *test limit* de 100.000. Además, se desactiva la optimización reduce (*flag* -rs). El segundo comando va a generar una abstracción en modo States (parámetro s) usando la herramienta VeriSol (parámetro verisol) con un *transaction bound* de 4 y un *timeout* de 30 segundos, usando las tres optimizaciones disponibles por defecto.

La herramienta, en un modo generalizado, sigue las siguientes etapas:

1. En base a los parámetros de entrada, el programa se encarga de crear contratos que luego van a ser ejecutados internamente por la herramienta de fondo. En general se crea uno denominado *init* que representa los estados a los que se puede llegar luego de ejecutar el constructor del contrato y otro (uno o varios) denominado *transitions* que analiza todas las posibles transiciones entre estados "internos" del contrato.

2. Una vez creados esos contratos, también en base a algunos parámetros de entrada como el *budget*, que puede ser, dependiendo de la herramienta utilizada en el *backend*, tanto `testLimit` como `txbound` y `timeout`, se analizan los contratos creados y se obtienen los resultados.

3. Con esos resultados, que representan transiciones entre estados, se crea la abstracción solicitada y es devuelta al usuario para su posterior análisis. Opcionalmente, también se cuenta con la posibilidad de imprimir los resultados por consola.

Durante el transcurso de estas 3 etapas (creación, ejecución, construcción de la abstracción), se realizan una serie de optimizaciones que permiten reducir el tiempo de ejecución del programa y la cantidad de estados que se deben analizar. Estas optimizaciones son explicadas en la sección 3.5. Asimismo, para garantizar el correcto funcionamiento de la herramienta, se consideraron diversos detalles de implementación, los cuales se describen con mayor detalle en la sección 3.6.

3.4. Integración con VeriSol

Tal y como se ha señalado, ya se contaba con una herramienta que genera abstracciones por predicados utilizando VeriSol.

Parte del trabajo consiste en integrar ambas herramientas para que el usuario pueda elegir qué herramienta utilizar, simplemente especificando el valor del parámetro `tool` al ejecutar el programa.

Dependiendo de la elección del usuario, el programa dispone de clases específicas para la creación y ejecución de los contratos intermedios necesarios para generar la abstracción final, como `VerisolContractCreator`, `VeriSolRunner`, `EchidnaContractCreator` y `EchidnaRunner`.

Además, existen casos en los que se aprovechan las capacidades de ambos artefactos de manera combinada en la misma ejecución. Esto ocurre cuando se emplea la optimización que descarta estados inalcanzables, que siempre se realiza con VeriSol, y luego se utiliza Echidna para generar las abstracciones. Esta optimización se detalla en la sección 3.5.3.

Por último, al combinar ambas herramientas, fue necesario tener en cuenta las diferencias de versiones entre los contratos que puede analizar VeriSol y los que puede analizar Echidna. Los detalles al respecto se encuentran en la sección 3.6.

3.5. Optimizaciones

Como se mencionó anteriormente, el programa cuenta con la posibilidad de realizar diversas optimizaciones. En general, el objetivo de estas optimizaciones es reducir el espacio de estados a analizar, ya que este espacio, sin optimizaciones, es exponencial con respecto a la cantidad de funciones del contrato.

Las mismas proceden del trabajo realizado en [34] y son las presentadas a continuación.

3.5.1. Manejo de funciones con precondición `true`

Tal como se ha indicado previamente, cada función tiene una serie de precondiciones que deben cumplirse para que pueda ser ejecutada. Esta información, además de estar presente en general en el contrato en forma de cláusulas `require`, también se encuentra en el archivo de configuración que es provisto a la herramienta.

Puede ocurrir que alguna de las funciones del contrato no tenga ninguna precondición explícita. En estos casos, se dice que la precondición es el predicado `true`. Entonces, esta función va a estar disponible para su ejecución en todo momento, en cualquier estado concreto del contrato. Es por esto que, con el fin de reducir el espacio de estados a analizar, se decide quitar del análisis a todos los estados abstractos que no tengan habilitada a esta función.

Esta optimización reduce la cantidad de estados a analizar por la mitad por cada función que tenga como precondición el predicado `true`, ya que, de los 2^n estados iniciales, la mitad tiene habilitada a la función y la otra mitad la tiene deshabilitada.

En nuestra herramienta, esta optimización lleva el nombre de `reduce_true`. Está activada por defecto y se puede desactivar utilizando la *flag* `-rt` al correr el programa.

3.5.2. Combinación de funciones con las mismas precondiciones

Al igual que en el caso anterior, esta optimización se basa en las precondiciones de las funciones del contrato y es usada cuando se quieren generar EPAs.

La misma consiste en identificar distintas funciones que tengan las mismas precondiciones. En ese caso, se decide agruparlas y que en todos los estados abstractos que analice la herramienta estén o bien habilitadas o bien deshabilitadas todas las funciones del grupo, ya que no tendría sentido tener un estado en el que una de esas funciones esté permitida y otra no.

En conclusión, se descartan los estados abstractos que tengan habilitada a alguna función y deshabilitada a otra con las mismas precondiciones. La cantidad de estados a analizar, al igual que en el caso anterior, se reduce a la mitad por cada par de funciones que tengan las mismas precondiciones.

El nombre que recibe esta optimización dentro de la herramienta es `reduce_equal`. Está activada por defecto y se puede desactivar utilizando la *flag* `-re`.

3.5.3. Descarte de estados inalcanzables

La última optimización que se va a usar también consiste en reducir el espacio de búsqueda descartando estados abstractos que no son alcanzables por el contrato.

A diferencia de las dos optimizaciones ya presentadas, que son realizadas de antemano y no dependen de la ejecución del contrato en ningún sentido, esta optimización requiere que el contrato sea evaluado por un analizador que lo ejecute (ya sea estática o dinámicamente) para determinar qué estados son alcanzables y cuáles no.

Las funciones que se van a ejecutar para descartar los estados inalcanzables para el contrato tienen la siguiente estructura:

```
// Preconditions
function vci() payable public {
    require(precondition_state_i);
    assert(false);
}
```

Listing 3.7: Función de consulta (preconditions)

De esta manera, si la herramienta de análisis logra llegar al `assert(false)`, eso quiere decir que el estado i es alcanzable y por lo tanto no se lo puede descartar.

Para poner en práctica esta optimización, se opta por utilizar VeriSol, ya que puede identificar si un estado es alcanzable o declararlo inalcanzable.

Sin embargo, esto depende de varios factores, como el valor del parámetro txbound y el timeout. Por ejemplo, si txbound tiene un valor de k, VeriSol solo garantiza que cierto estado es alcanzable si se puede llegar a él ejecutando a lo sumo k transacciones. Además, el tiempo de análisis también influye en la exploración del espacio de estados. Por ello, es fundamental elegir estos parámetros de manera adecuada, y en caso de tener constancia de que el contrato tiene estados que únicamente se pueden alcanzar ejecutando más de k transacciones, se debe aumentar el valor de txbound.

A modo de ejemplo, en 3.8 se puede ver un contrato en el que una de sus funciones ejecuta un ciclo de 12 iteraciones. Si se ejecuta VeriSol con `txbound=12`, no se va a poder llegar al estado en el que se haya ejecutado este ciclo (`State.ComputeTotal`), ya que se alcanzó el límite de *loop unrollings*. Sin embargo, si se ejecuta con `txbound=13`, se va a poder llegar a este estado. Este resultado se puede ver en detalle en la figura 3.5.

```
contract DefectiveComponentCounter {

    enum StateType {Create, ComputeTotal}

    StateType State;
    address Manufacturer;
    int[12] DefectiveComponentsCount;
    int Total;

    constructor(int[12] memory defectiveComponentsCount) public {
        Manufacturer = msg.sender;
        DefectiveComponentsCount = defectiveComponentsCount;
```

```
        Total = 0;
        State = StateType.Create;
    }

    function ComputeTotal() public {
        require(Manufacturer == msg.sender);
        require(State != StateType.ComputeTotal);

        // Ciclo que se ejecuta 12 veces.
        for (uint i = 0; i < 12; i++){
            Total += DefectiveComponentsCount[i];
        }

        // Si txbound <= 12, no se llega a este punto del programa,
        // por lo que el contrato nunca pasa al estado ComputeTotal.
        State = StateType.ComputeTotal;
    }

    function GetDefectiveComponentsCount() public view returns (int[12]
        memory) {
        return DefectiveComponentsCount;
    }
}
```

Listing 3.8: Contrato DefectiveComponentCounter

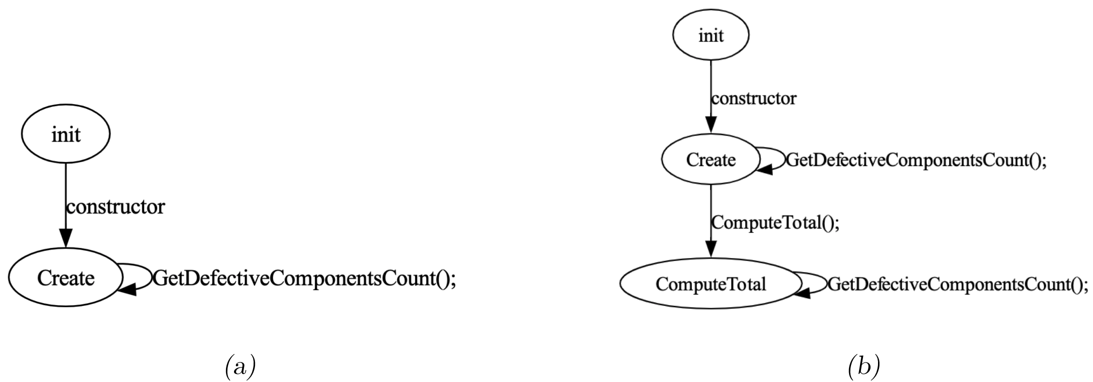

Fig. 3.5: DefectiveComponentCounter - TxBound=12 vs. TxBound=13

3.6. Consideraciones

3.6.1. Diferencia de versiones

Un factor a considerar durante el desarrollo fue la selección de la versión del compilador de Solidity adecuada para ejecutar la herramienta.

El compilador[13] evoluciona constantemente con cada versión, incorporando mejoras y nuevas características. Al momento de redactar este trabajo, la versión más reciente es la 0.8.28.

Es fundamental determinar cuál versión es más adecuada para la herramienta, considerando que algunas actualizaciones introducen *breaking changes* (es decir, modificaciones que rompen la compatibilidad con versiones anteriores). En este contexto, se debe tener en cuenta que VeriSol sólo es compatible con versiones del compilador hasta la 0.5.x, inclusive. Por otro lado, Echidna, que sigue siendo mantenido activamente por sus desarrolladores, funciona mejor con versiones más recientes, como las 0.8.0 en adelante.

En particular, se presenta un problema importante al usar Echidna con la versión 0.5.0: cuando se ejecuta un `assert(false)` o se produce un error por acceso fuera de rango, el código de error devuelto es el mismo. Esto dificulta distinguir por qué se detuvo la transacción.

Por esta razón, se decidió utilizar la versión 0.8.0 del compilador para ejecutar el *fuzzer*. Sin embargo, cuando un usuario proporciona un contrato compatible con la versión 0.8.0 y elige utilizar VeriSol, el programa cuenta con una función que adapta automáticamente el contrato para que sea compatible con la versión 0.5.0.

Los cambios realizados por esta función son los siguientes:

- Eliminar todas las declaraciones de `abstract`.
- Quitar los modificadores `virtual`.
- Eliminar las funciones marcadas como `payable()`.
- Reemplazar la función `receive()` con una función sin nombre.

Estos ajustes son realizados por la función `fix_for_version_5` de la clase `VerisolContractCreator`.

3.6.2. Separar en múltiples contratos

Otro aspecto que fue necesario considerar al momento de implementar la herramienta fue de qué manera crear los contratos que luego serían analizados por las herramientas de fondo. De acuerdo a lo ya mencionado en el listing 3.1, por cada par de estados abstractos i y j, se crea una *query* de la forma `from_i_to_j_by_k`, donde i y j son los estados abstractos y k es la función a ejecutar.

En un principio, lo más sencillo es insertar todas estas funciones dentro del contrato para el cual se quiere generar la abstracción y que luego la herramienta de fondo se encargue de ejecutarlas. Sin embargo, esto puede llevar a que el contrato se vuelva muy grande y difícil de analizar, por lo que se estudió la posibilidad de separar estas *queries* en múltiples contratos.

Se realizó un analisis al respecto y se llegó a la conclusión de que, para contratos grandes (i.e. con cantidad de queries mayor a 250) la mejor manera de proceder es separar

las *queries* en múltiples contratos. Esto trae un beneficio tanto en el tiempo de ejecución como en la capacidad de exploración de los posibles estados y transiciones alcanzables por herramienta de fondo. La razón es que, al haber menos funciones por ejecutar, cada función se ejecuta más veces (y por ende, con más parámetros distintos) y se pueden explorar más estados concretos del contrato inteligente bajo test.

Tras realizar el análisis sobre un conjunto de contratos considerablemente grandes, se decidió que la cantidad de contratos entre los que se dividen las *queries* es de 8.

3.6.3. Asserts internos

Uno de los posibles inconvenientes que pueden surgir al intentar correr la herramienta usando Echidna es que el contrato bajo análisis tenga asserts dentro de alguna de sus funciones.

```
function from_i_to_j_by_k() payable public{
    require(precondition_state_i);
    function_k();
    assert(!preconditions_state_j);
}
```

Listing 3.9: Función de consulta con un assert interno

Si la función `function_k()` tiene un assert que falla, la herramienta va a reconocer que hubo un `assert(false)` mientras estaba ejecutando la función `from_i_to_j_by_k` y por lo tanto va a considerar que el estado j es alcanzable desde el estado i, tras ejecutar la función k, lo cual no es necesariamente cierto.

Para evitar este problema, se puede modificar el código del contrato antes de correr el programa, reemplazando la expresión `assert(condition)` por `if(!condition) revert()` y de esta manera no se van a agregar transiciones que no deberían a la abstracción a causa de un assert interno. Sólo se tienen en cuenta los asserts de las *queries*.

Esta modificación no introduce ningún cambio en el comportamiento del contrato que afecte nuestro análisis. Tanto `assert(false)` como `revert()` revierten la transacción. La única diferencia es que si un assert falla, se consume todo el gas disponible en la transacción, mientras que si hay un revert o falla un require, el gas restante se devuelve al usuario.

3.6.4. Fuzzing en el constructor

Otra de las limitaciones que se encontraron al desarrollar la herramienta fue la incapacidad de Echidna de hacer *fuzzing* sobre el constructor de un contrato.

Esto se debe a una decisión de diseño de los desarrolladores, con el fin de que la herramienta sea lo más liviana posible y centrándonse en su utilización para *testing* de propiedades. Sin embargo, a nosotros nos interesa conocer todos los posibles estados del contrato, por lo que necesitamos desplegar el contrato con una gran variedad de valores de sus parámetros. De esta manera, seríamos capaces de determinar todos los posibles estados iniciales del contrato.

Para mitigar este obstáculo, se decidió implementar la lógica del constructor dentro de una función aparte, denominada `my_constructor()`, que funcione como un constructor alternativo del contrato. Es decir, nuestro programa automáticamente elimina el constructor del contrato y crea una función que tiene el mismo comportamiento que el constructor

original. A su vez, se agrega una variable booleana `hasInitialized` inicializada en falso que indica si esta función fue ejecutada o no.

Con todos estos elementos ya definidos, lo que se hace es agregar una cláusula require al nuevo constructor tal como se puede ver en 3.10.

```
function my_constructor() public {
    require(!hasInitialized);
    // Codigo del constructor original
    ...
    hasInitialized = true;
}
```

Listing 3.10: Constructor apto para fuzzing

Además, al resto de las funciones del contrato se les agrega la siguiente cláusula require:

```
function doSomething() payable public {
    require(hasInitialized);
    // Codigo de la funcion
    ...
}
```

Listing 3.11: Función con hasInitialized como condición

Así, se garantiza que el contrato no pueda ejecutar ninguna función hasta que se haya ejecutado la función `my_constructor()`, logrando simular el comportamiento del constructor original y realizando *fuzzing* en los parámetros del mismo.

4. EVALUACIÓN

4.1. Introducción

En esta sección se presenta una descripción general de los contratos utilizados para evaluar la herramienta, así como una breve enumeración de las preguntas de investigación que se buscan responder con dicha evaluación.

En síntesis, se pretende determinar si es posible generar abstracciones por predicados de forma dinámica a partir de contratos inteligentes, si las abstracciones generadas son útiles para comprender el comportamiento de los contratos y para identificar vulnerabilidades, y si se pierde información relevante al usar una herramienta como Echidna que, por su naturaleza, subaproxima.

La evaluación se realiza sobre dos conjuntos de contratos inteligentes, presentados a continuación.

- Benchmark 1: Azure Blockchain Workbench [2]

Nombre del contrato	Líneas de código	Cantidad de funciones
Asset Transfer	225	10
Basic Provenance	48	2
Defective Component Counter	33	2
Digital Locker	148	10
Frequent Flyer Rewards Calculator	50	3
Hello Blockchain	35	2
Refrigerated Transportation	129	3
Room Thermostat	48	3
Simple Marketplace	66	3

Tab. 4.1: Contratos Benchmark 1

- Benchmark 2: SmartPulse Evaluation Benchmarks [12]

Nombre del contrato	Líneas de código	Cantidad de funciones
Auction	51	4
Crowdfunding	55	4
EPXCrowdsale	171	6
EscrowVault	102	9
RefundEscrow	129	7
RockPaperScissors	66	4
SimpleAuction	50	4
ValidatorAuction	260	13

Tab. 4.2: Contratos Benchmark 2

En cuanto a algunos contratos del *benchmark 1*, estudios previos [18] [29] identificaron vulnerabilidades en su código, las cuales fueron corregidas. Como resultado, surgió una nueva versión de estos contratos, denominada *Fixed*, en la que cada contrato original que tiene un *bug* da lugar a uno nuevo con el mismo nombre seguido de la palabra *Fixed*. Los contratos que presentan esta actualización son *AssetTransfer*, *BasicProvenance*, *DefectiveComponentCounter*, *DigitalLocker*, *HelloBlockchain*, *RefrigeratedTransportation* y *SimpleMarketplace*. Estos contratos también son analizados en este trabajo.

4.2. Metodología de evaluación

Para el análisis, se ejecutará la herramienta sobre cada uno de los contratos seleccionados, utilizando Echidna como *backend*. Se llevará a cabo una ejecución con el objetivo de obtener la EPA y otra para generar la abstracción de tipo States. Asimismo, se han definido los valores de testLimits a evaluar: 1.000, 50.000 y 500.000. La variación de estos valores tiene como propósito analizar su impacto en la exploración de los estados del contrato. Se espera que a medida que el *test limit* aumente, el tiempo de ejecución se incremente, pero también se explore un mayor número de estados.

En relación con las optimizaciones empleadas, en todos los casos se utilizarán reduce_true 3.5.1 y reduce_equal 3.5.2, dado que omitirlas no proporcionaría información adicional. Es decir, su uso no conlleva el riesgo de pérdida de información relevante. Por otro lado, para cada combinación de contrato y testLimit, la ejecución se realizará en dos escenarios: uno aplicando la optimización descrita en 3.5.3 (descarte de estados inalcanzables) y otro sin emplearla. El propósito de esta optimización es reducir el espacio de estados posibles, facilitando la exploración a Echidna. Sin embargo, su aplicación implica el riesgo de descartar estados abstractos de interés. Cabe destacar que esta optimización es un ejemplo de la integración de VeriSol y Echidna en una misma ejecución.

En el caso de utilizar Echidna, se fija el valor del parámetro `seqLen` en 2 para el contrato *init* [1] (ya que sólo interesa saber en qué estado se está exactamente después de haber ejecutado el constructor) y en 100 para el contrato *transitions*[2]. El `balance` inicial del contrato es, por defecto, 0 y la cantidad de `workers` es 4, de acuerdo a las capacidades de la máquina en la que se realizaron las pruebas.

Las abstracciones obtenidas se compararán con aquellas generadas al ejecutar la herramienta utilizando únicamente VeriSol como herramienta de fondo, siguiendo la misma técnica empleada en [34], que a su vez contrastaba sus resultados con los de [29].

Se evaluará la capacidad de Echidna para replicar los resultados obtenidos por VeriSol, mediante el análisis de la cantidad de estados y transiciones en las abstracciones generadas. Adicionalmente, ejecutar la herramienta utilizando VeriSol en este contexto nos permite efectuar una comparación acerca de la eficiencia de la herramienta en cada caso, midiendo los tiempos de ejecución. El valor de *txBound* y *timeout* utilizados para estas ejecuciones son de 8 y 600, respectivamente, y se usan todas las optimizaciones disponibles.

Sumado a esto, en ciertos casos, la abstracción obtenida será comparada con el modelo conceptual desarrollado por los autores del contrato. Estos modelos están disponibles particularmente para los contratos del *benchmark 1*.

[1] Contrato creado por la herramienta con el fin de detectar los estados a los que se puede llegar luego de ejecutar el constructor del contrato bajo análisis.

[2] Contrato que analiza todas las posibles transiciones entre estados “internos” del contrato bajo análisis.

Los contratos, sus archivos de configuración y todos los resultados obtenidos están disponibles, junto con el código de la herramienta, en el repositorio del proyecto [3].

Se fija, para cada ejecución, un *timeout* total de dos horas. Todos los experimentos fueron ejecutados en un equipo con un procesador Intel Core i5 de 3.1 GHz (Dual-Core) y 8 GB de memoria RAM, bajo el sistema operativo macOS 13.7.1.

4.3. Resultados

4.3.1. Viabilidad de la generación de abstracciones mediante fuzzing

A continuación, se presentan distintas tablas con datos acerca de los resultados de todas las ejecuciones. La gran mayoría de los contratos pudo ser analizada con éxito, obteniendo para cada uno abstracciones de tipo EPA y States.

Entre los contratos del *benchmark 1*, hubo dos casos (*Asset Transfer* y *Digital Locker* y sus versiones *"fixed*) en los que no se pudo obtener la abstracción de tipo EPA. Por otro lado, el contrato *ValidatorAuction* del *benchmark 2* tampoco pudo ser ejecutado con éxito.

En estos casos, la ejecución de la herramienta alcanzó el timeout antes de completar el análisis, lo que impidió la generación de la abstracción correspondiente. Estos casos serán analizados por separado.

En las tablas, las columnas *Dif. Estados* y *Dif. Transiciones* indican la diferencia entre la cantidad de estados y transiciones encontradas por Echidna y la cantidad de estados y transiciones en la abstracción generada por VeriSol con *txBound* igual a 8, que usaremos como referencia. Los valores negativos indican que Echidna encontró menos estados o transiciones que la abstracción de referencia, mientras que los valores positivos indican que encontró más. Estos valores serán explorados en profundidad en secciones posteriores.

En los casos en que la herramienta no pudo completar la ejecución, se indicará con un guion (-) en lugar de un valor numérico.

Cabe recordar que la optimización 3.5.3 es aplicable únicamente cuando se pretende generar una EPA, por lo que habrá el doble de resultados disponibles para ese tipo de abstracción.

Resultados Benchmark 1

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	0	0	3.62
BasicProvenanceFixed	0	0	4.11
DefectiveComponentCounter	0	0	3.94
DefectiveComponentCounterFixed	0	0	3.87
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	-1	4.69
HelloBlockchain	0	0	3.72
HelloBlockchainFixed	0	0	3.96
RefrigeratedTransportation	-1	-3	6.02
RefrigeratedTransportationFixed	-1	-5	6.19
RoomThermostat	0	-2	4.41
SimpleMarketplace	0	-3	4.59
SimpleMarketplaceFixed	0	0	4.31

Tab. 4.3: Benchmark 1 - EPA - Test Limit 1.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	0	0	17.92
BasicProvenanceFixed	0	0	22.02
DefectiveComponentCounter	0	1	30.0
DefectiveComponentCounterFixed	1	2	23.85
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	0	53.64
HelloBlockchain	0	0	24.59
HelloBlockchainFixed	0	0	22.29
RefrigeratedTransportation	0	-1	54.76
RefrigeratedTransportationFixed	0	0	54.15
RoomThermostat	0	0	24.76
SimpleMarketplace	0	0	28.61
SimpleMarketplaceFixed	0	0	24.52

Tab. 4.4: Benchmark 1 - EPA - Test Limit 50.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	0	0	146.37
BasicProvenanceFixed	0	0	183.22
DefectiveComponentCounter	0	1	260.11
DefectiveComponentCounterFixed	1	2	208.11
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	0	475.09
HelloBlockchain	0	0	209.7
HelloBlockchainFixed	0	0	188.82
RefrigeratedTransportation	0	0	502.7
RefrigeratedTransportationFixed	0	0	516.23
RoomThermostat	0	0	210.51
SimpleMarketplace	0	0	247.76
SimpleMarketplaceFixed	0	0	200.22

Tab. 4.5: Benchmark 1 - EPA - Test Limit 500.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	-1	-1	6.91
BasicProvenanceFixed	-1	-1	6.65
DefectiveComponentCounter	0	0	5.69
DefectiveComponentCounterFixed	0	0	9.08
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	-1	6.47
HelloBlockchain	0	0	5.44
HelloBlockchainFixed	0	0	6.44
RefrigeratedTransportation	-1	-3	7.66
RefrigeratedTransportationFixed	-1	-6	8.99
RoomThermostat	-1	-3	6.52
SimpleMarketplace	0	-2	6.91
SimpleMarketplaceFixed	0	0	7.50

Tab. 4.6: Benchmark 1 - EPA - Test Limit 1.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	0	0	21.96
BasicProvenanceFixed	0	0	22.30
DefectiveComponentCounter	0	1	32.58
DefectiveComponentCounterFixed	0	0	33.14
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	0	59.15
HelloBlockchain	0	0	25.84
HelloBlockchainFixed	0	0	20.40
RefrigeratedTransportation	0	0	40.21
RefrigeratedTransportationFixed	0	0	47.76
RoomThermostat	0	0	21.04
SimpleMarketplace	0	0	26.70
SimpleMarketplaceFixed	0	0	21.70

Tab. 4.7: Benchmark 1 - EPA - Test Limit 50.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-	-	-
AssetTransferFixed	-	-	-
BasicProvenance	0	0	149.17
BasicProvenanceFixed	0	0	161.49
DefectiveComponentCounter	0	1	266.48
DefectiveComponentCounterFixed	0	0	264.89
DigitalLocker	-	-	-
DigitalLockerFixed	-	-	-
FrequentFlyerRewardsCalculator	0	0	554.87
HelloBlockchain	0	0	215.94
HelloBlockchainFixed	0	0	143.32
RefrigeratedTransportation	0	0	336.34
RefrigeratedTransportationFixed	0	0	407.69
RoomThermostat	0	0	151.61
SimpleMarketplace	0	0	205.45
SimpleMarketplaceFixed	0	0	155.90

Tab. 4.8: Benchmark 1 - EPA - Test Limit 500.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-7	-32	52.69
AssetTransferFixed	-9	-30	47.01
BasicProvenance	0	0	3.79
BasicProvenanceFixed	-1	-1	3.69
DefectiveComponentCounter	1	1	3.84
DefectiveComponentCounterFixed	0	0	3.77
DigitalLocker	0	-38	31.44
DigitalLockerFixed	-2	-14	25.39
FrequentFlyerRewardsCalculator	0	0	5.25
HelloBlockchain	0	0	3.89
HelloBlockchainFixed	0	0	3.51
RefrigeratedTransportation	0	-5	5.18
RefrigeratedTransportationFixed	-1	-4	4.90
RoomThermostat	-1	-3	3.65
SimpleMarketplace	0	-1	4.04
SimpleMarketplaceFixed	0	0	3.84

Tab. 4.9: Benchmark 1 - STATES - Test Limit 1.000

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
AssetTransfer	-6	-22	532.97
AssetTransferFixed	-6	-22	509.38
BasicProvenance	0	0	20.46
BasicProvenanceFixed	0	0	18.19
DefectiveComponentCounter	1	2	26.36
DefectiveComponentCounterFixed	1	2	23.37
DigitalLocker	0	-2	406.08
DigitalLockerFixed	0	-1	221.09
FrequentFlyerRewardsCalculator	0	0	64.85
HelloBlockchain	0	0	27.47
HelloBlockchainFixed	0	0	17.77
RefrigeratedTransportation	0	-1	43.64
RefrigeratedTransportationFixed	0	-2	37.43
RoomThermostat	0	0	18.33
SimpleMarketplace	0	0	24.22
SimpleMarketplaceFixed	0	0	20.36

Tab. 4.10: Benchmark 1 - STATES - Test Limit 50.000

Nombre del contrato	**Dif. Estados**	**Dif. Transiciones**	**Tiempo (s)**
AssetTransfer	-3	-18	4893.47
AssetTransferFixed	-3	-14	4742.92
BasicProvenance	0	0	169.55
BasicProvenanceFixed	0	0	148.81
DefectiveComponentCounter	1	3	217.15
DefectiveComponentCounterFixed	1	2	199.13
DigitalLocker	0	0	3820.48
DigitalLockerFixed	0	0	1986.26
FrequentFlyerRewardsCalculator	0	0	637.50
HelloBlockchain	0	0	241.99
HelloBlockchainFixed	0	0	143.41
RefrigeratedTransportation	0	0	388.29
RefrigeratedTransportationFixed	0	0	332.81
RoomThermostat	0	0	154.18
SimpleMarketplace	0	0	206.27
SimpleMarketplaceFixed	0	0	171.00

Tab. 4.11: Benchmark 1 - STATES - Test Limit 500.000

Resultados Benchmark 2

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	-2	-13	17.45
Crowdfunding	-1	-11	26.14
EPXCrowdsale	-3	-16	91.62
EscrowVault	-2	-11	168.02
RefundEscrow	-2	-11	29.51
RockPaperScissors	-3	-5	9.50
SimpleAuction	-4	-23	14.76
ValidatorAuction	-	-	-

Tab. 4.12: Benchmark 2 - EPA - Test Limit 1.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	-1	-5	57.73
Crowdfunding	-1	-7	59.55
EPXCrowdsale	-1	-11	443.36
EscrowVault	0	-2	1300.67
RefundEscrow	-1	-5	209.31
RockPaperScissors	-1	-3	44.37
SimpleAuction	-2	-13	56.34
ValidatorAuction	-	-	-

Tab. 4.13: Benchmark 2 - EPA - Test Limit 50.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	0	-1	373.13
Crowdfunding	0	0	387.32
EPXCrowdsale	-1	-9	3814.16
EscrowVault	0	0	11851.10
RefundEscrow	0	-2	1783.90
RockPaperScissors	0	0	362.27
SimpleAuction	0	-1	436.05
ValidatorAuction	-	-	-

Tab. 4.14: Benchmark 2 - EPA - Test Limit 500.000 - Sin Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	-2	-12	15.17
Crowdfunding	-1	-10	15.55
EPXCrowdsale	-1	-11	59.63
EscrowVault	-1	-8	21.72
RefundEscrow	0	-7	13.25
RockPaperScissors	-2	-4	10.38
SimpleAuction	-4	-21	19.07
ValidatorAuction	-	-	-

Tab. 4.15: Benchmark 2 - EPA - Test Limit 1.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	-1	-5	30.08
Crowdfunding	0	-1	25.66
EPXCrowdsale	-1	-9	96.87
EscrowVault	0	-1	84.66
RefundEscrow	0	-2	53.17
RockPaperScissors	0	0	28.45
SimpleAuction	0	-3	48.92
ValidatorAuction	-	-	-

Tab. 4.16: Benchmark 2 - EPA - Test Limit 50.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	0	0	153.07
Crowdfunding	0	0	131.93
EPXCrowdsale	-1	-8	403.60
EscrowVault	0	0	664.75
RefundEscrow	0	0	426.01
RockPaperScissors	0	0	185.15
SimpleAuction	0	-1	335.73
ValidatorAuction	-	-	-

Tab. 4.17: Benchmark 2 - EPA - Test Limit 500.000 - Con Optimización

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	-3	-13	36.77
Crowdfunding	-2	-17	12.35
EPXCrowdsale	-6	-35	13.61
EscrowVault	-2	-9	7.88
RefundEscrow	0	-3	5.41
RockPaperScissors	-5	-11	8.36
SimpleAuction	-3	-16	13.09
ValidatorAuction	-	-	-

Tab. 4.18: Benchmark 2 - STATES - Test Limit 1.000

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	0	-4	139.06
Crowdfunding	-1	-9	35.46
EPXCrowdsale	-3	-26	96.82
EscrowVault	0	0	60.21
RefundEscrow	0	0	34.06
RockPaperScissors	-3	-9	43.95
SimpleAuction	-2	-10	54.24
ValidatorAuction	-	-	-

Tab. 4.19: Benchmark 2 - STATES - Test Limit 50.000

Nombre del contrato	Dif. Estados	Dif. Transiciones	Tiempo (s)
Auction	0	0	1024.71
Crowdfunding	0	-4	250.26
EPXCrowdsale	-3	-22	913.58
EscrowVault	0	0	507.71
RefundEscrow	0	0	284.69
RockPaperScissors	-3	-9	356.21
SimpleAuction	0	0	418.1
ValidatorAuction	-	-	-

Tab. 4.20: Benchmark 2 - STATES - Test Limit 500.000

En síntesis, al ver los resultados de las ejecuciones, se puede afirmar que es posible generar abstracciones por predicados de forma dinámica a partir de contratos inteligentes, utilizando el *fuzzer* Echidna.

4.3.2. Utilidad y aplicación de las abstracciones generadas

Luego de ver que es posible generar las abstracciones de la manera planteada, se procede a analizar si estas abstracciones son útiles para comprender el comportamiento de los contratos y para identificar vulnerabilidades.

Esto se hará analizando en detalle la abstracción de tipo states obtenida para el contrato *Digital Locker*.

DigitalLocker y DigitalLockerFixed (Benchmark 1, States)

El contrato inteligente *Digital Locker* permite gestionar el acceso a documentos digitales de forma controlada, asegurando que el propietario mantenga el control sobre quién puede acceder y durante cuánto tiempo.

El flujo del contrato comienza cuando el propietario solicita al banco, que es quien mantiene el recurso, que habilite el documento para que sea compartido, lo que inicia el estado Requested. El agente del banco llama a la función `BeginReviewProcess()` para revisar la solicitud, cambiando el estado a *DocumentReview*. Una vez aprobada (por él mismo), el agente ejecuta la función `UploadDocument()`, lo que deja el documento en el estado AvailableToShare.

A partir de ahí, el propietario puede decidir si comparte el documento con un tercero específico o espera solicitudes de acceso de terceros aleatorios. Si un tercero aleatorio solicita acceso, se llama a la función `RequestLockerAccess()` y el estado cambia a *SharingRequestPending*. El propietario puede aceptar esta solicitud con `GrantAccess()` o rechazarla, en cuyo caso el documento vuelve al estado *AvailableToShare*.

Si el acceso es concedido, el estado cambia a *SharingWithThirdParty*. Tanto el propietario como el tercero con acceso pueden liberar el documento cuando ya no lo necesiten mediante la función `ReleaseDocumentAccess()`, que retorna el estado a *AvailableToShare*. Nótese que el documento no puede ser accedido por más de un participante a la vez.

Por último, el agente del banco tiene la opción de finalizar el proceso de compartición en cualquier momento usando la función `TerminateSharing()`, cambiando el estado a *Terminated* y bloqueando definitivamente el acceso al documento.

Este contrato fue sólo analizado generando su abstracción de tipo states. El diagrama esperado para *DigitalLocker*, diseñado por sus desarrolladores, se puede ver en la figura 4.1.

Al correrlo inicialmente, con un *test limit* de 500.000, se obtuvo un modelo con muchas más transiciones de las esperadas, lo que sugiere algún error en el código. Esa abstracción se halla en la figura 4.2.

Efectivamente, en ninguno de los métodos del contrato se estaba controlando en qué estado se encontraba el contrato antes de ejecutar una función, lo que permitía que se pudieran ejecutar funciones en estados no deseados, como por ejemplo llamar a la función `AcceptSharingRequest()` en el estado *SharingWithThirdParty*, lo cual significaría que el propietario acepta una solicitud de acceso de un tercero cuando ya está compartiendo el recurso con otro participante. Lo que se ve es que desde cualquier estado es posible llamar a cualquier función.

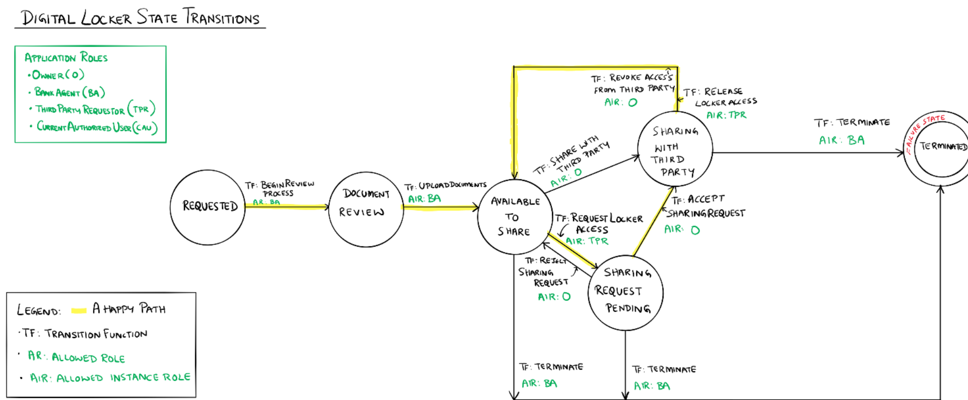

Fig. 4.1: DigitalLocker - Comportamiento esperado

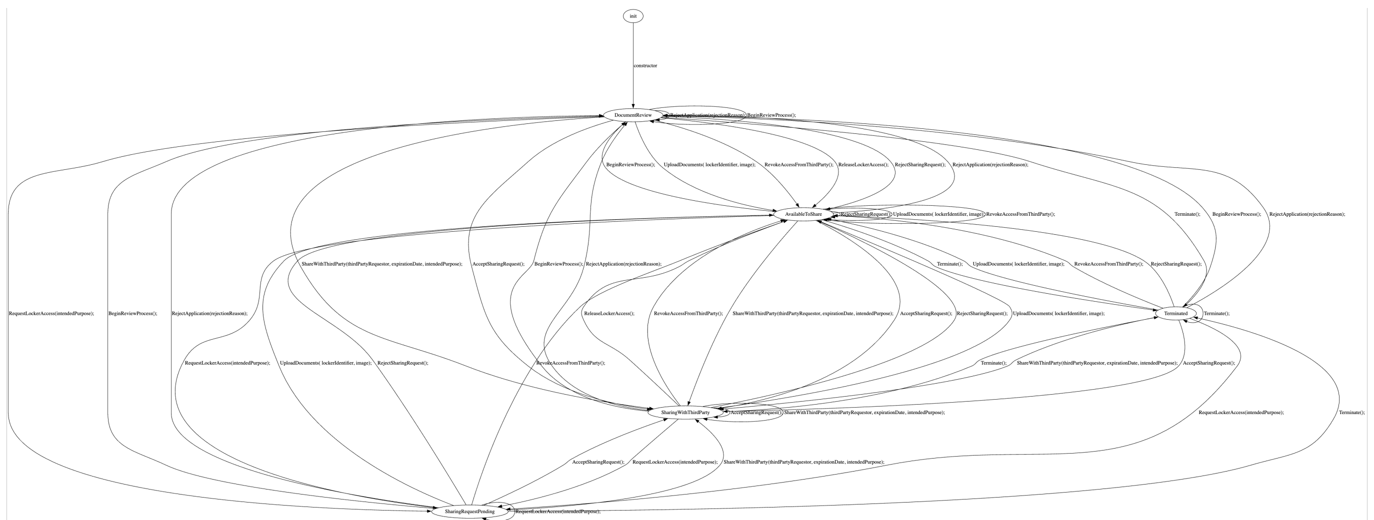

Fig. 4.2: DigitalLocker - States - 500.000

Estos defectos en los contratos inteligentes, entre otros, fueron encontrados en trabajos previos ([18], [29]), donde se generaron nuevos contratos con los errores corregidos.

En este caso, para corregir este error se agregaron las cláusulas require correspondientes en cada función y se volvió a ejecutar la herramienta. Por ejemplo, dentro de la función `AcceptSharingRequest()` se agregó la siguiente precondición:

```
require(State == StateType.SharingRequestPending);.
```

Al correr el contrato corregido (*DigitalLockerFixed*) con un *test limit* de 50.000, hubo una transición del modelo original, provisto por los desarrolladores del Azure Blockchain Workbench, que no pudo ser encontrada. Luego, al correrlo con 500.000, sí se logró explorar esa situación. Esta transición permite que el contrato llegue desde el estado *SharingWithThirdParty* al estado *AvailableToShare* tras ejecutar la función `RequestLockerAccess()`. Esto se puede encontrar en las figuras 4.3 y 4.4.

Además, se puede notar que las abstracciones devueltas por nuestra herramienta revelan un comportamiento que no había sido tenido en cuenta por los desarrolladores al crear su diagrama, que involucra volver al estado *DocumentReview* cuando el agente de banco

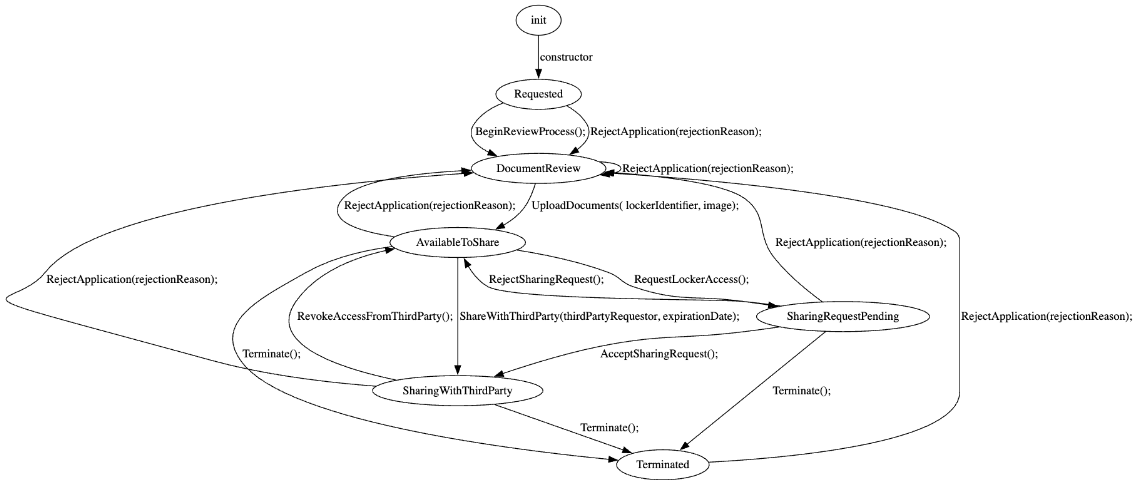

Fig. 4.3: DigitalLocker (Fixed) - States - 50.000

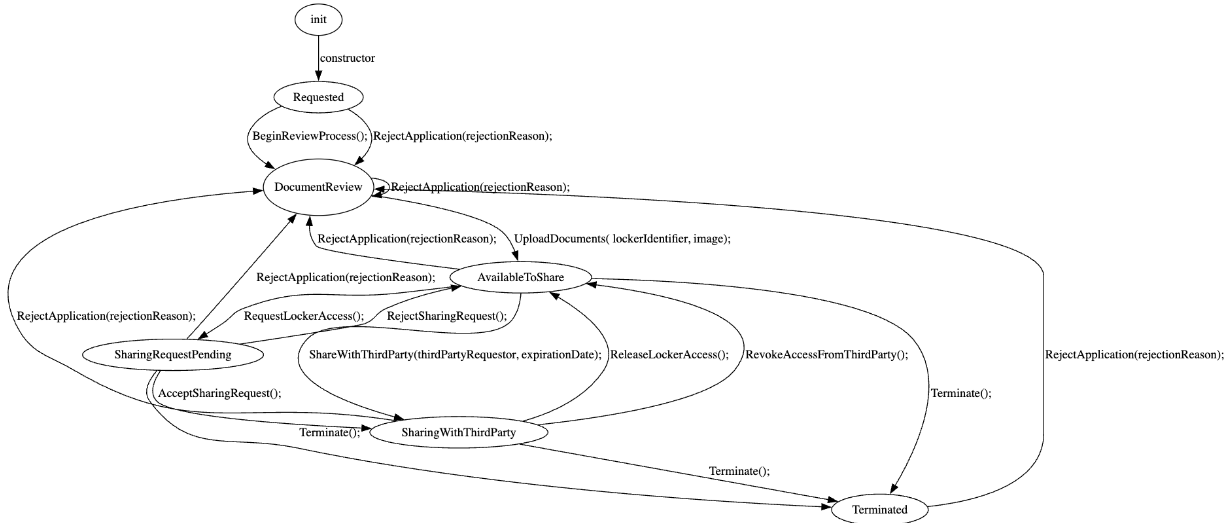

Fig. 4.4: DigitalLocker (Fixed) - States - 500.000

llama a la función `RejectApplication()` desde cualquiera del resto de los estados. Esta función ni siquiera aparece en el diagrama original.

Tras analizar detalladamente las abstracciones resultantes, particularmente las de tipo states de los contratos del *Benchmark* 1 (de cuyos modelos abstractos esperados disponemos), se puede confirmar que estas permitieron identificar todas las vulnerabilidades esperadas en los contratos, con excepción de *AssetTransfer*. Dichas vulnerabilidades fueron las que motivaron la corrección de los contratos y sus respectivas versiones *"fixed"* en trabajos previos. La ejecución de estas versiones *"fixed"*, a excepción de *AssetTransferFixed*, produjo en todos los casos una abstracción idéntica al modelo conceptual provisto por los desarrolladores. Además, como se mencionó anteriormente, la ejecución de *DigitalLockerFixed* reveló un comportamiento no contemplado en el modelo original.

Esto demuestra la utilidad de las abstracciones generadas por la herramienta, así como su potencial aplicación para comparar el comportamiento real del contrato con el deseado y detectar vulnerabilidades y comportamientos inesperados.

4.3.3. Comparación de cobertura con VeriSol

En esta sección se comparan algunos resultados obtenidos con Echidna con los obtenidos por VeriSol. Se analizan las abstracciones resultantes de algunos casos exitosos (contratos *RefrigeratedTransportation* y *RockPaperScissors*) y otros no exitosos (*EPX-Crowdsale*), siendo los casos exitosos aquellos en los que se encontraron todos los estados y transiciones encontradas por VeriSol.

A modo de resumen y previo a un análisis detenido sobre ciertos casos particulares, a continuación se presentan tablas que muestran qué contratos lograron alcanzar la cobertura deseada (es decir, la obtenida por VeriSol) para cada tipo de abstracción, usando la mejor ejecución de Echidna en cada caso. En las tablas, los casos exitosos se marcan con un símbolo de verificación (✓), mientras que los casos no exitosos se indican con una cruz (✗).

La información de estas tablas puede ser contrastada con la información presentada en la sección 4.3.1, donde se muestran los resultados de cobertura de todas las ejecuciones realizadas.

Nombre del contrato	EPA	States
Asset Transfer	✗	✗
Asset Transfer Fixed	✗	✗
Basic Provenance	✓	✓
Basic Provenance Fixed	✓	✓
Defective Component Counter	✓	✓
Defective Component Counter Fixed	✓	✓
Digital Locker	✗	✓
Digital Locker Fixed	✗	✓
Frequent Flyer Rewards Calculator	✓	✓
Hello Blockchain	✓	✓
Hello Blockchain Fixed	✓	✓
Refrigerated Transportation	✓	✓
Refrigerated Transportation Fixed	✓	✓
Room Thermostat	✓	✓
Simple Marketplace	✓	✓
Simple Marketplace Fixed	✓	✓

Tab. 4.21: Cobertura Contratos Benchmark 1

Nombre del contrato	EPA	States
Auction	✓	✓
Crowdfunding	✓	✗
EPXCrowdsale	✗	✗
EscrowVault	✓	✓
RefundEscrow	✓	✓
RockPaperScissors	✓	✓
SimpleAuction	✗	✓
ValidatorAuction	-	-

Tab. 4.22: Cobertura Contratos Benchmark 2

Cabe mencionar que el contrato *ValidatorAuction* no tiene clasificación ya que no pudo ejecutarse ni con Echidna ni con VeriSol dentro del límite de tiempo fijado.

RefrigeratedTransportation (Benchmark 1, EPA y States)

Para comenzar con un análisis más detallado se estudia el caso del contrato *RefrigeratedTransportation*. El mismo tiene la función de monitorear el estado de una cadena de suministro de algún producto en la que deben asegurarse ciertas reglas de conformidad relacionadas con el nivel de humedad y la temperatura del producto. Estas reglas son definidas por la contraparte que crea el contrato al inicializarlo. Si alguna de ellas no se cumple, el contrato indicará que no se cumplieron las condiciones (`OutOfCompliance`).

Cualquier usuario puede consultar el estado del contrato y a su vez, cada contraparte puede transferir la responsabilidad a una contraparte siguiente, siguiendo así la cadena de suministro.

Al correr la herramienta para este contrato, solamente usando Echidna como herramienta de fondo y con el fin de generar una EPA, se dio la situación en la que dándole el valor de 50.000 al parámetro *test limit*, una de las transiciones no fue encontrada, pero corriéndola con un valor de *test limit* igual a 500.000 la herramienta logró explorar esa transición. Esto se puede ver en la figura 4.5.

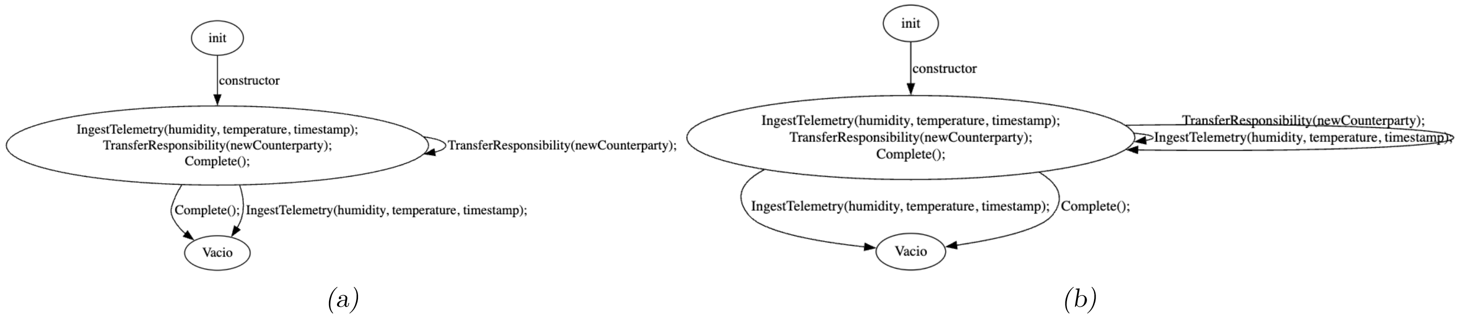

Fig. 4.5: RefrigeratedTransportation - EPA - 50.000 vs. 500.000

La misma situación se dio al generar la abstracción de tipo states (figura 4.6).

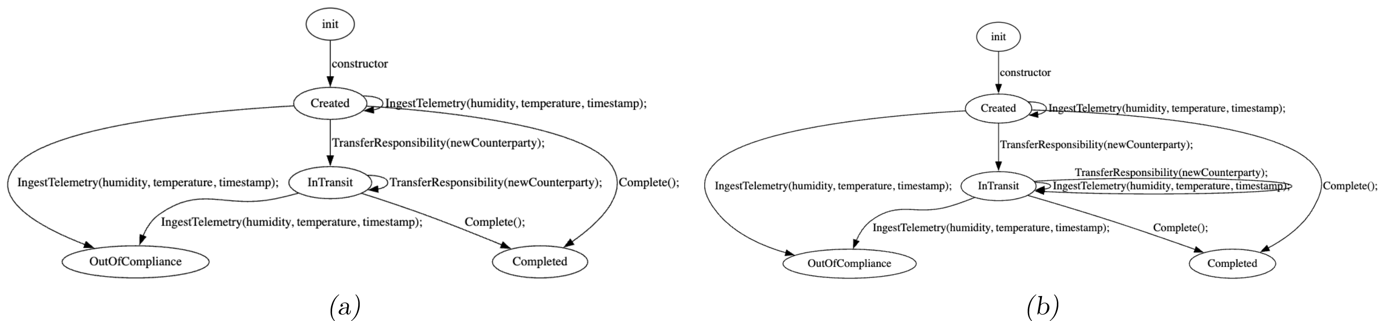

Fig. 4.6: RefrigeratedTransportation - States - 50.000 vs. 500.000

Esto deja en claro que, en muchas situaciones, la incapacidad de la herramienta para

identificar ciertos estados o transiciones se debe, simplemente, a una elección excesivamente restrictiva del *test limit*. Es posible que, con una mayor capacidad de procesamiento o un tiempo de ejecución más prolongado, todas las transiciones podrían ser identificadas.

Adicionalmente, las abstracciones resultantes nos permiten darnos cuenta de un error en el código del contrato con respecto a la especificación publicada por los desarrolladores. Nuestra herramienta encuentra una transición del estado `Created` al estado `Completed`, ejecutando la función `Complete()`, situación que los desarrolladores ignoraron. Este *bug* puede ser corregido fácilmente agregando una cláusula require que indique que la función `Complete()` sólo pueda ser llamada cuando el contrato se encuentra en el estado `InTransit`. Al agregar esa línea de código y volver a ejecutar la herramienta, la abstracción resultante es exactamente igual a la planteada en el *benchmark* por sus desarrolladores.

RockPaperScissors (Benchmark 2, States)

Continuando, se va a analizar una situación en la que un pequeño cambio en el código fuente del contrato puede ayudar a que la herramienta explore de manera mucho más eficiente los posibles estados del mismo.

El contrato inteligente en cuestión es RockPaperScissors, que implementa el clásico juego de piedra, papel o tijeras entre dos jugadores. Se puede pensar que el contrato tiene 3 fases: una de inicialización, otra en la que cada jugador indica cuál va a ser su elección (0: piedra, 1: papel ó 2: tijeras), y una última fase en la que se determina quién es el ganador.

Llamativamente, al ejecutar la herramienta con un *test limit* de 500.000, la abstracción generada por la herramienta encuentra únicamente 3 estados (además del inicial): uno en el que ninguno de los jugadores ha realizado su elección, otro en el que sólo el jugador 1 ha registrado su elección, y un tercero en el que únicamente el jugador 2 ha hecho lo propio. Es decir, no se llega al final del juego en ninguna situación. Este hallazgo puede visualizarse en la figura 4.7.

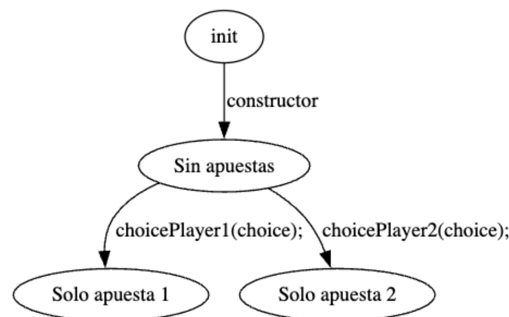

Fig. 4.7: RockPaperScissors - States - 500.000

Para resolver este problema y lograr que todos los estados fueran explorados, se analizó detalladamente el contenido del contrato, prestando especial atención al modo de operación de Echidna. Posteriormente, se volvió a ejecutar la herramienta tras modificar la forma en que se almacena el argumento (es decir, la elección del participante) en las funciones `choicePlayer1()` y `choicePlayer2()`.

Se eliminó la cláusula `require(choice >= 0 && choice <= 2);` y se modificó la lógica de almacenamiento de la elección de cada jugador, implementándola de la siguiente ma-

nera: `p1Choice = int(choice % 3);`. Con esta modificación, la función puede ejecutarse independientemente del valor que Echidna asigne al argumento, a diferencia de la implementación original, que requería que el valor de choice estuviera entre 0 y 2. Esto hacía que durante gran parte de la campaña de *fuzzing* fuera imposible ejecutar esas funciones, ya que el valor del parámetro `choice` elegido por Echidna de entre todos los números enteros de 256 bits, debía estar el rango mencionado.

Tras realizar esta nueva ejecución, se identificaron todos los estados y transiciones esperados en los resultados (figura 4.8).

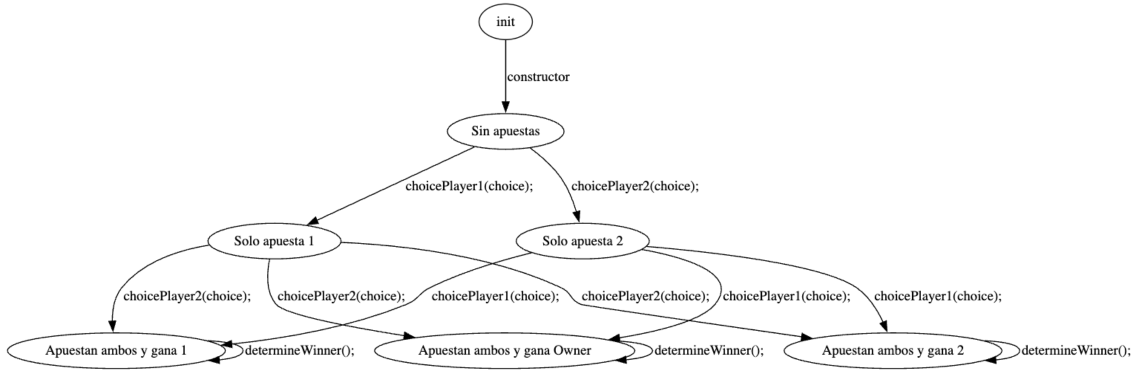

Fig. 4.8: RockPaperScissors (con optimización) - States - 500.000

Además de encontrar todos los estados y transiciones posibles, el tiempo de ejecución no presentó variaciones significativas.

EPXCrowdsale (Benchmark 2, EPA)

Si bien en la mayoría de los casos las abstracciones generadas lograron capturar todos los estados y transiciones esperadas, en otras ocasiones la herramienta no pudo alcanzar este objetivo. Más aún, tres de los contratos evaluados, *AssetTransfer*, *DigitalLocker* y *ValidatorAuction*, debieron ser descartados, ya que no lograron generar una EPA en un tiempo razonable. Ninguno de ellos finalizó su ejecución en menos de dos horas con un *test limit* de 1.000.

Este problema se debe a la gran cantidad de *queries* generadas, que a su vez surge del número de estados abstractos a explorar. En el caso de *AssetTransfer*, se añadieron aproximadamente 92.112 *queries*, mientras que en *DigitalLocker* (en su versión corregida) la cantidad ascendió a 22.518, lo que impactó significativamente en el tiempo de ejecución. Por su parte, el contrato *ValidatorAuction*, que también fue descartado en el trabajo [34], generó 12.228 *queries*. En calidad de referencia, el siguiente contrato con más *queries* es *EscrowVault* con 1.152, una cantidad en un orden de magnitud menor, y el 64.58 % de los contratos analizados generaron 100 *queries* o menos. Esto se puede ver en detalle en [3].

Paralelamente, hubo un caso en el que la ejecución de la herramienta finalizó de manera correcta, pero, incluso con el mayor *test limit* evaluado, no logró encontrar los mismos estados y transiciones que VeriSol. Este fue el caso del contrato *EPXCrowdsale* en modo EPA, donde se detectó un estado abstracto inalcanzable y ocho transiciones que no fueron descubiertas, tal como se puede ver en las figuras 4.9 y 4.10.

Es altamente probable que esta diferencia se deba a la granularidad de las precondiciones de cada función, que imponen múltiples restricciones sobre diferentes variables

del contrato antes de permitir su ejecución. Por ejemplo, la función `refund()` solo puede ejecutarse si la variable `isCrowdsaleClosed` está configurada en `true`, entre otras condiciones. Sin embargo, esta variable solo se asigna dentro del cuerpo de la función `checkGoalReached()`, cuyo código se muestra en el listing 4.1.

Es posible que la herramienta nunca haya logrado explorar la combinación exacta de condiciones necesarias para ejecutar `checkGoalReached()` en una rama que establezca `isCrowdsaleClosed = true`, impidiendo así la activación de `refund()` y limitando la exploración del espacio de estados.

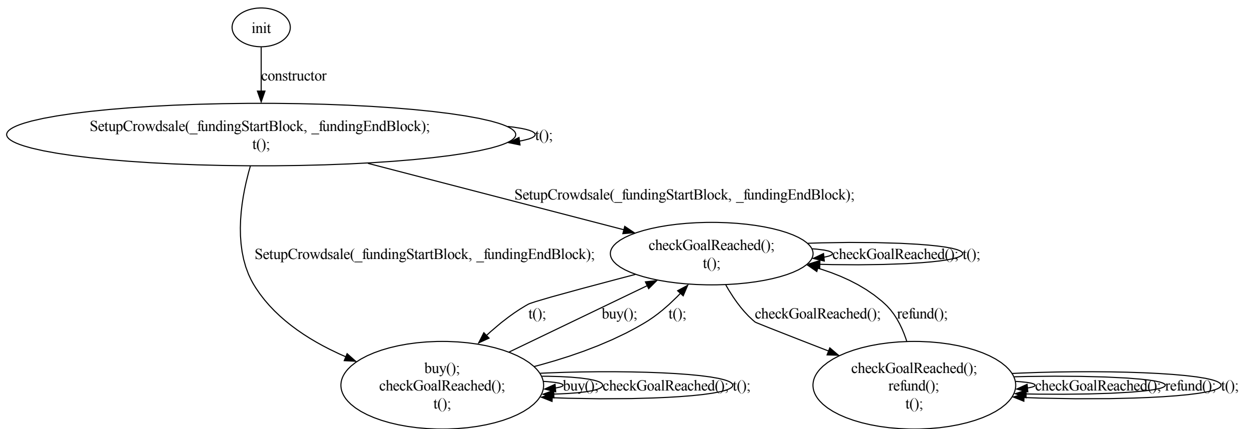

Fig. 4.9: EPXCrowdsale - EPA - VeriSol - Tx Bound 8

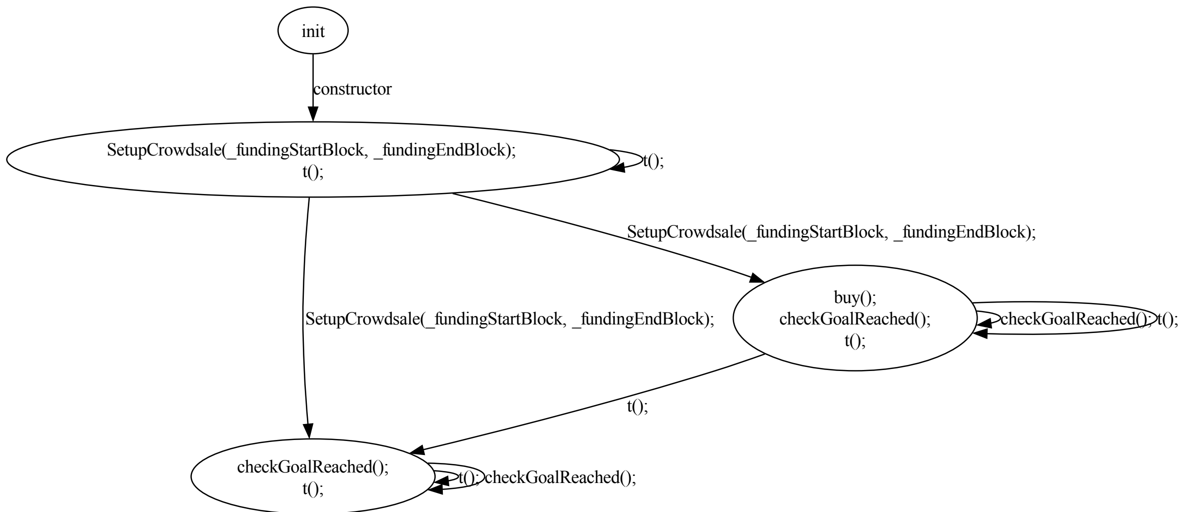

Fig. 4.10: EPXCrowdsale - EPA - Echidna - Test Limit 500.000

```
// Codigo de la funcion checkGoalReached()
function checkGoalReached() public onlyOwner {
    require (isCrowdSaleSetup);

    if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber <=
        fundingEndBlock && blockNumber >= fundingStartBlock)) { // ICO in
        progress, under softcap
        areFundsReleasedToBeneficiary = false;
        isCrowdSaleClosed = false;
        CurrentStatus = "In progress (Eth < Softcap)";
    } else if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber <
        fundingStartBlock)) { // ICO has not started
        areFundsReleasedToBeneficiary = false;
        isCrowdSaleClosed = false;
        CurrentStatus = "Crowdsale is setup";
    } else if ((amountRaisedInWei < fundingMinCapInWei) && (blockNumber >
        fundingEndBlock)) { // ICO ended, under softcap
        areFundsReleasedToBeneficiary = false;
        isCrowdSaleClosed = true;
        CurrentStatus = "Unsuccessful (Eth < Softcap)";
    } else if ((amountRaisedInWei >= fundingMinCapInWei) && (tokensRemaining ==
        0)) { // ICO ended, all tokens bought!
        areFundsReleasedToBeneficiary = true;
        isCrowdSaleClosed = true;
        CurrentStatus = "Successful (EPX >= Hardcap)!";
    } else if ((amountRaisedInWei >= fundingMinCapInWei) && (blockNumber >
        fundingEndBlock) && (tokensRemaining > 0)) { // ICO ended, over softcap!
        areFundsReleasedToBeneficiary = true;
        isCrowdSaleClosed = true;
        CurrentStatus = "Successful (Eth >= Softcap)!";
    } else if ((amountRaisedInWei >= fundingMinCapInWei) && (tokensRemaining >
        0) && (blockNumber <= fundingEndBlock)) { // ICO in progress, over
        softcap!
        areFundsReleasedToBeneficiary = true;
        isCrowdSaleClosed = false;
        CurrentStatus = "In progress (Eth >= Softcap)!";
    }
}
```

Listing 4.1: Función checkGoalReached() dentro del contrato EPXCrowdsale

En síntesis, para los contratos del *Benchmark* 1, en modo EPA, un *test limit* de 50.000 fue suficiente para identificar todos los estados y transiciones esperadas, especialmente cuando se utilizó la optimización 3.5.3. En el modo *states*, se obtuvo un comportamiento similar: con un *test limit* de 500.000, la herramienta logró cubrir todos los casos para todos los contratos excepto *AssetTransfer*, lo que sugiere que, incluso con una mayor cantidad de pruebas, este contrato seguiría siendo problemático debido a su complejidad estructural. El único caso particular en el que la ejecución con Echidna encontró más estados o transiciones que la abstracción de referencia fue para el contrato *DefectiveComponentCounter* (y *DefectiveComponentCounterFixed*), tanto en modo EPA como en modo *states*. El motivo

de este comportamiento es el *txBound* utilizado y su explicación se encuentra a modo de ejemplo sobre el final de la sección 3.5.3. En resumen, al ejecutarlo con *txBound* igual a 8, VeriSol nunca llega a explorar algunos puntos del programa.

Por otro lado, para los contratos del *Benchmark* 2, en modo EPA, con un *test limit* de 500.000 se identificaron correctamente los estados y transiciones en la mayoría de los casos, con excepción de *EPXCrowdsale*, y *SimpleAuction*. En el caso de *SimpleAuction*, se identificó la ausencia de una única transición, que sí apareció en el resultado de otras ejecuciones, por lo que es atribuible al no determinismo de Echidna. Sin la optimización, también se encontraron problemas en *Auction* y *RefundEscrow*, lo que sugiere que la estrategia de reducción del espacio de búsqueda mejora la cobertura de estados. En modo *states*, *Crowdfunding* presentó cuatro transiciones faltantes, mientras que *EPXCrowdsale* volvió a mostrar una cobertura deficiente y *RockPaperScissors* presentó dificultades relacionadas con la validación del módulo utilizado en sus condiciones, cuya solución se abordó en la sección 4.3.3.

4.3.4. Comparación de tiempos con VeriSol

Otra métrica relevante para comparar la utilización de Echidna y VeriSol es el tiempo de ejecución.

A continuación se presentan tablas con los resultados. Todas las ejecuciones realizadas utilizando a VeriSol como herramienta de fondo, lo hicieron con un *transaction bound* de 8 y un *timeout* de 600 segundos. A su vez, en todas las ejecuciones de VeriSol se habilitaron todas las optimizaciones disponibles, por lo que la comparación temporal se realiza con las ejecuciones de Echidna que también lo hicieron. Junto a los tiempos de ejecución de Echidna se presenta la diferencia porcentual con respecto a la ejecución de VeriSol.

TL indica el *test limit* correspondiente a la ejecución de Echidna y TXB el *transaction bound* usado por VeriSol.

Nombre del contrato	TL=1k	TL=50k	TL=500k	TXB=8
AssetTransfer	-	-	-	471.97
AssetTransferFixed	-	-	-	1566.53
BasicProvenance	6.91 (-44 %)	21.96 (+77 %)	149.17 (+1099 %)	12.44
BasicProvenanceFixed	6.65 (-69 %)	22.3 (+5 %)	161.49 (+663 %)	21.17
DefectiveComponentCounter	5.69 (-22 %)	32.58 (+349 %)	266.48 (+3571 %)	7.26
DefectiveComponentCounterFixed	9.08 (+5 %)	33.14 (+283 %)	264.89 (+2962 %)	8.65
DigitalLocker	-	-	-	28.98
DigitalLockerFixed	-	-	-	243.26
FrequentFlyerRewardsCalculator	6.47 (-26 %)	59.15 (+576 %)	554.87 (+6241 %)	8.75
HelloBlockchain	5.44 (-24 %)	25.84 (+260 %)	215.94 (+2908 %)	7.18
HelloBlockchainFixed	6.44 (-36 %)	20.4 (+104 %)	143.32 (+1335 %)	9.99
RefrigeratedTransportation	7.66 (-60 %)	40.21 (+109 %)	336.34 (+1645 %)	19.28
RefrigeratedTransportationFixed	8.99 (-69 %)	47.76 (+67 %)	407.69 (+1327 %)	28.56
RoomThermostat	6.52 (-52 %)	21.04 (+55 %)	151.61 (+1018 %)	13.56
SimpleMarketplace	6.91 (-66 %)	26.7 (+31 %)	205.45 (+909 %)	20.36
SimpleMarketplaceFixed	7.5 (-61 %)	21.7 (+13 %)	155.9 (+715 %)	19.13

Tab. 4.23: Tiempo (s) - Benchmark 1 - EPA - Con Optimización

Nombre del contrato	TL=1k	TL=50k	TL=500k	TXB=8
AssetTransfer	52.69 (-96 %)	532.97 (-64 %)	4893.47 (+226 %)	1500.64
AssetTransferFixed	47.01 (-97 %)	509.38 (-66 %)	4742.92 (+219 %)	1484.59
BasicProvenance	3.79 (-79 %)	20.46 (11 %)	169.55 (+823 %)	18.36
BasicProvenanceFixed	3.69 (-84 %)	18.19 (-20 %)	148.81 (+557 %)	22.65
DefectiveComponentCounter	3.84 (-73 %)	26.36 (+87 %)	217.15 (+1441 %)	14.09
DefectiveComponentCounterFixed	3.77 (-76 %)	23.37 (+47 %)	199.13 (+1152 %)	15.90
DigitalLocker	31.44 (-93 %)	406.08 (-7 %)	3820.48 (+777 %)	435.44
DigitalLockerFixed	25.39 (-93 %)	221.09 (-38 %)	1986.26 (+454 %)	358.51
FrequentFlyerRewardsCalculator	5.25 (-56 %)	64.85 (+439 %)	637.50 (+5203 %)	12.02
HelloBlockchain	3.89 (-63 %)	27.47 (+162 %)	241.99 (+2207 %)	10.49
HelloBlockchainFixed	3.51 (-66 %)	17.77 (+70 %)	143.41 (+1274 %)	10.44
RefrigeratedTransportation	5.18 (-88 %)	43.64 (+1 %)	388.29 (+799 %)	43.17
RefrigeratedTransportationFixed	4.90 (-89 %)	37.43 (-13 %)	332.81 (+677 %)	42.85
RoomThermostat	3.65 (-75 %)	18.33 (+26 %)	154.18 (+961 %)	14.53
SimpleMarketplace	4.04 (-84 %)	24.22 (-4 %)	206.27 (+717 %)	25.26
SimpleMarketplaceFixed	3.84 (-86 %)	20.36 (-25 %)	171.00 (+531 %)	27.11

Tab. 4.24: Tiempo (s) - Benchmark 1 - STATES

Nombre del contrato	TL=1k	TL=50k	TL=500k	TXB=8
Auction	15.17 (-72 %)	30.08 (-44 %)	153.07 (+183 %)	54.16
Crowdfunding	15.55 (-72 %)	25.66 (-53 %)	131.93 (+142 %)	54.63
EPXCrowdsale	59.63 (-80 %)	96.87 (-68 %)	403.60 (+32 %)	305.58
EscrowVault	21.72 (-78 %)	84.66 (-14 %)	664.75 (+573 %)	98.76
RefundEscrow	13.25 (-79 %)	53.17 (-16 %)	426.01 (+572 %)	63.37
RockPaperScissors	10.38 (-69 %)	28.45 (-14 %)	185.15 (+460 %)	33.07
SimpleAuction	19.07 (-86 %)	48.92 (-63 %)	335.73 (+155 %)	131.56
ValidatorAuction	-	-	-	-

Tab. 4.25: Tiempo (s) - Benchmark 2 - EPA - Con Optimización

Nombre del contrato	TL=1k	TL=50k	TL=500k	TXB=8
Auction	36.77 (-91 %)	139.06 (-65 %)	1024.71 (+157 %)	399.47
Crowdfunding	12.35 (-90 %)	35.46 (-70 %)	250.26 (+110 %)	119.28
EPXCrowdsale	13.61 (-99 %)	96.82 (-98 %)	913.58 (-83 %)	5447.05
EscrowVault	7.88 (-93 %)	60.21 (-48 %)	507.71 (+342 %)	114.99
RefundEscrow	5.41 (-90 %)	34.06 (-35 %)	284.69 (+444 %)	52.33
RockPaperScissors	8.36 (-92 %)	43.95 (-56 %)	356.21 (+259 %)	99.28
SimpleAuction	13.09 (-91 %)	54.24 (-64 %)	418.10 (+175 %)	152.14
ValidatorAuction	-	-	-	-

Tab. 4.26: Tiempo (s) - Benchmark 2 - STATES

Comparando los tiempos de ejecución, se observa que en general, VeriSol con un *tx-Bound* de 8 obtiene resultados en tiempos similares a los de Echidna con un *test limit*

de 50.000, en comparación con los otros valores de *test limit*. De todos modos, para evaluar la eficiencia del uso de una herramienta u otra, es necesario analizar también las abstracciones generadas en cada caso.

Por ejemplo, en contratos pequeños como *DefectiveComponentCounter*, *HelloBlockchain* y *SimpleMarketplaceFixed*, todos pertenecientes al *benchmark 1* y con menos de 25 *queries* generadas cada uno, con un *test limit* de 1.000 se produce la misma abstracción que VeriSol, aún con un menor tiempo de ejecución.

En cambio, en contratos con una mayor cantidad de *queries*, como *Auction* o *Crowdfunding* en modo EPA (ambos con 96 *queries*), recién con un *test limit* de 500.000 se encuentran todos los estados y transiciones, pero a costa de un tiempo de ejecución más de dos veces mayor al de VeriSol.

Cuando Echidna se ejecuta con un *test limit* de 500.000, el tiempo de análisis se incrementa considerablemente con respecto al resto de las ejecuciones. Esto indica que, si bien mayores límites permiten descubrir más transiciones, el costo computacional asociado es significativamente superior al de VeriSol (excepto una, todas las ejecuciones de Echidna con un *test limit* de 500.000 demoraron más que VeriSol con un *transaction bound* de 8).

En resumen, el análisis de las tablas indica que, en términos generales, VeriSol es más eficiente, especialmente cuando el *test limit* de Echidna es alto. No obstante, como se mencionó, para contratos con una baja cantidad de estados abstractos por detectar, podría ser viable ejecutar Echidna con un *test limit* bajo. Esto sugiere la posible implementación de una técnica que estime un *test limit* adecuado en función del tamaño del contrato de entrada y la cantidad de *queries* generadas para el mismo, entre otros factores, lo cual será abordado en la sección de trabajo futuro 6.

4.4. Discusión

Los resultados obtenidos demuestran que es factible generar abstracciones por predicados utilizando Echidna y que, en general, estas abstracciones resultan útiles para representar el comportamiento de los contratos inteligentes. Sin embargo, en algunos casos, la herramienta no logró completar la ejecución dentro de un tiempo razonable. En particular, los contratos *AssetTransfer* y *DigitalLocker* no pudieron ser analizados debido a la gran cantidad de estados generados, lo que provocó que el proceso excediera el tiempo límite establecido.

En cuanto a la cobertura de estados y transiciones, se observa que el *test limit* tiene un impacto directo en la cantidad de información obtenida.

Entonces, los resultados sugieren que incrementar el *test limit* mejora la exploración de estados y transiciones, pero con un aumento considerable en el tiempo de ejecución. Además, se observa que pequeñas modificaciones en el código del contrato, que no afectan su comportamiento, así como la incorporación de predicados en el archivo de configuración, pueden influir significativamente en la capacidad exploratoria de la herramienta y en la cantidad de información obtenida en la abstracción resultante.

Asimismo, la optimización que descarta estados aparentemente inalcanzables (3.5.3) desempeña un papel clave en la cobertura obtenida, facilitando una exploración más eficiente en la mayoría de los casos. La efectividad de esta optimización resalta los beneficios de combinar ambas herramientas, dado que su implementación se basa en la ejecución de VeriSol. Finalmente, los casos en los que la herramienta no logró completar la ejecución sugieren limitaciones en su escalabilidad, lo que podría abordarse en trabajos futuros

mediante estrategias de optimización adicionales.

Ideas sobre cómo optimizar la herramienta para mejorar la cobertura de estados y transiciones, así como reducir el tiempo de ejecución, se discuten en la sección 6.

5. TRABAJO RELACIONADO

A pesar de ser una tecnología relativamente nueva, el análisis de contratos inteligentes ha ganado gran relevancia en los últimos años debido a la veloz adopción de las criptomonedas como activos digitales y a la importancia de prevenir vulnerabilidades y mitigar ataques, que pueden tener consecuencias significativas en términos monetarios.

El desarrollo de contratos inteligentes ha impulsado la creación de diversas herramientas especializadas. Se han realizado diversos estudios y herramientas con el objetivo de analizar la seguridad de contratos inteligentes, ya sea por medio de ejecución simbólica [8] [9] [10] [31], verificación formal [23] [25] [35] o *fuzzing* [4]. En cuanto a visualización, existen herramientas [24] [16] que representan gráficamente la estructura del código y relaciones de herencia en Solidity con el fin de mejorar el entendimiento de los usuarios sobre los contratos, pero no logran capturar el comportamiento dinámico que nuestras abstracciones proporcionan.

La generación y análisis de EPAs también ha sido aplicada en diversos contextos [22] [20] [21]. Un trabajo relevante [28] presenta un método para generar estas abstracciones dinámicamente y utilizarlas para guiar una técnica de generación de tests *search-based*, aunque limitado a programas Java.

Mas aún, ha habido trabajos que exploraron la utilidad, el alcance y las posibles aplicaciones de las EPAs sobre contratos inteligentes de la red Ethereum. La finalidad es que estos modelos ofrezcan utilidad tanto a los desarrolladores como a los auditores de los programas, tanto para mejorar su comprensión sobre los mismos como para validar sus comportamientos.

En el primero de estos trabajos [29] la creación de estas abstracciones se lleva a cabo de manera semi-automática, sin usar concretamente el código del contrato inteligente. En cambio, lo que se realiza es una traducción manual del código a un lenguaje de modelado llamado Alloy [30]. El prototipo usado en este trabajo requiere que se definan las pre y post condiciones de cada función del contrato inteligente, además de los predicados que describen su comportamiento, escritos en Alloy. En base a eso, se ejecuta el analizador de Alloy y se construye la abstracción según los resultados obtenidos.

Otro trabajo vinculado con estos temas aborda la misma problemática generando automáticamente la abstracción del contrato a partir de su código fuente [34]. Esto implica que el usuario no requiere un conocimiento exhaustivo sobre cómo elaborar un modelo de contrato en un lenguaje de modelado particular. En ese estudio se crea una herramienta con el fin de generar abstracciones por predicados automáticamente, utilizando un analizador estático de programas escritos en lenguaje Solidity llamado VeriSol.

La herramienta creada en [34], sin embargo, cuenta con notables limitaciones ligadas al uso de VeriSol. Entre ellas, las más destacadas son:

- Llega hasta la versión 0.6.0 (exclusive), que fue introducida en el año 2019. Además, VeriSol ya no tiene soporte ni mantenimiento desde 2021. Entonces, los contratos actuales no pueden ser analizados con esta herramienta.

  En general, en todos los contratos inteligentes se especifica con qué versión son compatibles y se han incorporado funcionalidades muy importantes y rápidamente

adoptadas por la comunidad desde esa versión hasta hoy en día. Por eso, no se suelen ver muchos contratos con esa versión, especialmente en producción.

- En la red Ethereum, hay ciertas variables globales que son comunes a todos los nodos de la red y suelen tener una influencia sustancial en la ejecución de los contratos inteligentes. Por ejemplo, la variable `block.number` indica cuál fue el último bloque agregado a la blockchain, y esto se suele utilizar como una medida de tiempo en los contratos (a partir de este bloque se libera cierta cantidad de Ether).

  Los valores de estas variables cambian a lo largo de la ejecución de un contrato, pero lamentablemente VeriSol no toma en consideración estos cambios al ser un analizador estático.

Por último, en [36] se propone un enfoque similar al de este trabajo y al de [34], utilizando una herramienta que genera abstracciones por predicados a partir del código fuente del contrato. Sin embargo, en este caso, en lugar de un analizador estático, se emplea una herramienta de ejecución simbólica dinámica llamada Manticore [8]. Una de las principales ventajas de este enfoque es la capacidad de Manticore para emular la red de Ethereum, permitiendo la ejecución de múltiples contratos que interactúan entre sí en un entorno controlado. No obstante, los resultados obtenidos en [36] evidenciaron limitaciones significativas, en particular el alto tiempo de ejecución requerido para generar las abstracciones. Además, Manticore ha dejado de recibir soporte y mantenimiento, siendo su última versión publicada a principios de 2022.

6. TRABAJO FUTURO

A continuación se presentas diversas opciones para continuar este trabajo en el futuro y extender las capacidades de la herramienta desarrollada:

- **Exploración de la implementación de Echidna**

  Una posible mejora sería profundizar en la implementación de Echidna para adaptar su comportamiento a nuestros objetivos. Por ejemplo, aprovechar la existencia de la variable `block.number` y mejorar la distribución de los valores de `msg.value`. En relación al número de bloque, no se encontró el comportamiento deseado al usar la variable global `block.number`, cosa que sí sucedió implementándola por medio de una variable interna y la función `t()`, explicada en 2.3. En cuanto al `msg.value`, al momento de realizar este trabajo, la herramienta no siempre genera valores adecuados para ciertos escenarios, como el envío frecuente del valor 0 (que en muchos contratos provoca casos bordes interesantes), lo que puede limitar la exploración de ciertos estados. Aprovechar estos parámetros permitiría un análisis más representativo del comportamiento real de los contratos, sin necesidad de alterar su código.

- **Optimización mediante *filterFunctions***

  Hemos hablado de distintas optimizaciones orientadas a mejorar la eficiencia de la herramienta, y una de ellas consiste en el uso de *filterFunctions* para reducir la complejidad asociada a la separación de *queries* en múltiples contratos. Actualmente, esta división es necesaria debido al alto número de transiciones a evaluar, pero con el uso de la variable de configuración de Echidna *filterFunctions*, podría ser posible simplificar esta gestión y mejorar el rendimiento global del análisis. Esto se haría creando un solo contrato a ejecutar y corriendo Echidna sólo para un subconjunto de *queries* cada vez, tal como lo hacemos ahora pero sin necesidad de crear múltiples contratos para ello.

- **Restricción del rango de valores en el *fuzzing* mediante ejecución simbólica**

  Una posible optimización consiste en integrar la herramienta con técnicas de ejecución simbólica, permitiendo extraer automáticamente restricciones sobre los valores de entrada que deben cumplir ciertos parámetros para que una función sea ejecutada. En lugar de realizar *fuzzing* sin restricciones o definir manualmente los rangos de valores válidos, la ejecución simbólica podría analizar el código y determinar bajo qué condiciones una función puede ser ejecutada correctamente.

  Por ejemplo, en el contrato *RockPaperScissors*, la función que recibe la elección del jugador debería aceptar únicamente valores en el rango `[0,1,2]`. Actualmente, es necesario modificar manualmente el código del contrato o dejar que el *fuzzing* genere valores arbitrarios, lo que introduce ineficiencias en la exploración de estados. Si se utilizara una herramienta de ejecución simbólica, esta podría inferir que los valores válidos están acotados por la expresión `require(choice >= 0 && choice <= 2)`, y comunicar esta restricción a la herramienta para que realice *fuzzing* exclusivamente dentro de ese rango.

Este enfoque mejoraría significativamente la precisión del análisis, ya que reduciría la cantidad de ejecuciones innecesarias con valores que el contrato nunca aceptaría, optimizando tanto el tiempo de ejecución como la cobertura de estados relevantes.

- **Generación automática del archivo de configuración**

  Actualmente, el *config file* de cada contrato debe ser escrito manualmente, lo que introduce un proceso tedioso para el usuario y propenso a errores. Una mejora clave sería desarrollar un módulo capaz de analizar automáticamente el contrato y generar el archivo de configuración con la estructura y los valores adecuados, extrayendo del contrato sus funciones, precondiciones, variables y demás. Esto reduciría notablemente la carga de trabajo del usuario y permitiría que el análisis sea efectivamente automático.

- **Uso de SMT Solvers para descartar estados contradictorios**

  Una optimización clave para mejorar la precisión y eficiencia del análisis es la incorporación de un *SMT Solver* para detectar y descartar automáticamente estados abstractos que sean inherentemente inalcanzables. Actualmente, la herramienta evalúa transiciones y estados sin verificar si las condiciones requeridas para alcanzarlos contienen contradicciones lógicas, lo que puede llevar a la exploración de estados que en realidad nunca podrían ocurrir en una ejecución real del contrato.

  Por ejemplo, si un estado sólo fuera accesible bajo la siguiente condición:

  $$A \land B \land \neg Q \land \neg A$$

  se puede observar que la presencia simultánea de A y $\neg A$ genera una contradicción lógica, lo que significa que esta condición nunca podrá satisfacerse. En estos casos, la herramienta debería detectar automáticamente la inconsistencia y descartar dicho estado sin necesidad de ejecutar pruebas adicionales, evitando así el uso innecesario de recursos computacionales.

- **Mejora en la estrategia de separación de contratos**

  Otra posible modificación es la optimización del cálculo de la cantidad de contratos en los que se divide una búsqueda. Actualmente, se establece un umbral de 250 *queries*, y si este valor es superado, el contrato se divide en 8 partes. Sin embargo, una estrategia más dinámica podría mejorar tanto el tiempo de ejecución como la exploración del espacio de estados.

  Un enfoque más adaptable sería que la cantidad de contratos generados dependa exclusivamente de la cantidad de *queries*, estableciendo un límite fijo por contrato. Por ejemplo, si se define un máximo de 50 *queries* por contrato, entonces un contrato con 100 *queries* se dividiría en 2, uno con 250 en 5, y así sucesivamente. Esta estrategia podría optimizar especialmente el análisis de contratos muy grandes, en el orden de 1.000+ *queries*.

- **Ejecuciones múltiples para una mejor abstracción**

  En este trabajo, cada contrato fue ejecutado una única vez por configuración, tanto para el análisis temporal como en la generación de abstracciones. Sin embargo, dado

que la herramienta opera de manera dinámica y su ejecución no es completamente determinista, una posible mejora sería realizar múltiples ejecuciones para cada configuración y analizar los resultados en conjunto.

Esto permitiría mejorar la calidad de las abstracciones generadas, ya que diferentes ejecuciones pueden descubrir transiciones o estados que otras no detectaron. Un enfoque viable sería construir la abstracción final como la unión de todas las obtenidas en distintas ejecuciones, logrando así un modelo más completo del comportamiento del contrato. Además, por cada ejecución se podrían definir distintos valores para los parámetros de Echidna, como el balance inicial del contrato y la máxima cantidad de Ether a enviar en cada transacción.

- **Estimar el *test limit* utilizado**

  Los experimentos realizados evidenciaron que el tiempo de ejecución de Echidna aumenta a medida que se incrementa el *test limit* utilizado. En particular, el análisis se llevó a cabo con valores de *test limit* de 1.000, 50.000 y 500.000, observándose una relación directa entre el valor establecido y la duración de la ejecución.

  Asimismo, se constató que para contratos pequeños, un *test limit* de 1.000 era suficiente para identificar todos los estados y transiciones esperadas. Esto sugiere que, en ciertos casos, ejecutar Echidna con valores elevados podría no aportar beneficios significativos, mientras que reducir el *test limit* podría mejorar la eficiencia del proceso.

  Por lo tanto, como trabajo futuro se propone incorporar a la herramienta un mecanismo que estime automáticamente el *test limit* a utilizar en función de las características del contrato a analizar. Este cálculo podría basarse en factores como el tamaño del código fuente y la cantidad de *queries* generadas con la intención de optimizar el equilibrio entre precisión y eficiencia en la generación de abstracciones.

7. CONCLUSIONES

Tomando como punto de partida la necesidad de fortalecer la seguridad y confiabilidad de los contratos inteligentes en Ethereum, este trabajo explora la generación y el uso de abstracciones para analizar su comportamiento y detectar vulnerabilidades. Para esto se introduce el concepto de blockchain, resaltando la importancia de Ethereum y la validación de sus contratos inteligentes para garantizar la seguridad de las aplicaciones descentralizadas.

En paralelo, se introducen formalmente las *enabledness-preserving abstractions* (EPAs), utilizadas en otros dominios, y se analizan distintos tipos de abstracciones ya aplicadas en trabajos previos sobre contratos inteligentes.

A su vez, se presenta también el *fuzzer* Echidna, junto con los fundamentos del *fuzzing*, como herramienta principal para la generación de abstracciones. A partir de ello, se desarrolla una herramienta capaz de construir automáticamente abstracciones por predicados para contratos inteligentes escritos en Solidity, utilizando análisis dinámico mediante *fuzzing*.

Mediante la evaluación de la herramienta sobre un conjunto de contratos inteligentes, se demuestra que es factible generar abstracciones de manera dinámica con Echidna. Además, se analiza la utilidad de estas abstracciones en la detección de vulnerabilidades, comparándolas con diagramas proporcionados por los propios desarrolladores de los contratos, lo que permite identificar errores y comportamientos inesperados en el código.

Observando los resultados, se realiza también una comparación con el uso del analizador estático VeriSol, estudiado previamente en el trabajo [34]. Los resultados muestran que ambas herramientas generan abstracciones similares en la mayoría de los casos, aunque el uso de Echidna conlleva un tiempo de ejecución considerablemente mayor, especialmente en los contratos de mayor tamaño. Además, se observa una cuota de no-determinismo en Echidna, lo que implica que diferentes ejecuciones pueden producir resultados distintos, incluso con los mismos o mayores recursos.

Por otra parte, se aprovecha la posibilidad de combinar ambos analizadores en una misma ejecución, lo que permite mejorar la exploración del espacio de estados. Asimismo, se analiza el impacto del *budget* (*test limit*) en el rendimiento y cobertura de la herramienta, concluyendo que un mayor budget incrementa la cantidad de estados y transiciones exploradas, aunque también aumenta el tiempo de ejecución de manera significativa. También se menciona una posible limitación de Echidna en la generación de valores para alcanzar ciertos estados cuyas precondiciones cuentan con múltiples restricciones sobre diferentes variables.

A modo de conclusión, la herramienta desarrollada representa un aporte valioso para desarrolladores y auditores de contratos inteligentes, al permitir la generación automática de abstracciones útiles mediante análisis dinámico. Sin embargo, se identificaron múltiples áreas de mejora, detalladas en la sección 6, que podrían potenciar aún más el prototipo presentado en este trabajo, ampliando su aplicabilidad y eficiencia en futuros desarrollos.

Bibliografía

[1] American Fuzzy Lop, https://github.com/google/AFL.

[2] Azure Blockchain Workbench, https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples.

[3] Dynamic Smart Contract Abstractor - Herramienta y resultados, https://github.com/igrinspan/dynamic-smart-contract-abstractor.

[4] Echidna: A Fast Smart Contract Fuzzer, https://github.com/crytic/echidna.

[5] Ethereum Virtual Machine. https://ethereum.org/en/developers/docs/evm/.

[6] Fuzzing, https://www.fuzzingbook.org/.

[7] LibFuzzer, a library for coverage-guided fuzz testing, https://llvm.org/docs/LibFuzzer.html.

[8] Manticore: Symbolic Execution for Humans, https://github.com/trailofbits/manticore.

[9] Mythril: A security analysis tool for EVM bytecode, https://github.com/Consensys/mythril.

[10] Pakala: A tool to search for exploitable bugs in Ethereum smart contracts, https://github.com/palkeo/pakala.

[11] Slither, the smart contract static analyzer, https://github.com/crytic/slither.

[12] SmartPulse Evaluation Benchmarks, https://github.com/utopia-group/SmartPulseTool/tree/master/benchmarks/evalBenchmarks.

[13] Solidity Compiler Versions, https://soliditylang.org/blog/category/releases/.

[14] Solidity Language Portal, https://soliditylang.org.

[15] Solidity Language Repository, https://github.com/ethereum/solidity.

[16] Surya, The Sun God: A Solidity Inspector. https://github.com/ConsenSys/surya.

[17] VeriSol: A formal verifier for Solidity based smart contracts, https://www.microsoft.com/en-us/research/project/verisol-a-formal-verifier-for-solidity-based-smart-contracts/publications/.

[18] Pedro Antonino and A. W. Roscoe. Solidifier: bounded model checking solidity using lazy contract deployment and precise memory modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1788–1797, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. `https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf`.

[20] Guido Caso, Víctor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Abstractions for validation in action. volume 7320, pages 192–218, 06 2012.

[21] Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 381–390, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Automated abstractions for contract validation. *IEEE Transactions on Software Engineering*, 38(1):141–162, 2012.

[23] Wang Duo, Huang Xin, and Ma Xiaofeng. Formal analysis of smart contract based on colored petri nets. *IEEE Intelligent Systems*, 35(3):19–30, 2020.

[24] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. *CoRR*, abs/1908.09878, 2019.

[25] Ikram Garfatta, Kaïs Klai, Mohamed Graïet, and Walid Gaaloul. Model checking of vulnerabilities in smart contracts: a solidity-to-cpn approach. SAC '22, page 316–325, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. Technical Report MSR-TR-2007-154, November 2007.

[27] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. November 2008.

[28] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastián Uchitel. Enabledness-based testing of object protocols. *ACM Trans. Softw. Eng. Methodol.*, 30(2), January 2021.

[29] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastian Uchitel. Predicate abstractions for smart contract validation. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS '22, page 289–299, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, aug 2019.

[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Charlie Miller and Zachary N. J. Peterson. Analysis of mutation and generation-based fuzzing. 2007.

[33] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.

[34] Edén Torres. *Generador de abstracciones para smart contracts.* PhD thesis, Universidad de Buenos Aires, 2023.

[35] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain, 2019.

[36] Daniel Wappner. *Construcción de Abstracciones de comportamiento para contratos inteligentes mediante ejecución simbólica.* PhD thesis, Universidad de Buenos Aires, 2024.

[37] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, 2017.