Abstracción de contratos inteligentes mediante ejecución simbólica dinámica

Daniel Wappner[1]
dwappner@dc.uba.ar

Universidad de Buenos Aires - Facultad de Ciencias Exactas y Naturales, Pabellón II
Ciudad Universitaria - CABA (cp:1428), TEL (54.11) 4576.3300/09

Abstract. Los contratos inteligentes son programas inmutables que se despliegan en una blockchain. Dado que a menudo manejan activos de alto valor real, su verificación y validación antes de desplegarlos es de gran importancia. Por esta razón, es una práctica común contratar empresas de seguridad especializadas para auditar el código de los contratos inteligentes. Sin embargo, se han explotado numerosas vulnerabilidades en los últimos años provocando pérdidas a miles de personas.
En este trabajo presentamos el desarrollo de un prototipo que, dado el código fuente de un contrato inteligente, genera máquinas de estado finitas que abstraen el comportamiento del contrato. Estas abstracciones que se basan en predicados sobre la habilitación de los métodos del contrato han resultado útiles anteriormente como herramienta para la validación de código contra especificaciones informales y para descubrir errores latentes. El prototipo implementado hace uso y extensión de una herramienta open source de ejecución simbólica dinámica denominada Manticore. Además, hacemos pública la implementación del prototipo, junto con las pruebas realizadas contra contratos ejemplo.

Keywords: Contrato Inteligente · Ejecución simbólica dinámica · Validación por Modelos.

1 Introducción

La tecnología de blockchain permite mantener un registro inmutable, transparente y descentralizado de las transacciones que ocurren en ella. Algunas redes permiten la ejecución de software de esta misma manera, llamando contratos inteligentes a las aplicaciones presentes en la red. Esta capacidad de cómputo está íntimimamente integrada al diseño de la blockchain, y suele estar atada a un modelo de cómputo particular como la *EVM* de Ethereum [1] o la *AVM* de Algorand [2].

La tecnología de blockchain garantiza la correcta ejecución de los contratos inteligentes, y la inmutabilidad de las transacciones registradas en la red asegura que estos últimos no pueden ser modificados, otorgando garantía a los usuarios de que el comportamiento de los contratos se mantendrá siempre estable. Sin embargo, esto significa que los defectos en la implementación de los contratos

inteligentes no pueden ser reparados, y las transacciones no deseadas producto de estos defectos no pueden ser revertidas. Por eso la validación y verificación de los contratos inteligentes antes de desplegarlos son de suma importancia si se quieren evitar defectos. De hecho, históricamente los ataques a contratos inteligentes para abusar *bugs* han causado grandes pérdidas materiales a miles de personas [9].

Por esto, es común en la industria contratar a auditores independientes para esta tarea, quienes a menudo usan herramientas para facilitar la validación del código fuente, y a menudo cuentan sólo con especificaciones informales del comportamiento. En estas situaciones, una técnica útil para la validación es la construcción de abstracciones del comportamiento del contrato en una máquina de estados finita [3]. Estas máquinas abstraen al nivel de 'llamado a función", y están fuertemente basadas en las *EPAs* (Enabledness Preserving Abstraction) [8] que son máquinas de estado que describen las posibles secuencias de llamados a funciones del contrato. Para construir estas abstracciones Godoy et al. [3] utiliza un prototipo donde traducen manualmente las pre y post condiciones de los contratos inteligentes a un lenguaje intermedio, Alloy [5].

En este trabajo nos enfocamos en la construcción de EPAs para contratos inteligentes implementados en Solidity [4] utilizando ejecución simbólica dinámica. Presentamos un prototipo que trabaja automáticamente sobre el código fuente, con mínima intervención manual. Hacemos uso de Manticore, una herramienta open source para ejecución simbólica desarrollada por trailofbits [6]. Manticore respalda la simulación de una blockchain completa, manteniendo el estado de múltiples contratos y usuarios dentro de la red. El trabajo fue realizado dentro del programa Becas de Iniciación a la Investigación en Ciencias de la Computación, otorgado por el ICC (Instituto de Ciencias de la Computación) bajo la dirección de Javier Godoy, Diego Garbervetsky, Juan Pablo Galeotti y Sebastián Uchitel.

2 Contratos Inteligentes y Solidity

Una blockchain es una estructura similar a un libro de contabilidad distribuido en el que las entradas son inmutables. A menudo, estas entradas describen propiedades sobre *direcciones* en la blockchain, como ocurre con el balance de criptomonedas en la blockchain de Bitcoin o en la Ethereum blockchain. Las transacciones se agrupan en conjuntos de *bloques*, que son la unidad mínima con la que se actualiza el estado de la blockchain. El historial de bloques agregados a la blockchain (y por ende, las transacciones ejecutadas) es público, y el estado actual de la red puede calcularse a partir de este historial. Además, los protocolos de consenso implementados otorgan garantías de seguridad a la red de que ningún actor podrá forzar el ingreso de información falsificada a ella [7].

Ethereum, entre otras blockchain, codifica en la blockchain un estado de cómputo distribuido, y permite modificar este estado mediante aplicaciones programadas llamadas contratos inteligentes. El código ejecutable de los contratos inteligentes se encuentra almacenado en direcciones dentro de la blockchain, de la misma forma que es almacenada la información de un usuario tradicional. Esto

significa que al igual que las transacciones pasadas de la blockchain no pueden cambiarse, tampoco puede cambiarse el código de un contrato inteligente. El código de los programas consiste en bytecode que se ejecuta en una máquina de pila conocida como la *EVM* o *Etherum Virtual Machine* [1]. Este código es ejecutado siempre que el contrato se selecciona como destinatario de una transacción, siendo necesario calcular el estado final de la ejecución de la *EVM* para conocer el siguiente estado de la blockchain. La *EVM* es un modelo de cómputo casi turing completo. Está limitado únicamente por la cantidad de operaciones que se pueden ejecutar dentro de una misma transacción, debido a que el remitente de la transacción debe pagar una cantidad en *gas* por cada instrucción ejecutada al nodo que publique el próximo bloque.

Para programar los contratos inteligentes, Ethereum provee Solidity. Es un lenguaje imperativo curly-brace que provee para los contratos inteligentes interfaces similares a las de las clases en lenguajes orientados a objetos. En la figura 1a presentamos un ejemplo de un programa en Solidity llamado `SimpleMarketplace`. Las primeras líneas indican las variables de estado del contrato, diferenciando aquellas que serán accesibles desde contratos externos de las que no, mientras que las siguientes implementan métodos. Una instancia de `SimpleMarketplace` comienza mediante un llamado al constructor, en el que el `InstanceOwner` publica un objeto que planea vender, indicando la descripción y el precio del producto. Luego, un potencial comprador puede realizar ofertas sobre el producto llamando a `MakeOffer`. Si el dueño original no está satisfecho con la oferta puede rechazarla llamando a `RejectOffer`, regresando a un estado en el que se aceptan nuevas ofertas. Si eventualmente un comprador realiza una oferta que es de interés al dueño, este puede aceptarla llamando a `AcceptOffer`, terminando satisfactoriamente la ejecución del contrato.

3 Enabledness-Preserving Abstractions

Una EPA busca abstraer el comportamiento de un contrato a un Labeled Transition System (LTS) finito, en el que se agrupan los estados en base a qué metodos están habilitados [8]. Las trancisiones en estos LTS representan el llamado a una función del contrato. En la figura 1b presentamos la EPA de SimpleMarketplace. Luego de ejecutar el constructor se llega al estado B. La etiqueta `_MakeOffer` indica que es el único método que se encuentra habilitado en ese estado. La transición del estado B al estado C indica que existe un llamado a `MakeOffer` que nos hace llegar a un estado del contrato donde los métodos habilitados son `AcceptOffer` y `Reject`. Desde el estado C ejecutar `Reject` nos llevará al estado B y ejecutar `AcceptOffer` nos llevará al estado D. La etiqueta del estado D indica que ningún método se encuentra habilitado, por lo que representa el fin forzoso de la ejecución. Este comportamiento se asemeja a la descripción semántica del contrato que dimos en la sección anterior.

Antes de exhibir la construcción de EPAs, es necesario presentar brevemente la formalización necesaria de los contratos inteligentes y las abstracciones utilizadas. Estas formalizaciones de los artefactos de código son propuestas por De

```
contract SimpleMarketplace {
    enum StateType {ItemAvailable, OfferPlaced, Accepted}
    address public InstanceOwner;
    string public Description;
    int public AskingPrice;
    StateType public StateEnum;
    address public InstanceBuyer;
    int public OfferPrice;

    constructor(string memory description, int price, address sender) public{
        InstanceOwner = sender;
        AskingPrice = price;
        Description = description;
        StateEnum = StateType.ItemAvailable;
    }

    function MakeOffer(int offerPrice) public{
        require (offerPrice != 0 && StateEnum == StateType.ItemAvailable && InstanceOwner == msg.sender);
        InstanceBuyer = msg.sender;
        OfferPrice = offerPrice;
        StateEnum = StateType.OfferPlaced;
    }

    function Reject() public{
        require ( StateEnum == StateType.OfferPlaced  && InstanceOwner == msg.sender);
        StateEnum = StateType.ItemAvailable;
    }

    function AcceptOffer() public{
        require ( StateEnum == StateType.OfferPlaced && msg.sender == InstanceOwner );
        StateEnum = StateType.Accepted;
    }
}
```

Fig. 1(a): Contrato Inteligente `SimpleMarketplace` en Solidity

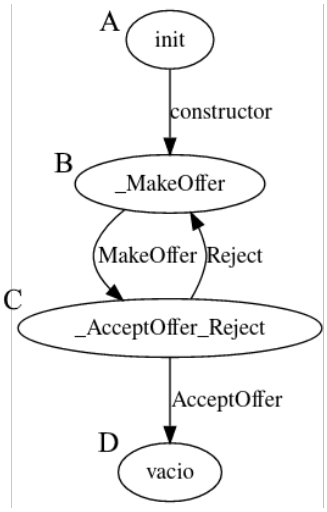

Fig. 1(b): EPA de `SimpleMarketplace`. Las etiquetas en los estados indican los métodos que se encuentran habilitados. Las etiquetas en las transiciones indican el método por el que ocurre la transición.

Caso et al. [8] y extendidas para considerar que el estado del programa dependa del estado global de la blockchain por Godoy et al. [3].

3.1 Modelo formal

Dado que trabajamos sobre el código fuente de un contrato escrito en Solidity, es necesario formalizar qué aspectos del contrato consideramos relevantes. Llamaremos C al conjunto de configuraciones, donde una configuración describe el estado de las variables de estado del contrato y el estado de la blockchain.

Definición 1. *(Formalización de un contrato inteligente) Definimos a un contrato inteligente como la tupla* $SC = \langle M, F, R, inv, init \rangle$ *donde:*

- $M = m_1, \dots m_n$ es el conjunto finito de métodos definidos en la interfaz del contrato
- F es un conjunto de funciones indexadas por M.
  Para cada $F_m \in F$, $F_m : C \times \mathbb{Z} \to (C \cup \bot)$ implementa el método m.
- R es un conjunto de precondiciones indexado por M.
  Para cada $R_m \in R$, $R_m : C \times \mathbb{Z} \to \{\textbf{true}, \textbf{false}\}$ indica si el método m está habilitado para la configuración y parámetros indicados
- $inv : C \to \{\textbf{true}, \textbf{false}\}$ indica si se cumple el invariante del contrato
- $init : C \to \{\textbf{true}, \textbf{false}\}$ indica si la configuración puede ser resultante de ejecutar los constructores del contrato

Una particularidad presente en los métodos de los contratos inteligentes es que siempre tienen dos parámetros implícitos, `(msg.sender)` que indica qué *account* en la blockchain efectuó la transacción y `(msg.value)` que indica la cantidad de criptomonedas transferidas. Sin embargo, podemos codificar los parámetros implícitos y explícitos en $\mathbb{Z}$[1] sin pérdida de generalidad. La semántica de de un contrato la definimos como el siguiente *Labeled Transition System*.

Definición 2. *(Semántica de un contrato inteligente) Dado* $SC = \langle M, F, R, inv, init \rangle$ *un contrato inteligente, su semántica está provista por el LTS concreto* $L_c = \langle \sigma, S_c, S_{0c}, \Delta_c \rangle$ *que satisfaga:*

- $S_c = \{conf | conf \in C \wedge inv(conf) = \textbf{true}\}$
- $S_{0c} = \{conf | conf \in S_c \wedge init(conf) = \textbf{true}\}$
- $\sigma = (F \times \mathbb{Z}) \cup \tau$ es el conjunto de todos los posibles llamados a funciones, junto con un símbolo τ que representa los cambios en la blockchain que ocurren independientemente del contrato
- $\Delta_c \subseteq S_c \times \sigma \times S_c$
- $\forall s_1, s_2 \in S_c, m \in M, z_1 \in \mathbb{Z}$.
  $(s_1, (F_m, z_1), s_2) \in \Delta_c \iff \Big(R_m(s_1, z_1) = \textbf{true} \wedge F_m(s_1, z_1) = s_2 \Big)$
  $(s_1, \tau, s_2) \in \Delta_c \iff$ puede ocurrir un cambio en la blockchain partiendo del estado s_1 llegando al estado s_2

[1] $\mathbb{Z}$ es el conjunto de los números enteros.

Notar que el conjunto S_c de estados del LTS concreto de un contrato es infinito. Para cada diferencia posible en las variables internas o de la blockchain, habrá s_1 y s_2, dos estados distintos representando esa diferencia. Esto es verdad incluso para contratos donde las configuraciones de las variables internas es finita, pues siempre habrá infinitas configuraciones de las variables de la blockchain.

A la EPA (es decir, el LTS abstracto) la definimos entonces de la siguiente manera: El conjunto de estados es 2^R (el conjunto de partes de las precondiciones). La forma en la que abstraemos los estados s_c del LTS concreto es $\alpha(s_c) =$ "el conjunto de precondiciones satisfechas por s_c". Una transición en la EPA etiquetada con el método m entre los estados s y s' significa que existe algún estado concreto s_c para el que $\alpha(s_c) = s$ y un valor de entrada z tal que $F_m(s, z) = s'_c$ con $\alpha(s'_c) = s'$. Una transición de s a s' etiquetada con $\hat{\tau}$, la versión abstracta de τ, indica que existe un estado concreto s_c con $\alpha(s_c) = s$ y que puede ocurrir $\alpha(\tau(s_c)) = s'$. Formalmente:

Definición 3. *(Enabledness-Preserving-Abstraction) Dado* $SC = \langle M, F, R, inv, init\rangle$ *un contrato inteligente y* $L_c = \langle \sigma, S_c, S_{0c}, \Delta_c\rangle$ *su LTS concreto, entonces el LTS asbtracto* $L_A = \langle M \cup \hat{\tau}, 2^R, P_0, \Delta_A\rangle$ *es una EPA del contrato con* $\alpha : S_c \to 2^R$ *la función de abstracción, donde se cumple que:*

- 2^R es el conjunto de partes de R
- $\forall s \in S_c \,.\, \alpha(s) = \{R_m | R_m \in R \wedge \exists z \in \mathbb{Z}. R_m(s, z) = \textbf{true}\}$
- $P_0 = \{\alpha(s_0) | s_0 \in S_{0c}\}$
- $\forall s_1, s_2 \in S_c, m \in M, z_1 \in \mathbb{Z}.$
  $(s_1, (F_m, z_1), s_2) \in \Delta_c \implies (\alpha(s_1), m, \alpha(s_2)) \in \Delta_A$
  $(s_1, \tau, s_2) \in \Delta_c \implies (\alpha(s_1), \hat{\tau}, \alpha(s_2)) \in \Delta_A$

4 Construcción de *EPAs*

Tradicionalmente, existe un algoritmo genérico para la construcción de *EPAs* de artefactos de código. Este busca cuáles estados de la EPA pertenecen a P_0 y luego realiza BFS (*Breadth-First-Search*) sobre la porción alcanzable de la máquina de estados de la EPA [8]. Para decidir si una transición pertenece o no la EPA, es necesario resolver problemas de validez de fórmulas del estilo $\exists z \,.\, R_m(s_0, z) = \textbf{true}$ para s y m fijos, lo que puede ser indecidible. El algoritmo de construcción de *EPAs* propone transformar estas preguntas de validez de fórmulas de primer orden en problemas de alcanzabilidad de código, dado que espera que las precondiciones y los invariantes definidos en la formalización 1 estén debidamente implementados en el contrato.

Implementar el invariante o las precondiciones de los métodos del contrato manualmente es a menudo exigente para el programador y propenso a errores. Sin embargo, es importante en la definición 2 del LTS la presencia del invariante, de lo contrario el comportamiento de la EPA sería demasiado sobreaproximado [8]. Por eso, en continuación del trabajo presentado por Godoy et al. [3], trabajaremos con el invariante implícito dado por los métodos del contrato y con precondiciones implícitas dadas por las declaraciones `require` al comienzo de los métodos.

Algoritmo 1 Construcción de EPAs mediante ejecución simbólica

Input $C = \langle M, F, R \rangle$ contrato
Output La *EPA* $L_A = \langle \Sigma, S, P_0, \Delta \rangle$

$\Sigma = M$
$S = \emptyset$; $P_0 = \emptyset$
$\Delta(s, m) = \emptyset \quad \forall s \in 2^R, m \in M$
$Paths_{pre} = \{p | p$ es un posible camino de ejecución de $F_{Constructor}; R\_1; R\_2; \dots R\_n\}$
$Symb_{pre} = \{$resultado simbólico luego de ejecutar $p | p \in Paths_{pre}\}$
$P_0 = \{s \in 2^R | \exists sy \in Symb_{pre} \,.\, SAT(sy = s)\}$
$S_{current} = P_0$
$marcados = \emptyset$
$W =$ Pila vacía ;W.Push($(S_{current}, Symb_{pre})$)
while W.NotEmpty() **do**
    $S =$ Update con $S_{current}$
    if $\exists m \in M \,.\, \{s \in S_{current} | m \in s \wedge (s, m) \notin marcados\} \neq \emptyset$ **then**
        Elegir tal m
        $Paths_{post\_m} = \{p | p$ es un posible camino de ejecución de $F_m; R\_1; R\_2; \dots R\_n\}$
        $Symb_{post} = \{$resultado simbólico luego de ejecutar $p1; p2 \mid p1 \in Paths_{pre} \wedge p2 \in Paths_{post\_m}\}$
        $S_{next} = \emptyset$
        for $\{s_1 \in S_{current} | m \in s_1 \wedge (s_1, m) \notin marcados\}$ **do**
            $marcados \mathrel{+}= \{(s_1, m)\}$
            $\Delta(s_1, m) = \{s_2 \in 2^R | \exists sy_1 \in Symb_{pre}, sy_2 \in Symb_{post\_m} \,.\, SAT(sy_1 = s_1 \wedge sy_2 = s_2)\}$
            $S_{next} = \Delta(s_1, m)$
        end for
        $S_{current} = S_{next}$
        $Symb_{pre} = Symb_{post}$
        $Paths_{pre} = (Paths_{Pre};\ Paths_{Post})$
        W.Push($(S_{current}, Symb_{pre})$)
    else
        $(S_{current}, Symb_{pre}) = W$.Pop()
    end if
end while
return $\langle \Sigma, S, P_0, \Delta \rangle$

El algoritmo 1 realiza una versión modificada de DFS (*Depth-First-Search*) sobre la porción alcanzable de la EPA en el que se exploran varios nodos del grafo a la vez. Para esto, utiliza herramientas de ejecución simbólica y smt-solving para encontrar las transiciones.

El estado inicial y la función de transición comienzan vacíos. El conjunto $Paths_{pre}$ contiene todos los caminos posibles que puede tomar la ejecución del constructor seguida por la ejecución de las precondiciones de los métodos. Si llamamos *init* al estado (simbólico) luego de ejecutar $F_{Constructor}$, en $Symb_{pre}$ hay para cada uno de los caminos en $Path_{pre}$ una expresión que representa el resultado de la ejecución realizada. Esa expresión es de la forma $(R_1(init) = r_1, R_2(init) = r_2 \cdots, R_n(init) = r_n)$. Luego, puede calcularse P_0 mediante una sucesión de preguntas de satisfacibilidad de fórmulas. En particular, para cada expresión $sy \in Symb_{pre}$, nos interesa saber con qué estado de la EPA es consistente sy.

Por ejemplo, en la construcción de la EPA de SimpleMarketplace, sean $s_1 = \{R_{MakeOffer}, R_{Reject}\}$ y $sy \in Symb_{Pre}$ tal que los resultados de las precondiciones en sy son $\{R_{MakeOffer}(init) = r_1, R_{AcceptOffer}(init) = r_2, R_{Reject}(init) = r_3\}$. La fórmula $(r_1 = \textbf{true} \wedge r_2 = \textbf{false} \wedge r_3 = \textbf{true} \wedge sy)$ significa que para el camino de ejecución asociado a sy, existen valores de entrada que lo recorren y que llegan a un estado final en el que `MakeOffer` y `Reject` se encuentran habilitados (o, lo que es lo mismo, un estado que se abstrae a s_1). En la línea 6 del algoritmo notamos a esta fórmula que expresa que el camino asociado a sy es compatible con s_1 como $(sy = s_1)$.

Habiendo determinado P_0, el algoritmo marca $S_{current}$ como el conjunto de estados de la EPA con los que se corresponde $Symb_{pre}$, y agrega ambos a la pila W, que representa conjuntos de estados de la EPA por visitar. Para continuar la exploración desde $S_{current}$, se elige uno de los métodos que esté habilitado en alguno de los estados de $S_{current}$ y no se haya explorado ya. Luego, para cada estado $s \in S_{current}$ que permite la ejecución de m, se calculan los nuevos estados a los que se puede llegar ejecutando m, de manera similar a la que se calcula P_0.

El conjunto $Symb_{post}$ de la línea 15 representa los resultados de continuar la ejecución donde la había dejado $Symb_{pre}$. En la línea 19, donde se calculan las próximas transiciones, es necesario revisar no sólo la satisfacibildad de que el estado simbólico nuevo sea compatible con el estado de la EPA objetivo, sino que es necesario que aún se mantenga la compatibilidad entre el estado simbólico viejo y el estado de la EPA desde el que se está transisionando. Esto es porque las expresiones en $Symb_{pre}$ pueden ser compatibles con s_1, pero puede ocurrir una contradicción al pedir que $Symb_{post}$ sea compatible con s_2 también. De no exigir esto, consideraríamos que valores de entrada que permiten alcanzar s_2 pero que nunca alcanzaban s_1 previamente indican una transición entre s_1 y s_2.

5 Análisis

El prototipo implementado [11] utiliza Manticore para llevar a cabo el algoritmo 1. Para esto genera una simulación de la blockchain en la que se despliega el contrato, y luego realiza ejecución simbólica de sus métodos llamándolos desde *accounts* externas en la blockchain. Las limitaciones de la herramienta exigen que el parámetro `(msg.sender)`, a pesar de que pueda ser simbólico, se corresponda con una *account* definida explícitamente por el usuario. Por esto en la simulación contamos con un número fijo de *accounts*, asignando variables simbólicas únicamente la información asociada a ellas (Dirección, Balance, etc).

Al evaluar el funcionamiento del prototipo buscamos responder las siguientes preguntas:

1. ¿Son correctas las EPAs que genera el prototipo?
2. ¿Cuál es su performance en contratos inteligentes reales?

Para responder estas preguntas, pusimos el prototipo a prueba contra algunos contratos provenientes de Microsoft Azure Blockchain Workbench [10]. Este conjunto de contratos había sido utilizado anteriormente por Godoy et al. [3], por lo que convenientemente ya contamos con las EPAs correspondientes. Ejecutamos

el prototipo cinco veces, obteniendo el promedio de su tiempo de ejecución sobre los contratos seleccionados y luego corroboramos que la EPA generada fuera isomórfica a la obtenida en los estudios anteriores. Los resultados intermedios indicaron que para los contratos propuestos era suficiente realizar la simulación con dos *accounts*, por lo que utilizamos esa cantidad para los experimentos. Los resultados de esta experimentación se ven resumidos en la tabla 1. El prototipo generó EPAs correctas en todos los casos. Sin embargo, el tiempo de cálculo es de entre media y casi tres horas, considerando incluso que los ejemplos utilizados son relativmente pequeños. Algunos resultados intermedios indican que este tiempo es consumido principalmente por Manticore para la generación de *path conditions*. Debido a la rigurosidad con la que emula Manticore el comportamiento de la blockchain, la herramienta demora incluso para la ejecución simbólica de transiciones sencillas.

Tabla 1: Resumen de la experimentación. **LOC** es cantidad de lineas de código, **Tiempo de ejecución** es el promedio del tiempo de ejecución medido en minutos, $\boldsymbol{\sigma}$ es el desvío estandard medido en segundos y **¿Es correcto?** indica si la EPA generada es isomorfa con la provista anteriormente.

Contrato	LOC	Tiempo de ejecución (min)	σ (s)	¿Es correcto?
DefectiveComponentCounter	33	29	7	Sí
SimpleMarketplace	66	186	800	Sí
BasicProvenance	48	40	6	Sí
RoomThermostat	48	138	600	Sí

6 Otras limitaciones y líneas de trabajo

La forma de construir la EPA garantiza que cada transición encontrada tiene un caso de test que la deja en evidencia. El estado simbólico que construimos es el resultado de ejecutar una sucesión de métodos del contrato que comienza siempre por el constructor. Esto significa que las soluciones otorgadas por el smt-solver son valores de entrada que pueden ser usados para realizar una ejecución concreta que se corresponda con el camino encontrado en la EPA. De hecho, el prototipo implementado utiliza funcionalidades de Manticore para generar estos casos de test automáticamente, guardándolos en memoria secundaria.

Sin embargo, esta manera de construir la EPA no explora todas las instancias concretas que se le corresponden a cada estado abstracto. En cambio, sólo explora las instancias que puedan ser producto de la traza de ejecución “elegida”. Es posible que el prototipo actual nunca encuentre algunas transiciones en la EPA, dependiendo del orden elegido para los métodos en la línea 13 del algoritmo 1.

Esta limitación no está presente en el algoritmo de construcción de EPAs original [8]. Con una implementaciçón del invariante del contrato podría conseguirse un estado simbólico genérico que satisfaga las condiciones de un estado de la EPA. Buscar un estado totalmente genérico que no se restrinja al cumplimiento del invariante resulta en una sobreaproximación demasiado burda [8]. Además, es posible que razonar sobre todas las variables a la vez, como lo hace el invariante, sea comparativamente costoso en tiempo. Actualmente estamos poniendo a prueba estas diferencias contra una implementación del algoritmo original.

Por otro lado, es importante señalar que el prototipo demostrado no implementa los cambios en la blockchain representados por la función τ. Esta limitación puede resolverse modelando el comportamiento de τ mediante algunas modificaciones específicas a variables de la blockchain, como el número de bloque y el balance de accounts externas. En la actualidad estamos poniendo a prueba implementaciones de esto contra contratos cuya EPA posee estados que dependen del número de bloque.

7 Conclusiones

En este trabajo proponemos un método para construir EPAs mediante ejecución simbólica. Construimos un prototipo que funciona a partir de código fuente Solidity, generando la EPA y casos de test para las transiciones encontradas de forma automática. El análisis sobre el prototipo implementado demuestra que genera aproximaciones satisfactorias de las EPAs, en tiempos de entre 30 y 190 minutos. Además, hacemos pública la implementación de este prototipo. [11]

Referencias

1. Ethereum Yellow Paper https://ethereum.github.io/yellowpaper/paper.pdf
2. Algorand Virtual Machine https://developer.algorand.org/docs/get-details/dapps/avm/
3. Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, Sebastian Uchitel. 2022. Predicate abstractions for smart contract validation. In: Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems (MODELS).
4. Lenguaje Solidity https://docs.soliditylang.org/en/latest/
5. Lenguaje de especificación Alloy https://github.com/AlloyTools/org.alloytools.alloy
6. Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, Artem Dinaburg. 2020. Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).
7. Sitio web de Ethereum, protocolos de consenso. https://ethereum.org/en/developers/docs/consensus-mechanisms/
8. Guido De Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2013. Enabledness-based program abstractions for behavior validation. ACM Trans. Softw. Eng. Methodol. 22, 3, Article 25 (July 2013), 46 pages.
9. DAO Attack (2016) https://www.nytimes.com/2016/06/18/business/dealbook/hacker-may-have-removed-more-than-50-million-from-experimental-cybercurrency-project.html
10. Microsoft Azure Blockchain Workbench https://github.com/Azure-Samples/blockchain/tree/master/blockchain-workbench/application-and-smart-contract-samples
11. Repositorio del prototipo implementado. https://github.com/DaniWppner/manticore-predicate-abstraction