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

Verificación de contratos en Algorand

Tesis de Licenciatura en Ciencias de la Computación

Patricio Sabogal

Director: Diego Garbervetsky

Buenos Aires, 2024

VERIFICACIÓN DE CONTRACTOS EN ALGORAND

La creciente adopción de tecnologías blockchain ha generado un interés significativo en la seguridad y confiabilidad de los contratos inteligentes. Algorand, una plataforma blockchain de código abierto, utiliza el lenguaje de script `TEAL` (*Transaction Execution Approval Language*) para escribir contratos inteligentes .

La verificación formal de estos contratos es esencial para garantizar su correctitud y prevenir vulnerabilidades. La presente tesis tiene como objetivo desarrollar un enfoque de verificación de `TEAL` utilizando un acercamiento de traducción a un lenguaje intermedio llamado Boogie [7] , que ha sido utilizado exitosamente para la traducción de varios lenguajes [14, 10]

El enfoque aplicado se basa en traducir de `TEAL` a `Boogie` y la de utilizar la herramienta `Corral` [13] como verificador, idea similar utilizada en `VeriSol` [10] para la verificación de contratos inteligentes en Ethereum. Al traducir `TEAL` a `Boogie` se podrá utilizar `Corral` para realizar la verificación formal del código resultante.

Keywords: Algorand, software verification, TEAL, formal verification, Boogie

A mi familia, amigos, los cuarenta ladrones y los héroes de la Avenida.

Índice general

1.. Introducción 1

1.1. Aportes 1

1.2. Metodología 2

2.. Preliminares 3

2.1. Introducción a Algorand 3

2.2. Introducción a `Boogie` 7

2.3. Introducción a `Corral` 9

2.3.1. Ejemplo de `Corral` 11

3.. VeriTEAL 13

3.1. Traducción de `TEAL` a `Boogie` 13

3.1.1. Tealift 15

3.1.2. Comparación de métodos 21

3.1.3. Representación del estado de la AVM basada en registros 24

3.2. Generación de la interfaz del contrato 24

3.3. Validación de condiciones 28

3.3.1. Generación 28

4.. Evaluación 30

4.1. Funcionalidad 30

4.2. Caso de estudio: Tinyman 33

5.. Consideraciones y limitaciones 36

5.1. Consideraciones específicas 36

6.. Trabajos relacionados 37

7.. Conclusiones 38

1. INTRODUCCIÓN

La tecnología *blockchain* ha revolucionado la manera en que gestionamos y transferimos datos, ofreciendo nuevas posibilidades para la creación de aplicaciones descentralizadas seguras y confiables. Algorand, una innovadora plataforma blockchain, utiliza un algoritmo de consenso basado en prueba de participación y una máquina virtual avanzada llamada *Algorand Virtual Machine* (AVM) para ejecutar contratos inteligentes. Estos contratos, escritos en el lenguaje `TEAL` (*Transaction Execution Approval Language*), permiten la implementación de lógica compleja directamente en la *blockchain*, lo que facilita una amplia gama de aplicaciones, desde la gestión de activos hasta la automatización de procesos.

La importancia de los contratos inteligentes ya ha sido discutida en detalle, destacando su impacto en sectores como finanzas, gestión de la cadena de suministro, atención médica y procesos legales. La seguridad de los contratos inteligentes es esencial para mantener la integridad de las plataformas *blockchain*, proteger a los usuarios, y fomentar la confianza en la tecnología. Requiere una combinación de prácticas de programación seguras, pruebas rigurosas, auditorías y monitoreo continuo para identificar y mitigar posibles vulnerabilidades. En materia de pruebas rigurosas, la verificación formal es una herramienta fundamental[10] para garantizar el correcto funcionamiento de los contratos inteligentes. Proporciona una manera de demostrar matemáticamente la correctitud de la lógica del contrato, fomentando la confianza entre los usuarios y las partes interesadas. Por la naturaleza de los contratos inteligentes, errores de la implementación pueden resultar pérdidas multimillonarias, como lo fue el caso de Tinyman [11], donde atacantes robaron 2.9M USD.

La seguridad y la correcta ejecución de estos contratos inteligentes son fundamentales, ya que cualquier vulnerabilidad puede resultar en pérdidas significativas y comprometer la integridad de la red. Por lo tanto, la verificación formal de contratos inteligentes es una tarea crítica que asegura que estos funcionen según lo previsto y sin errores. La verificación formal implica el uso de métodos matemáticos y herramientas automatizadas para demostrar la correctitud de los contratos, proporcionando una capa adicional de confianza para los usuarios y desarrolladores.

1.1. Aportes

En esta tesis, se propone un enfoque para la verificación formal de contratos inteligentes en Algorand. Este enfoque se basa de la traducción de `TEAL` a `Boogie`[7], un lenguaje intermedio utilizado para la verificación formal de programas. Utilizando la herramienta de verificación `Corral`[8], se pueden analizar y asegurar matemáticamente las propiedades de los contratos inteligentes. Este método no solo mejora la seguridad de los contratos en Algorand, sino que también proporciona un marco robusto que puede ser adaptado para otras plataformas *blockchain*.

1.2. Metodología

El desarrollo de este enfoque incluye la implementación de un traductor de `TEAL` a `Boogie`, la definición de un conjunto de propiedades de seguridad a verificar, y la utilización de la herramienta `Corral` para la verificación formal.

2. PRELIMINARES

2.1. Introducción a Algorand

Algorand es una *blockchain* que usa como algoritmo de consenso un protocolo de acuerdo bizantino basado en prueba de apuesta[4].

La *Algorand Virtual Machine*, o AVM, es un intérprete de pila basado en bytecode que ejecuta programas asociados con transacciones de Algorand. Los programas se escriben en `TEAL` (*Transaction Execution Approval Language*), una sintaxis de lenguaje ensamblador para especificar un programa que termina convirtiéndose a bytes. Estos programas se llaman Contratos Inteligentes, o Aplicaciones.

Además de las Aplicaciones, existen Firmas Inteligentes, también referidas como *LogicSignatures* o *LogicSigs*, que aprueban transacciones basándose en un programa `TEAL` asociado. Las aprobaciones pueden ser parte de una cuenta de un usuario (Firma Delegada), o de la propia cuenta única asociada al programa de la firma (Cuentas Contrato). Las Cuentas Contratos son de particular interés ya que se verán en el caso de estudio.

En Algorand las transacciones se implementan como operaciones por lotes irreducibles, donde un grupo de transacciones se envía como una unidad, llamada *Atomic Transfer* o *Group Transaction*, y todas las transacciones del lote se aprueban o fallan. La figura 1 muestra como tres transacciones de diferentes cuentas se agrupan en un grupo para luego ser firmado por las cuentas involucradas. Existen varios tipos de transacciones en Algorand, esta tesis se centra únicamente en las de tipo *ApplicationCall*.

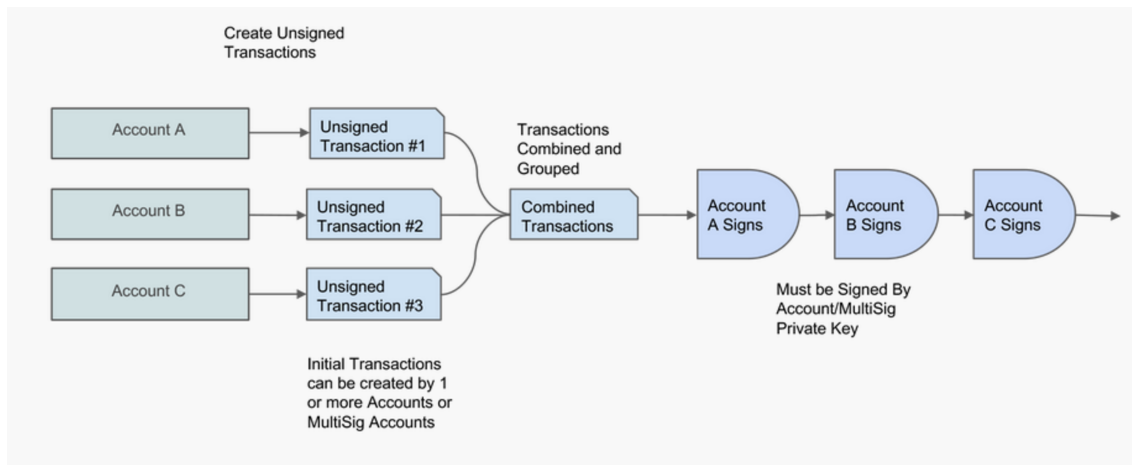

Fig. 1: Flujo de las transacciones

Las llamadas a contratos inteligentes utilizan transacciones de tipo *ApplicationCall*, que tienen diferentes subtipos. Cada subtipo de transacción sirve a un propósito específico en la interacción con contratos inteligentes, cuyos detalles se verán más adelante.

Los contratos inteligentes en Algorand tienen dos programas asociados: el *Approval-Program* y el *ClearStateProgram*. La ejecución de uno u otro depende del subtipo de la transacción de *ApplicationCall*.

Los contratos inteligentes en Algorand pueden requerir datos de la blockchain para su evaluación. Esto se provee con un conjunto de arreglos de referencia pasadas con cada transacción de tipo *ApplicationCall* que definen los elementos particulares de la blockchain que están disponibles durante la ejecución, como puede verse en la figura 2. Estos arreglos incluyen *Accounts*, *Assets*, *Applications* y *Boxes*, con un límite total de ocho valores combinados por transacción de tipo *ApplicationCall*.

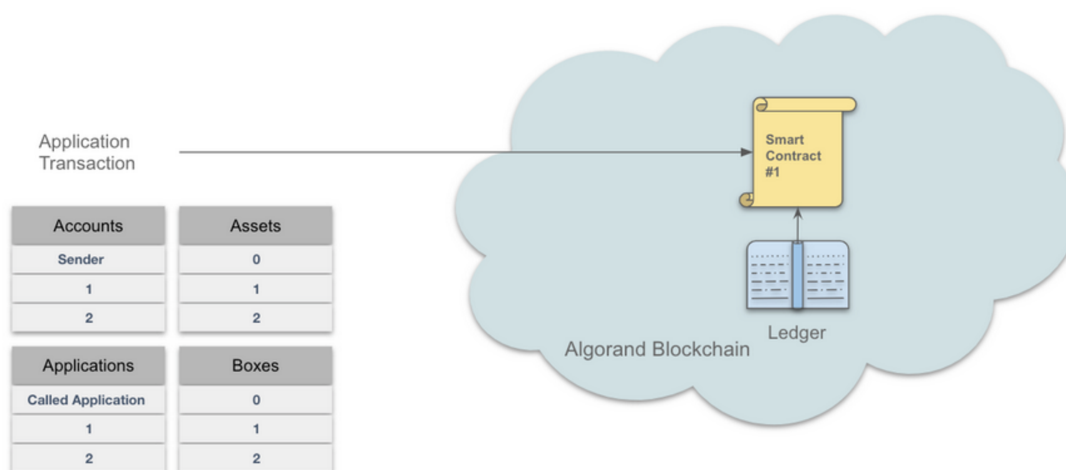

Fig. 2: Arreglos de referencia en una transacción de tipo ApplicationCall

Además pueden acceder a algunos valores globales de la transacción y otros globales de la blockchain. Ambos pueden verse en la figura 3 bajo las columnas de *Transaction*(s) y *Globals* respectivamente.

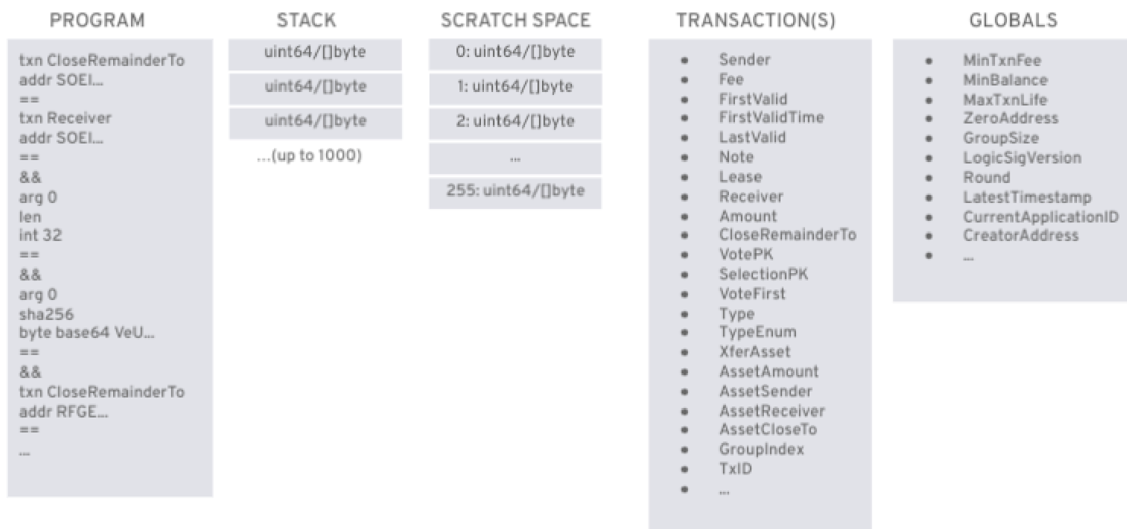

Fig. 3: Subconjunto de los elementos disponibles en un contrato

Los contratos inteligentes pueden mantener un estado global y uno local por cada cuenta que opte por participar en el contrato. como se puede ver en la figura 4 [9].

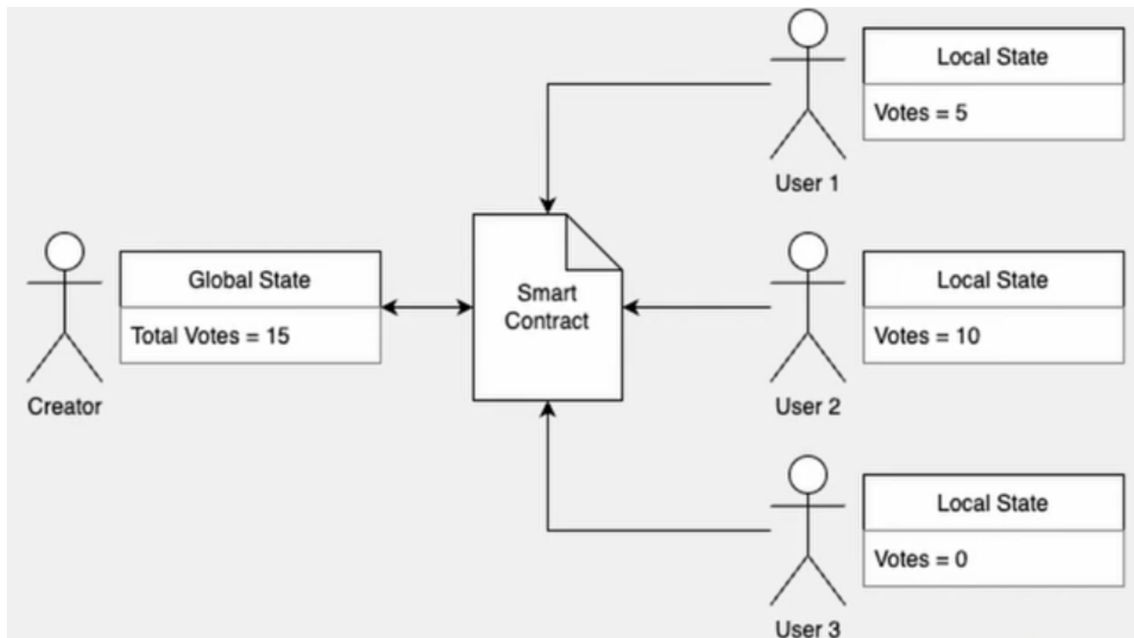

Fig. 4: Estado global y local de un contrato que representa un proceso de votación. En el estado global se mantiene el total de votos, y en el local, la cantidad de votos realizados por usuario.

El estado global puede ser accedido desde cualquier transacción, mientras que para acceder al estado local de una cuenta especifica, es necesario incluirla en el arreglo *Accounts* de la transacción.

Además de la pila y el almacenamiento, existen posiciones o *slots* de espacio temporal

llamados *Scratch* slots. Se accede a este espacio temporal mediante los códigos de operación de *store* y *load* que mueven datos desde o hacia el espacio temporal, respectivamente[19].

A continuación se muestra un ejemplo simple de la sintaxis de `TEAL`:

Listing 2.1: Ejemplo del ApprovalProgram de un contrato

```
1  int 12
2  dup
3  *
4  store 1
5  int 10
6  dup
7  *
8  load 1
9  +
10 int 244
11 ==
12 return
```

Cada linea en el programa representa una operador de la AVM. De manera coloquial, el contrato realiza la cuenta (12*12)+(10*10) y verifica que el resultado sea igual a 244, lo que siempre es cierto, por lo tanto, cualquier transacción que invoque al contrato siempre será aprobada.

En la figura 5 podemos ver cómo cambia el estado interno (de manera simplificada) del programa ejemplificado en el listado 2.1.

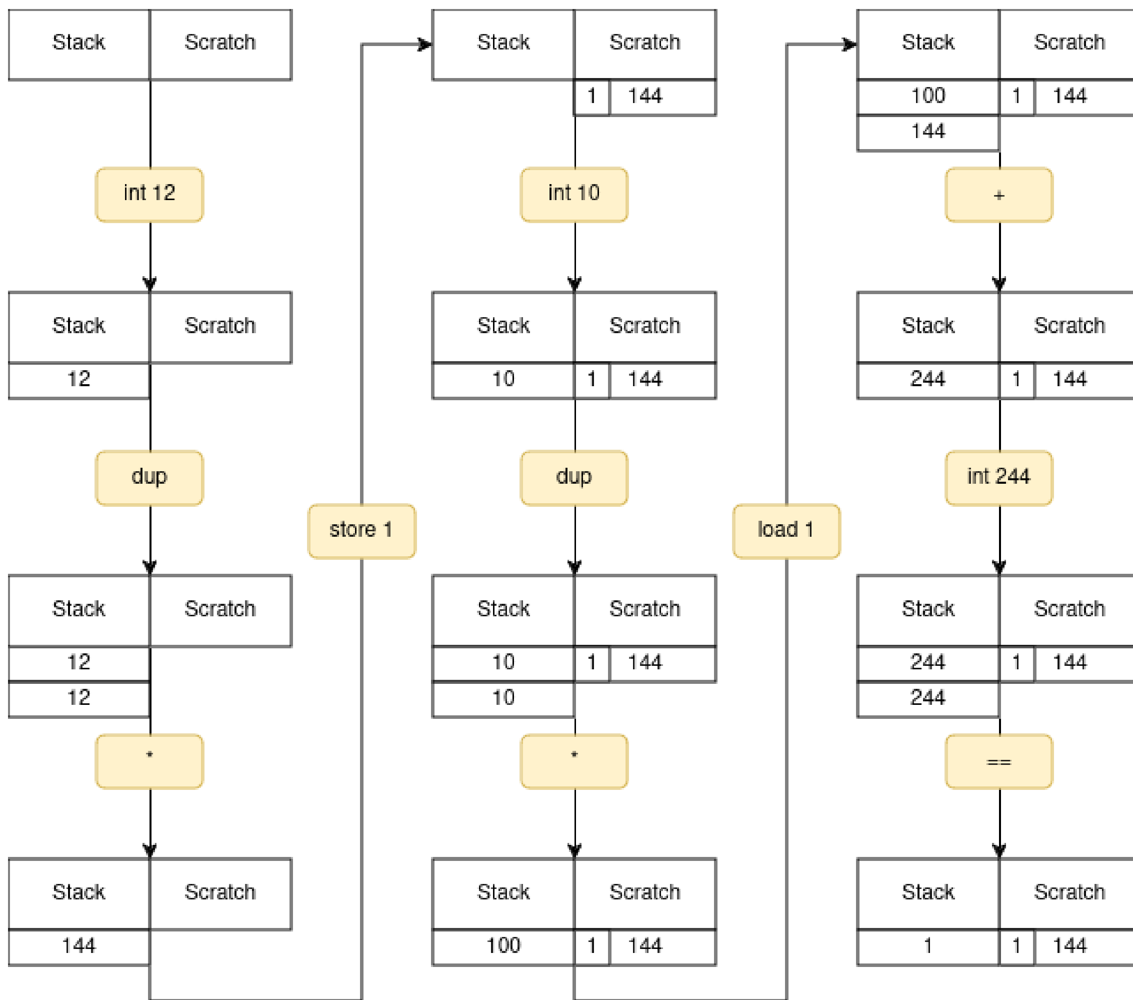

Fig. 5: Al inicio, la pila está vacía. La instrucción **int 12** apila el número 12 en el tope de la pila. Luego, **dup** duplica el valor en el tope de la pila, apilándolo dos veces. A continuación, la instrucción ***** consume los dos valores en el tope de la pila, los multiplica y apila el resultado. La instrucción **store 1** mueve el valor en el tope de la pila al registro Scratch slot número 1. Este proceso se repite con el número 10 utilizando **int 10** y **dup**. Luego, **load 1** apila el valor almacenado en el Scratch slot 1, y la instrucción + suma los dos valores en el tope de la pila y apila el resultado. Finalmente, == compara los dos últimos valores en la pila y apila un 1 si son iguales, o un 0 en caso contrario.

2.2. Introducción a Boogie

Una técnica estándar en la verificación de programas es transformar un programa dado, en un conjunto de condiciones de verificación, o VC por sus siglas en inglés. Estas son fórmulas lógicas cuya validez implica que el programa satisface las propiedades de correctitud consideradas. Las condiciones de verificación se procesan luego con la ayuda de un demostrador de teoremas, donde un intento de demostración exitoso muestra la correctitud del programa, y un intento fallido puede indicar un posible error en el programa.

Un verificador de programas se construye a partir de varias piezas complejas de tec-

nología: un lenguaje de programación fuente, sus reglas de uso y semántica formal, una codificación lógica adecuada para el razonamiento automático, dominios abstractos para el análisis de programas e inferencia de propiedades, procedimientos de decisión para cumplir con las obligaciones de prueba, y una interfaz de usuario que permita a un usuario comprender los resultados del proceso de verificación. Abordar estas complejidades, al igual que otros problemas de ingeniería de software, requiere una arquitectura modular con límites de interfaz bien establecidos.

Los autores de [20] sugieren que la compleja tarea de generar condiciones de verificación para lenguajes de programación modernos puede gestionarse en dos pasos: una transformación del programa y sus pre y postcondiciones en un lenguaje intermedio (una representación intermedia que se estructura más como un programa que como una fórmula lógica), y luego una transformación a fórmulas lógicas. El primero de estos pasos codifica la semántica de las partes del programa fuente en términos de primitivas, registra como suposiciones propiedades que están garantizadas en cualquier ejecución del programa fuente, y prescribe qué significa la correctitud (es decir, qué condiciones deben mantenerse para que el programa se considere correcto).

`Boogie` es un Lenguaje Intermedio de Verificación (IVL) y un motor de verificación correspondiente desarrollado para facilitar la verificación de programas de software. Sirve como una capa intermedia entre lenguajes de alto nivel y herramientas de verificación de bajo nivel, permitiendo la traducción de especificaciones de programas en fórmulas lógicas que pueden ser analizadas por demostradores automáticos de teoremas. `Boogie` está diseñado para ser independiente del lenguaje, lo que lo convierte en una herramienta versátil en el ámbito de la verificación formal.

Los programas `Boogie` se convierten en condiciones de lógica de primer orden, un proceso que requiere invariantes de ciclo. Aunque estos pueden provenir de los programas fuente, muchos invariantes de ciclo pueden ser "aburridos" u "obvios" para el programador, en cuyo caso la tarea de proporcionarlos manualmente es onerosa y tenerlos como parte del texto fuente proporciona más desorden que claridad. A veces, los invariantes de ciclo son incluso imposibles de expresar en el lenguaje fuente, como es el caso cuando el invariante necesita referirse a variables o funciones de la codificación `Boogie` del programa fuente. Por lo tanto, `Boogie` incluye un *framework* para interpretación abstracta, que puede inferir invariantes de ciclo del programa `Boogie`. Estos invariantes se insertan como declaraciones de suposición en las cabeceras de los ciclos del programa `Boogie`, de modo que puedan ser asumidos por la condición de verificación para mantenerse al inicio de cada iteración del ciclo.

Para combinar modularmente dominios abstractos y apoyar el uso de referencias de objetos que desreferencian el heap en el programa fuente, `Boogie` innova conectando sus dominios abstractos a un dominio abstracto especial que, esencialmente, nombra simbólicamente ubicaciones en el heap, nombres que luego son utilizados por los otros dominios abstractos[7].

El flujo de trabajo típico que involucra `Boogie` puede resumirse de la siguiente manera:

Traducción: Una herramienta traduce un programa escrito en un lenguaje fuente al lenguaje intermedio de `Boogie`. Esta traducción no solo incluye la generación de código `Boogie` para la lógica del programa, sino también la incrustación de las especificaciones,

las cuales son provistas por el usuario.

Programa Boogie: El programa Boogie resultante consiste en procedimientos con especificaciones anotadas, junto con las construcciones necesarias de flujo de control y manipulación de datos.

Condiciones de Verificación: Boogie traduce el programa Boogie anotado en condiciones de verificación, que son fórmulas lógicas que representan la correctitud del programa de acuerdo con las especificaciones.

Demostradores Automáticos de Teoremas: Las condiciones de verificación se envían a demostradores automáticos de teoremas, como Z3, que intentan probar o refutar la correctitud de las fórmulas.

Interpretación de Resultados: Los resultados del demostrador de teoremas se interpretan para determinar si el programa original cumple con sus especificaciones. Si una prueba falla, los contraejemplos proporcionados por el demostrador de teoremas pueden guiar la depuración y refinamiento del programa o sus especificaciones.

A continuación se muestra un ejemplo de la sintaxis de Boogie. En este, se verifica una variable sea mayor que 10.

Listing 2.2: Ejemplo de programa escrito en Boogie

```
1 procedure main();
2 implementation main() {
3     var x: int;
4     assume x > 10;
5     assert x > 10;
6     return
7 }
```

En el ejemplo del listado 2.2 se puede ver por línea:

1. Define el procedimiento **main**
2. Indica el comienzo de la implementación del proceso **main**
3. Declara una variable entera **x**.
4. El *keyword assume* realiza la suposición, o *assumption*, de que **x** es mayor que 10. Las suposiciones se utilizan para proporcionar información al motor de verificación.
5. Verifica que **x** sea mayor a 10. Las afirmaciones, o *assertions*, se utilizan para especificar propiedades que deben cumplirse en ese punto del programa. Si **x** pudiese tomar un valor menor o igual a 10, la afirmación fallaría.

2.3. Introducción a Corral

El problema de “Alcanzabilidad”, o *Reachability* en inglés, tiene origen en el estudio de máquinas de estado finito, y consiste en responder la siguiente pregunta: dado un grafo, un nodo origen y un nodo destino, ¿Existe un camino en el grafo de origen a destino? Debido a que una gran clase de programas puede representarse con esta abstracción, la pregunta

atrajo la atención de los investigadores dedicados a la verificación de software. Con los años el modelo original ha sido extendido para aumentar su expresividad con el objetivo de analizar clases más amplias de programas. Los autores de `Corral` [8] introducen un lenguaje de representación intermedia para la verificación. Éste tiene el objetivo de disminuir la brecha semántica entre los lenguajes de programación utilizados en la industria, como C, C# y Java, y el modelo utilizado para resolver efectivamente el problema de *Reachability*. El problema de *Reachability* en el lenguaje introducido por `Corral` es conocido como *Reachability Modulo Theories* o RMT.

`Corral` es un proyecto de *Microsoft Research* que resuelve *queries* de “Reachability” en modelos escritos en Boogie. Para utilizarlo se traduce un programa a este lenguaje, y luego se plantea la “Reachability” como una función con una o más aserciones. Luego el motor buscará un contraejemplo que las viole.

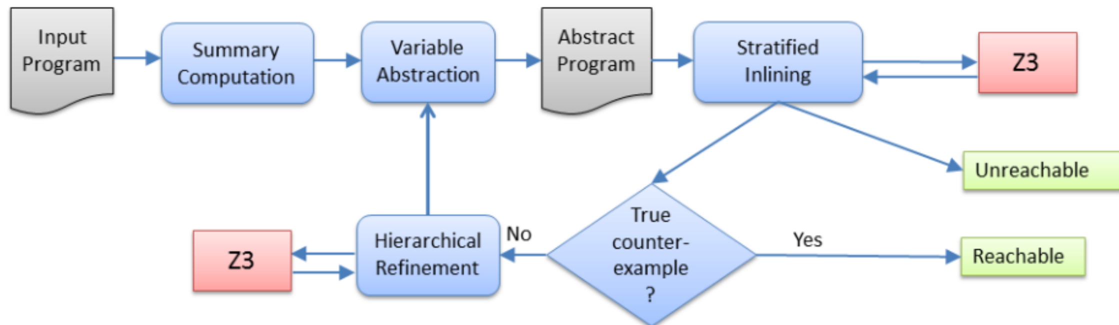

Fig. 6: Arquitectura de `Corral`

Boogie viene equipado con algoritmos de generación de condiciones de verificación que transforma código con aserciones a formulas de *Satisfability Modulo Theories* o SMT. Como se muestra en la figura 6, `Corral` hace uso del *solver* de SMT Z3, convirtiéndolos de “Reachability queries”, a condiciones de verificación en SMT.

La ejecución de Corral se centra en un ciclo de dos niveles de refinamiento de soluciones guiado por contraejemplos, llamado CEGAR *loop*, por las siglas en inglés de *CounterExample Guided Abstraction Refinement*. El primer nivel del ciclo se encarga de abstraer al *solver* de las variables globales del programa. En la práctica no suele ser necesario tenerlas en cuenta para verificar una parte del mismo, e incluir las mismas es muy costoso en la VC. Por lo tanto, inicialmente se ignoran todas las variables globales y se pasa al siguiente nivel. Este nivel, llamado *Stratified Inlining* es el encargado de los llamados a funciones en el programa a analizar. Como hacer *inlining* estático de todos los llamados es muy costoso, este módulo utiliza sub y sobre aproximaciones de los mismos para decidir donde hacer, o no, *inlining* en cada iteración. Si se decide no hacer *inlining* se utiliza la construcción “*assume false*” de Boogie que recorta toda posible traza de modo tal que solo se analice hasta ese punto. En caso contrario se utiliza una abstracción de la función. Haciendo uso de estos dos niveles, la resolución de una *query* P comienza con ésta y un conjunto de todos los *call-sites* de la misma, C. En todo momento, se mantiene un programa P parcialmente *inlined*, junto con un conjunto de *call-sites* C en P que todavía no han sido *inlined*.

Cada iteración se compone de dos etapas. En la primera, cada *callsite* abierto en P

es reemplazado por su sub-aproximación para así obtener un programa cerrado *P'*, que es chequeado usando Z3. Lo que se intenta en este caso es determinar la satisfacibilidad de las aserciones en el programa. Si es posible alguna, el algoritmo concluye que hay un *bug*. En caso contrario se procede a la segunda etapa. En esta se recurre a la sobre-aproximación, mediante el uso de los resúmenes, o *summaries*. Cada *call-site* es reemplazado con el *summary* del método al que hace referencia. Si el programa resultante es correcto, debido a que todas las llamadas fueron sobre-aproximadas, se puede concluir que el programa original P es correcto. Si no se pudiese concluir en la etapa anterior que el programa es correcto, entonces se obtiene una prueba de ello que involucra a algunos *call-sites*. Como consecuencia, dichas llamadas son *inlined* en P y se vuelve a comenzar.

Es importante aclarar que en una ejecución de `Corral` el usuario acota el espacio de búsqueda limitando la cantidad de veces que se hace *inlining* en un llamado recursivo. Esto lleva a que existen tres posibles respuestas arrojadas al intentar resolver una *query.True Bug* en caso de encontrar un *bug*, *No Bug* en caso contrario, o *Maybe Bug* cuando se alcanza la cota de *inlining*[16].

2.3.1. Ejemplo de `Corral`

El siguiente programa escrito en Boogie implementa un procedimiento defectuoso:

```
1  procedure faulty(x: int);
2  implementation faulty(x: int){
3  assert x > 0;
4  }
```

A simple vista, es fácil ver que la aserción fallará con cualquier entero menor o igual a cero. Se muestra a continuación el output de usar `Corral` para verificar este programa:

```
1   Verifying program while tracking: {assertsPassed}
2   Program has a potential bug: True bug
3   PersistentProgram(9,1): error PF5001: This assertion can fail
4
5   faulty.bpl(3,1): error PF5001: This assertion can fail
6
7   Execution trace:
8   Format: (tid,k)  filename(line,col): blockName   (extra info)
9   (1,0)    faulty.bpl(3,1): anon0
10  (1,0)    faulty.bpl(3,1): anon0  (ASSERTION FAILS assert x > 0;
11   )
12  (1,0)    faulty.bpl(3,1): anon0  (Done)
13
14  Boogie verification time: 0.08 s
15  Time spent reading-writing programs: 0.01 s
16
17  Time spent checking a program (1): 0.21 s
18  Time spent checking a path (1): 0.02 s
19
20  Number of procedures inlined: 1
21  Number of variables tracked: 1
22  Total Time: 0.3354716 s
23  Total User CPU time: 0.32 s
```

La herramienta verifica el programa mientras rastrea específicamente que las aserciones no fallen. `Corral` ha detectado un posible error en el programa, indicando un *True Bug*, lo que confirma que se trata de un problema real y no de un falso positivo. El mensaje

de error `PersistentProgram(9,1): error PF5001: This assertion can fail` señala la ubicación específica del error. El código PF5001 es un código de error, y el mensaje implica que la herramienta ha identificado una aserción en el código que podría fallar potencialmente bajo ciertas condiciones. La notación `faulty.bpl(3,1)` especifica que el problema ocurre en la línea 3, columna 1 del archivo faulty.bpl. El seguimiento de ejecución proporciona un resumen paso a paso de cómo se ejecutó el programa hasta el punto en el que falló la aserción. La línea `(1, 0) faulty.bpl(3, 1) : anon0(ASSERTION FAILS assert x > 0;)` muestra que, mientras se ejecutaba el código en faulty.bpl en la línea 3, el programa encontró una aserción fallida. La condición fallida es `assert x > 0;`, lo que indica que la variable x no era mayor que 0 en ese punto de la ejecución. La línea `(1, 0) faulty.bpl(3, 1) : anon0(Done)` indica que la ejecución del bloque se completó después de encontrar la aserción fallida. Luego `Corral` provee distintas métricas de performance al final de su output.

3. VERITEAL

Para desarrollar un verificador formal para Algorand, se tomó de ejemplo `VeriSol`[10], una herramienta de verificación formal para contratos de Ethereum. Esta herramienta traduce contratos inteligentes escritos en *Solidity* a programas escritos en `Boogie`. Luego, este programa puede ser analizado por verificadores como `Z3` y `Corral`.

De esta idea surge `VeriTEAL`, como `VeriSol`, traduce contratos escritos en `TEAL` a `Boogie` y utiliza `Corral` para verificar el contrato.

El proceso de verificación de `VeriTeal` está compuesto por cuatro etapas principales: La traducción de `TEAL` a `Boogie`, donde se utiliza la herramienta `Tealift` para pasar del modelo de ejecución basado en una pila, a uno basado en registros. La generación de la interfaz del contrato, que define cómo interactuar con el mismo. La definición de pre y post condiciones, las cuales son esenciales para especificar el comportamiento esperado del contrato antes y después de su ejecución. Y finalmente, la ejecución de `Corral` permite analizar el contrato y verificar que cumpla con las condiciones establecidas, asegurando de esta manera su correctitud.

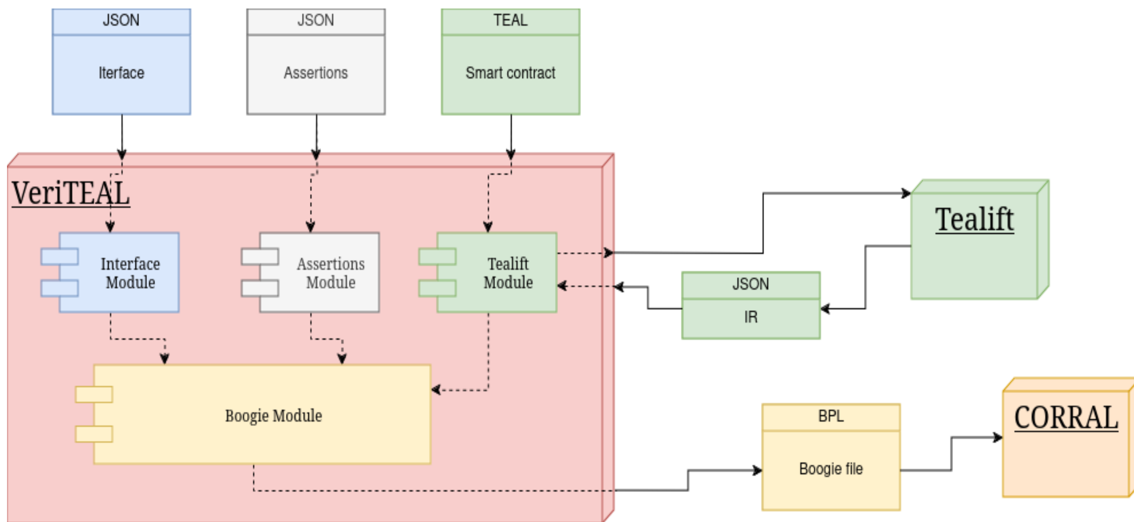

Fig. 7: Arquitectura de `VeriTEAL`

3.1. Traducción de `TEAL` a `Boogie`

El primer enfoque adoptado fue crear una implementación de la AVM en `Boogie`. Principalmente, modelar la ejecución de los contratos con una pila. Luego, para cada operador de la AVM, puede implementarse una función equivalente en `Boogie`. Esto permite una traducción simple y directa de `TEAL` a `Boogie`.

Para este enfoque, el puntero al tope de la pila se representó con un entero global y

la pila con un mapa global. De esta manera, los distintos operadores de la AVM pueden incrementar y decrementar el puntero, obtener valores y guardar resultados en los lugares correspondientes. A su vez, el almacenamiento global, local y las transacciones de un grupo, también fueron representados con mapas.

Para cada operación AVM, se creó una función un equivalente en `Boogie`. Esto permitió una traducción simple y directa de `TEAL` a `Boogie`. Como podemos ver en el listado 3.1, la traducción a `Boogie` genera código muy similar al ejemplo del listado 2.1.

Listing 3.1: Representación en Boogie utilizando una pila.

```
1  procedure contract();
2  implementation contract(){
3    call Int(12);
4    call Dup();
5    call Multiply();
6    call Store(1);
7    call Int(10);
8    call Dup();
9    call Multiply();
10   call Load(1);
11   call Sum();
12   call Int(244);
13   call Equal();
14 }
```

Por otra parte, este enfoque resultó ser demasiado costoso desde un punto de vista computacional. Las pruebas preliminares mostraron que seguimiento de variables de `Corral` para contratos simples podía demorarse varias horas y el tiempo de ejecución aumentaba junto con la complejidad de los contratos inteligentes. Esto condujo a la búsqueda de alternativas más eficientes.

El enfoque principal tomado fue reemplazar la pila por registros. Para lograr esto, se utilizó la herramienta Tealift.

Listing 3.2: Representación Boogie utilizando registros.

```
1  procedure contract();
2  implementation contract(){
3    var x: int;
4    var y: int;
5    var z: int;
6    var scratch_slot_1: int;
7    x := 12;
8    y := x;
9    z := x * y;
10   scratch_slot_1 := z;
11   x := 10;
12   y := x;
13   z := x * y;
14   x := scratch_slot_1;
15   y := x + z;
16   x := 244;
17   z := to_int(x == y);
18 }
```

En el listado 3.2 podemos ver una posible traducción del ejemplo del listado 2.1, en donde el proceso de apilado y desapilado, es reemplazados por asignación de variables. Las sentencias apilado de enteros, como `int 12`, es representada por $x := 12$. Como puede verse, los *Scratch slots* también son modelados con variables. Un detalle es que `Boogie`

requiere declarar las variables, y su tipo, al principio de la implementación. Para simplificar la generación de código, las operaciones que retornan booleanos son envueltas en la función to_int que retorna 1 si recibe *True* y 0 si recibe *False*, como puede verse en la linea 17.

3.1.1. Tealift

Tealift es una herramienta actualmente en desarrollo por CoinFabrik que transforma código TEAL a un lenguaje de representación intermedia [1]. Este lenguaje utiliza registros (o variables) para representar el flujo de datos, y en particular, cumple la propiedad de estar en forma Single Static Assigment (SSA). Esto significa que a cada variable se le asigna un único valor antes de ser usada.

Convertir código a forma SSA consiste principalmente en reemplazar el lado izquierdo de cada asignación con una nueva variable y reemplazar cada uso de una variable con la "versión" de esa variable que llega a ese punto. Por ejemplo, dado el grafo de flujo de control de la figura 8.

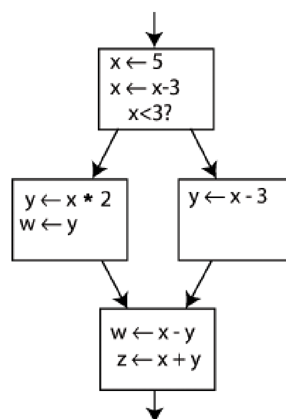

Fig. 8: Grafo de control de flujo

Si se cambia el nombre en el lado izquierdo de $x \leftarrow x - 3$ y actualizamos los usos posteriores de x a ese nuevo nombre, el comportamiento del programa se mantiene. En SSA, esto se aprovecha creando dos nuevas variables: $x1$ y $x2$, cada una de las cuales se asigna solo una vez. De manera similar, al añadir subíndices distintivos a todas las demás variables obtenemos el grafo de la figura 9.

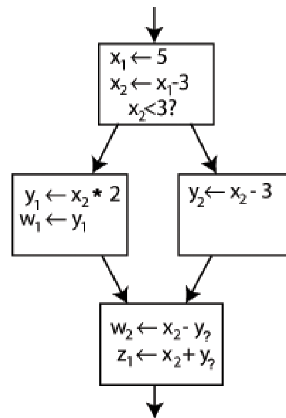

Fig. 9: Grafo de control de flujo, parcialmente convertido a SSA.

Es claro a qué definición se refiere cada uso, excepto en un caso: ambos usos de y en el bloque inferior podrían referirse a $y1$ o $y2$, dependiendo del camino que haya tomado el flujo de control.

Para resolver esto, se inserta una función especial en el último bloque, llamada ϕ (Phi). Esta función generará una nueva definición de y, llamada $y3$, eligiendo entre $y1$ o $y2$, según el flujo de control anterior.

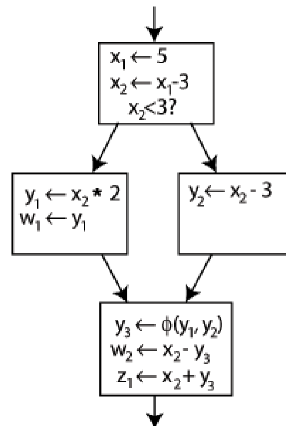

Fig. 10: Grafo de control de flujo, completamente convertido a SSA

Ahora, el último bloque puede usar $y3$, y se obtendrá el valor correcto en cualquier caso. No se necesita una función ϕ para x, ya que solo una versión de x, concretamente $x2$, llega a este punto, por lo que no hay problema (en otras palabras, $\phi(x2, x2) = x2$) [21].

Tealift genera el grafo de flujo de datos en el siguiente formato JSON:

```
1 {
2   entrypoint: <idx_into_bbs>
3   basic_blocks: [
4     {
5       incoming_edges: [<idx_into_bbs>],
```

```
6        outgoing_edges: [<idx_into_bbs>],
7        phis: [[<idx_into_bb_instructions>]],
8        instructions: [
9          {
10           op: <operation_name>,
11           args: [<operation_arguments>],
12           consumes: [<idx_into_bb_instructions> | <negative_idx_into_phis>]
13         }
14       ],
15       terminal: {
16         op: <terminal_name>,
17         args: [<operation_arguments>],
18         consumes: [<idx_into_bb_instructions> | <negative_idx_into_phis>],
19       }
20     }
21   ]
22 }
```

El valor en $phis[i][j]$ es el i-esimo phi viniendo del bloque en $incoming_edges[j]$ y los índices del arreglo $phis$ se codifican como $-i-1$.
VeriTEAL utiliza este grafo para reconstruir el contrato en código Boogie.

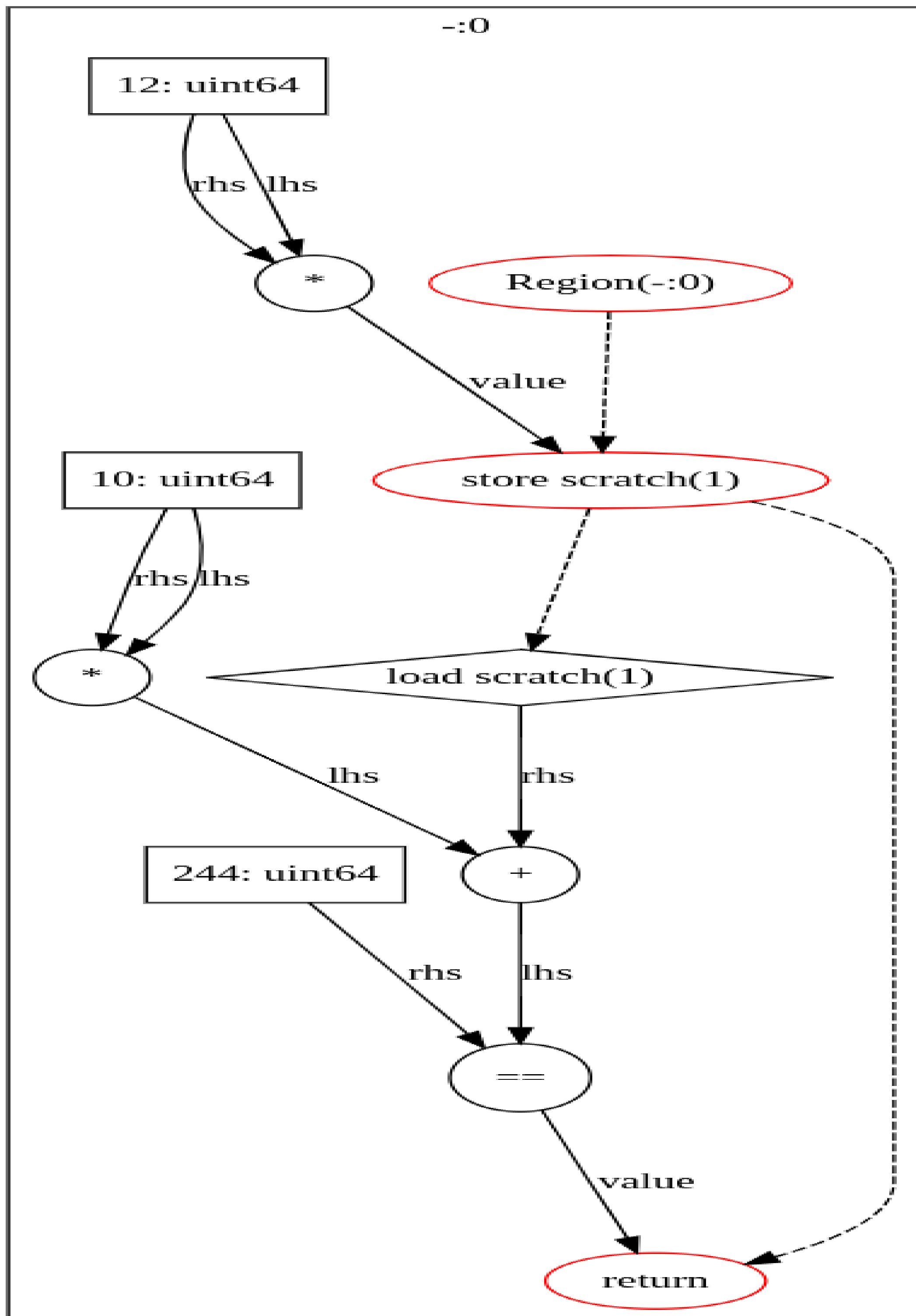

Fig. 11: Grafo de flujo de datos generado por Tealift generado en base al ejemplo del listado 2.1.

Se detalla el algoritmo de reconstrucción en el siguiente listado:

Algorithm 1 Contract Procedure

```
 1: procedure CONTRACT_PROCEDURE(tealift: Tealift)
 2:     var_names ← {}
 3:     consumed_phis_consumer ← tealift.consumed_phis_consumer()
 4:     procedure_statements ← []
 5:     for block_index, block ∈ tealift.basic_blocks do
 6:         procedure_statements.append(label_declaration_statement(block_index))
 7:         for instruction_index, instruction ∈ block.instructions do
 8:             parsed_instruction ← parse_instruction(
                  instruction,
                  instruction_index,
                  block,
                  block_index
                )
 9:             instruction_statement ← parsed_instruction.to_boogie()
10:             if parsed_instruction.returns_value() then
11:                 var_names.add(parsed_instruction.returned_variable_name)
12:             end if
13:             procedure_statements.append(instruction_statement)
14:             if (block_index, instruction_index) ∈ phi_values then
15:                 values ← phi_values[(block_index, instruction_index)]
16:                 for value ∈ values do
17:                     phi_block_index, phi_instruction_index, phi_consumes_index ← value
18:                     phi_var_name ← phi_variable_name(
                          phi_block_index,
                          phi_instruction_index,
                          phi_consumes_index
                        )
19:                     var_names.add(phi_var_name)
20:                     procedure_statements.append(
                          variable_assigment(
                            phi_var_name,
                            parsed_instruction.returned_variable_name
                          )
                        )
21:                 end for
22:             end if
23:         end for
24:     end for
25:     procedure_variables_declarations ← []
26:     for var_name in var_names do
27:         procedure_variables_declarations.append(variable_declaration_statement(var_name))
28:     end for
29:     full_procedure ← procedure_variables_declarations + procedure_statements
30:     return full_procedure
31: end procedure
```

La primera linea declara el procedimiento. Esta toma un objeto Tealift que representa el JSON provisto por la herramienta Tealift. Este objeto facilita el manejo de la información y provee funciones de utilidad. En la línea número 2 se definen *var_names* como un conjunto en donde se guardaran los nombres de las variables del procedimiento. Como ya ha sido mencionado, las variables en Boogie deben ser declaradas al principio del procedimiento, por lo tanto, a medida que se procesan las instrucciones, si retornan una variable, se guarda en el conjunto (líneas 10-11) y luego se declara anteriormente al resto de las instrucciones (líneas 25-28). En la línea número 3 se define el mapa *consumed_phi_consumer*. Las claves de este mapeo tienen la forma $(block_index, instruction_index)$, y son las instrucciones que son consumidas por algún Phi, y los valores del mapeo son los Phi que consumen la instrucción de la clave. Más adelanté se detalla cómo se calcula el mapeo. En la línea 4 se define el arreglo de sentencias, estas representan el groso de la traducción del contrato inteligente. Luego, para cada bloque se declara una etiqueta (línea 6). Cada bloque representa una rama del árbol de control de flujo y las etiquetas permiten accederlas mediante saltos. Desde la linea 7 la 13, se procesa cada instrucción de cada bloque, transformándola en un objeto que maneja el generado de la sentencia de código Boogie y luego, se la agrega a la lista de sentencias. Los Phi se modelaron como variables identificadas por el bloque donde se consume, el índice de instrucción que la consume (por bloque) y la posición en los argumentos de la instrucción que la consume. Por ejemplo:

```
1  ...
2  block_i:
3    x = 1
4  ...
5  block_j:
6    x = 2
7  ...
8  block_k:
9    a = foo()
10   b = foo2()
11   bar(a, phi(x), b)
```

Asumiendo que los caminos de los bloques i y j ambos convergen en el bloque, k obtendríamos:

```
1  ...
2  block_i:
3    x = 1
4    phi_k_2_1 = 1
5  ...
6  block_j:
7    x = 2
8    phi_k_2_1 = 2
9  ...
10 block_k:
11   a = foo()
12   b = foo2()
13   bar(a, phi_k_2_1 , b)
14 ...
```

En las líneas 14 a la 24, sucede este proceso. Si la instrucción es usada en algún Phi, se agrega una sentencia de asignación de variable en donde el valor asignado es el valor retornado para la instrucción y la variable asignada es la Phi. Luego el procedimiento completo está compuesto por la declaración de las variables como preámbulo y el resto de la instrucciones (lineas 25-30).

El mapa *consumed_phi_consumer* se calcula de la siguiente manera: De forma resu-

Algorithm 2 Consumed Phi Values to Phi Consumer

```
1: procedure CONSUMED_PHI_VALUES_TO_PHI_CONSUMER(self: Tealift)
2:     phis_map ← {}
3:     for basic_block_index, basic_block ∈ self.basic_blocks do
4:         for instruction_index, instruction ∈ basic_block.instructions do
5:             for consumes_index, consumes ∈ instruction.consumes do
6:                 if consumes < 0 then                          ▷ Consumes a phi value
7:                     values ← []
8:                     for incoming_edge ∈ basic_block.incoming_edges do
9:                         for consumed_instruction ∈ basic_block.phis[abs(consumes)-1] do
10:                            values ← phis_map[(incoming_edge, consumed_instruction)]
11:                            phi ← (basic_block_index, instruction_index, consumes_index)
12:                            if phi ∉ values then
13:                                values.append(phi)
14:                            end if
15:                            phis_map[(incoming_edge, consumed_instruction)] ← values
16:                        end for
17:                    end for
18:                end if
19:            end for
20:        end for
21:    end for
22:    return phis_map
23: end procedure
```

mida, si una instrucción de un bloque consume un Phi (línea 6), se definen como clave los valores que consume, y como valor, al consumidor (linea 15).

3.1.2. Comparación de métodos

Como comparativa inicial, se tomaron ambas traducciones (listado 3.1 y 3.2) del contrato ejemplo (listado 2.1) y se midió el tiempo de ejecución de ambas. Adicionalmente, se generaron versiones extendidas de las traducciones repitiendo el código original para ver el comportamiento de programas más extensos.

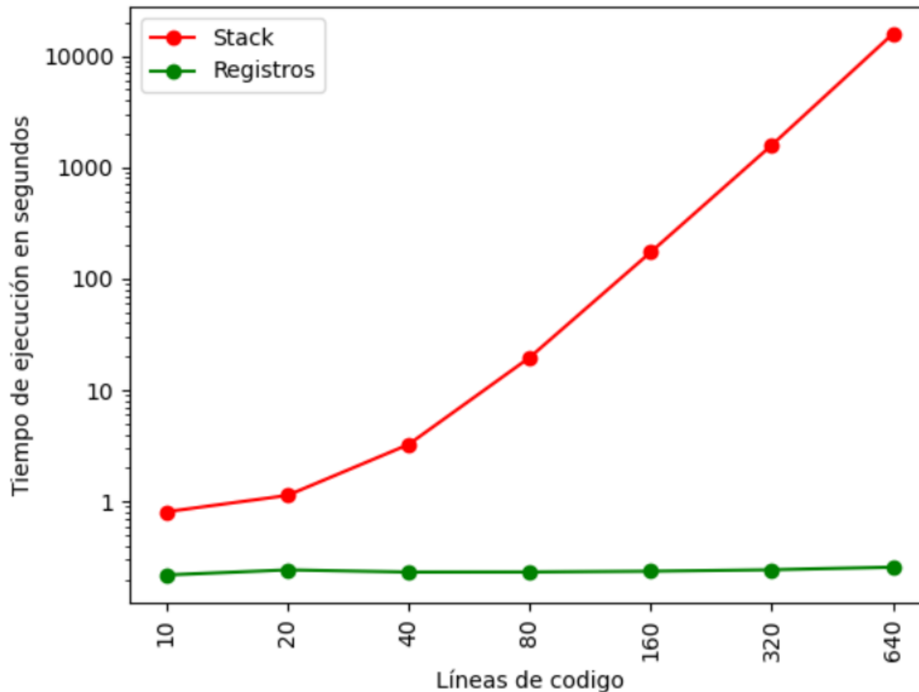

Fig. 12: Comparación de tiempos de ejecución para los listados 3.1 y 3.2

En la figura 12, a simple vista se puede apreciar cómo el tiempo de ejecución de las traducciones basadas en la máquina de pila crece exponencialmente junto con la cantidad de líneas de código, mientras que en las versiones con registros, el tiempo se mantiene constante.

Como métrica adicional, se tomó el tiempo de ejecución de encontrar errores dentro de un ciclo. El siguiente contrato muestra una manera simple de implementar un ciclo y chequear una condición:

```
1  int 0
2  check_equals:
3    dup
4    int N
5    ==
6    bz add_1
7    return
8  add_1:
9    int 1
10   +
11   b check
```

La etiqueta *check_equals*, en la línea 2, compara el tope de la pila con un entero N y si son iguales termina. En caso contrario, salta a la etiqueta *add_1*, que como su nombre indica, suma 1 al tope de la pila y salta incondicionalmente a la etiqueta *check_equals*.

Los listados 3.3 y 3.4 muestran las traducciones a Boogie utilizando los modelos de que utilizan la pila y registros respectivamente. Además, en la línea 12 de la figura 3.3 se agregó la condición que el tope de la pila sea menor que N y en la línea 22 de la figura 3.4 la condición de que la variable contadora sea menor estrictamente que N.

Listing 3.3: Traducción a Boogie utilizando una pila

```
procedure contract();
implementation contract() {
  call Int(0);
  check_equals:
    call Dup();
    call Int(N);
    call Equal();
    if (Stack[StackPointer] == 0) {
      call Pop();
      goto add_1;
    }
    assert Stack[StackPointer] < N;
    call Pop();
    return;
  add_1:
    call Int(1);
    call Sum();
    goto check_equals;
}
```

Listing 3.4: Traducción a Boogie utilizando registros

```
var N: int;

procedure contract();
implementation contract(){
  var a: int;
  var b: int;
  var c: int;
  var d: int;
  var e: int;
  a := 0;
  check:
    b := a;
    c := arg;
    d := to_int(b == c);
    if (d == 0) {
        goto add_1;
    }
    assert a < N;
    return;
  add_1:
    e := 1;
    a := a + e;
    goto check;
}
```

La figura 13 muestra la comparación del tiempo de ejecución para distintos valores de N, donde N se define como $N = 2^k$. Se puede apreciar a simple vista que para valores relativamente pequeños, como 128 iteraciones, el modelo que utiliza la pila toma varias horas para encontrar la contradicción, mientras que el modelo que utiliza registros se mantiene en el orden de la decena de segundos.

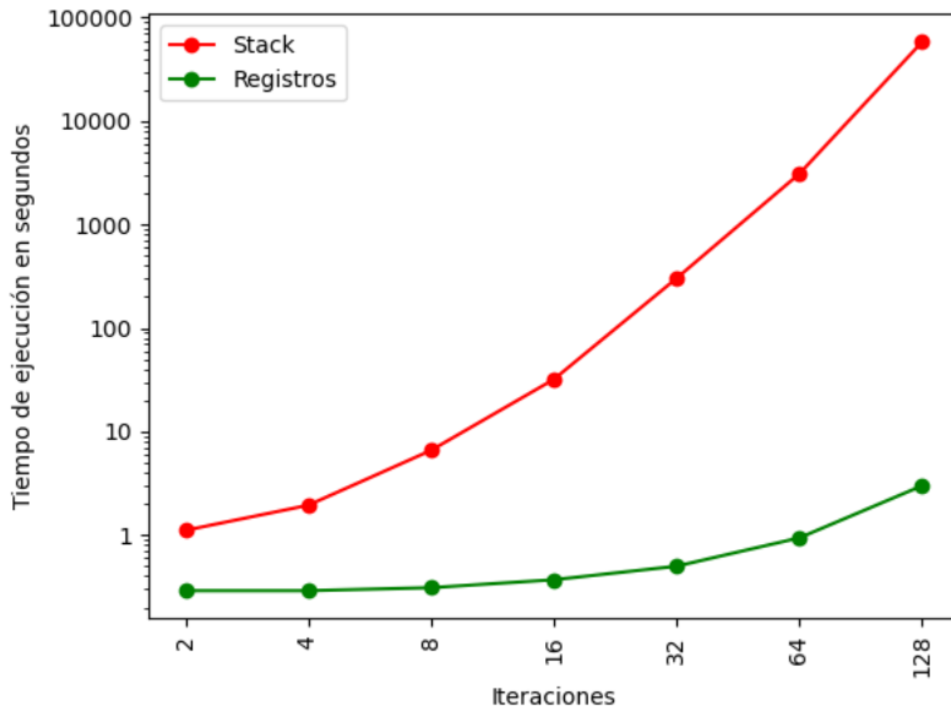

Fig. 13: Comparación de tiempos de ejecución para los listados 3.3 y 3.4

En conclusión, la traducción con uso de registros ofrece un mejor desempeño en términos de tiempo de ejecución.

3.1.3. Representación del estado de la AVM basada en registros

Otro beneficio de esta representación es que permite acceder a las variables de los contratos y sus estructuras de datos directamente. Al estar limitada la longitud de los arreglos, cada elemento puede ser representado con una variable explícitamente declarada.

```
1 var arg_0 : int;
2 ...
3 var arg_15 : int;
```

3.2. Generación de la interfaz del contrato

Para la verificación se implementó una función de *harness* basado en VeriSol[10]. Esta función se encarga de llamar al constructor del contrato e invocar los métodos del contrato repetidamente de manera no-determinística.

El problema con esta idea proviene de que `TEAL`, al ser un lenguaje ensamblador, no provee información de cómo invocar a los métodos del contrato, ni de sus parámetros. Para solucionar esto, nos basamos en los *Algorand Requests for Comments* (ARC), en particular ARC-04 [5].

Listing 3.5: Ejemplo del JSON de condiciones

```
procedure harness();
implementation harness(){
  while (true) {
      if (*) {
          assume pre_conditions_for_method_1;
          call method_1();
          assert post_conditions_for_method_1;
      }
      else if (*) {
          assume pre_conditions_for_method_2;
          call method_2();
          assert post_conditions_for_method_2;
      }
      ...
      else if (*) {
          assume pre_conditions_for_method_N;
          call method_N();
          assert post_conditions_for_method_N;
      }
  }
}
```

El ARC-04 define la convención para codificar llamadas a métodos, junto con sus argumentos y sus valores de retorno en transacciones de tipo ApplicationCall. El documento proporciona estructuras JSON para describir métodos, interfaces y contratos. Estas estructuras ayudan a los desarrolladores a definir y documentar la estructura de sus contratos. Para `VeriTEAL`, estos JSON son fundamentales para poder generar la función de harness.

Listing 3.6: Especificación ARC-04 en formato JSON de un contrato con dos métodos: `add` y `multiply`.

```
{
  "name": "Calculadora",
  "desc": "Contrato de una calculadora basica que admite
    sumas y multiplicaciones",
  "methods": [
    {
      "name": "add",
      "desc": "Calcular la suma de dos enteros de 64 bits",
      "args": [
        { "type": "uint64", "name": "a", "desc": "El primer
            termino a sumar" },
        { "type": "uint64", "name": "b", "desc": "El segundo
        termino a sumar" }
      ],
      "returns": { "type": "uint128", "desc":
        "La suma de a y b" }
    },
    {
      "name": "multiply",
      "desc": "Calcular el producto
        de dos enteros de 64 bits",
      "args": [
        { "type": "uint64", "name": "a", "desc": "El primer
            factor a multiplicar" },
        { "type": "uint64", "name": "b", "desc": "El segundo
            factor a multiplicar" }
      ],
      "returns": { "type": "uint128", "desc":
        "El producto de a y b" }
    }
  ]
```

```
32 }
```

Utilizando la interfaz en el listado 3.6, puede generarse la función de harness del listado 3.7 junto con las invocaciones de los métodos.

Listing 3.7: Pseudo implementación de la función de harness el ejemplo 3.6

```
procedure harness();
implementation harness(){
  call constructor();
  while (true) {
      if (*) call add();
      else if (*) call multiply();
  }
}

procedure add();
implementation add(){
    // Calcula el selector y lo asigna a la posicion esperada.
    arg_0 := method_signature("add(uint64,uint64)uint128");
    // Evalua las posiciones del arreglo de argumentos
    // en numeros enteros aleatoreos
    arg_1 := random_int_1;
    arg_2 := random_int_2;
    // Evalua el campo OnComplete en un entere del 0 al 5
    on_complete := random_on_complete_action
    call contract();
}
```

Por convención, la primera posición del arreglo de argumentos se utiliza para el selector de la función a invocar. El selector se define como los primeros 4 bytes del hash SHA-512/256 de la firma del método, que, a su vez, la firma se define como el nombre del método, un paréntesis de apertura, una lista separada por comas de los tipos de sus argumentos, un paréntesis de cierre y el tipo de retorno del método, o *void* si no devuelve un valor.

Adicionalmente, es necesario evaluar la variable global `on_complete` para invocar a la función con distintos tipos de transacciones. Cada uno de los distintos valores posibles, tienen una función específica para la AVM. A continuación se listan los valores posibles, junto con su comportamiento asociado. acceder

0. *NoOp*. Ejecuta el *ApprovalProgram* la aplicación. No tiene efectos adicionales.
1. *OptIn*. Asigna un estado local para la aplicación en los datos de la cuenta del remitente. Ejecuta el *ApprovalProgram* después de la asignación. En Algorand, cada cuenta tiene un requisito de saldo mínimo, que aumenta a medida que la cuenta opta por más activos o aplicaciones. El saldo mínimo está diseñado para prevenir el spam y asegurar que la blockchain se mantenga eficiente.
2. *CloseOut*. Borra cualquier estado local de la aplicación de los datos de la cuenta del remitente. Ejecuta el *ApprovalProgram* antes de borrar. Esta operación puede ser rechazada por el *ApprovalProgram*
3. *ClearState*. Ejecuta el *ClearStateProgram*. Aunque el programa rechace la transacción, ésta siempre será aprobada permitiendo a la cuenta decrementar su saldo mínimo requerido. Borra cualquier estado local de la aplicación de los datos de la cuenta

del remitente, similar a *CloseOut*. El propósito del programa de *ClearState* es permitir que la aplicación maneje la eliminación de ese estado local de manera adecuada.

4. *UpdateApplication*. Ejecuta el *ApprovalProgram*. Reemplaza el *ApprovalProgram* y *ClearStateProgram* asociados con el ID de la aplicación con los programas especificados en esta transacción.

5. *DeleteApplication*. Ejecuta el *ApprovalProgram*. Borra los parámetros de la aplicación de los datos de la cuenta del creador de la aplicación después de la ejecución.

La validación de este campo es fundamental para la integridad del contrato. Por ejemplo, un atacante podría enviar una transacción de *DeleteApplication* al contrato para eliminarlo.

Actualmente, frameworks de desarrollo de contratos inteligentes para Algorand, como `Beaker` [6], generan automáticamente archivos JSON que obedecen la convención definida en ARC-04, por lo cual estos contratos pueden ser analizados sin necesidad de trabajo manual. Para poder analizar contratos previos a la existencia de la convención de llamadas a métodos o que directamente no la respetan, se definió un esquema JSON que no impone restricciones para las llamadas al contrato. En este JSON se pueden definir los argumentos necesarios para la invocación de un método y los valores libres. Tomando como ejemplo la interfaz del listado 3.6, en el listado 3.2 puede verse como se representaría en el esquema JSON libre de restricciones.

```
1  {
2    "methods": [
3      {
4        "name": "add",
5        "required": {
6          "arguments": {"0": "\\xfek\\xdfi"},
7          "accounts": {},
8          "applications": {},
9          "assets": {}
10       },
11       "reserved": {
12         "arguments": [1,2],
13         "accounts": [],
14         "applications": [],
15         "assets": []
16       }
17     }
18   ]
19 }
```

En la clave *methods* se define un arreglo con los métodos del contrato. Cada método está compuesto por su nombre y sus parámetros. Los parámetros se dividen en 2 clases, los requeridos y los reservados. Estos parámetros pueden pensarse como variables y fijas y variables ligadas. Es decir, los argumentos requeridos están fijos en posiciones y valores específicos, ya que esto es necesario para la invocación del método, y sin estos, no es posible acceder al método, como por ejemplo, el selector de un método en los contratos que cumplen las convenciones definidas en ARC-04. El selector se espera que esté ubicado en la posición 0 del arreglo de argumentos y que su valor sean los primeros 4 bytes del hash `SHA512/256` de la firma del método. Los parámetros reservados son como variables libres, es decir, que pueden tomar cualquier valor. Usando al ejemplo anterior, los parámetros en el arreglo de argumentos “a” y “b” están libres. Por lo tanto, `Boogie` les asignará un

valor aleatorio. Los *string* en el *array* son interpretados directamente como byte *strings* de unicode crudo, para evitar problemas de codificación.

3.3. Validación de condiciones

En esta tesis se consideraron dos enfoques para la validación de un contrato inteligente. La validación puede realizarse de manera global, como invariantes del contrato, o locales, como pre y post condiciones para los métodos. Un invariante global debe mantenerse válido después de la ejecución de cualquier método, mientras que las condiciones locales solo debe hacerlo previa y posteriormente de la ejecución de su método específico. En particular, se implementó el segundo enfoque, ya que el primero puede representarse como una condición local para todos los métodos.

Las condiciones se subdividen en pre y post condiciones. Las precondiciones de un método son representadas como un *assume* de `Boogie`. Esto limita el espacio de búsqueda de `Corral` a encontrar contra ejemplos para invocaciones validas de un método. Idealmente, cualquier transacción que intente invocar un método y no respete las "precondiciones de llamada" al método, debería ser rechazada. Por ejemplo, el método `add`, no debería ejecutarse sin el selector correcto, ni la transacción debería aprobarse si fuese de tipo *DeleteApplication*. Dado que `TEAL` es código de bajo nivel y muchos de los contratos inteligentes fueron desplegados previos al ARC-04 o la existencia de herramientas o *frameworks* de desarrollo en `TEAL` que automatizan la generación de código que válida la correcta invocación de un método, también es de interés que la herramienta verifique que los métodos no puedan ejecutarse cuando no se respeten sus precondiciones. Además también existen las precondiciones de un método en sí. Por de ejemplo, un hipotético método `set_admin` debería requerir que el *Sender* de la transacción sea una cuenta privilegiada y cualquier intento de invocar el método por una cuenta no privilegiada, debería ser rechazada.

3.3.1. Generación

Se definió un esquema de JSON en donde se puede proveer a la herramienta una lista de nombres de funciones junto con sus respectivas postcondiciones a validar. Basándose en esto, la herramienta generará el código `Boogie` correspondiente. Como la AVM no utiliza registros y `TEAL` un lenguaje de bajo nivel, cualquier condición del estado del contrato inteligente está limitada a los posibles valores del almacenamiento, previa y posteriormente de haber ejecutado una transacción.

Listing 3.8: Ejemplo del JSON de condiciones

```
1 {
2   "set_admin": {
3     "precondition": "Global['admin'] == CurrentTx.Sender",
4     "postcondition": "Global['admin'] == CurrentTx.ApplicationArgs[0]"
5   }
6 }
```

Para la definición de la sintaxis de las condiciones y su procesamiento, se utilizó la biblioteca de *python* PLY[18]. `PLY` es una implementación de las herramientas de compilado `lex` y `yacc`[15].

Una posible futura iteración de la herramienta debería automatizar el proceso de generación de las validaciones. Idealmente, el código de alto nivel del contrato inteligente debería ser anotado con las condiciones y éstas deberían generarse en conjunto con el esquema correspondiente de ARC-04, similar a como funciona VeriSol.

4. EVALUACIÓN

4.1. Funcionalidad

Como método de evaluación se generaron una serie contratos inteligentes hechos a medida para la búsqueda de errores en las distintas partes involucradas en las transacciones de Algorand.

Los siguientes contratos implementan un contrato "contador". Este contrato busca un valor en el almacenamiento global y lo incrementa en uno. La única diferencia entre ellos es que uno usa una clave de tipo entero (izquierda), y el otro de tipo *byte* (derecha). A continuación, una descripción paso a paso:

1. Las lineas 4 y 5 cargan la clave y duplican su valor en la pila.
2. Linea 6 obtiene el valor asociado a la clave del estado global.
3. Lineas 9 y 10 suman 1 al valor obtenido.
4. Lineas 13 y 14 duplican el valor incrementado y lo guardan en el *Scratch slot* 0.
5. La linea 17 actualiza el estado global con el nuevo valor asociado a la clave 42.
6. 20 y 21 cargan el valor guardado en la posición 0 y lo devuelven como resultado.

Listing 4.1: Contrato contador con clave de tipo entero

```
1  #pragma version 4
2
3  // read global state
4  int 42
5  dup
6  app_global_get
7
8  // increment the value
9  int 1
10 +
11
12 // store to scratch space
13 dup
14 store 0
15
16 // update global state
17 app_global_put
18
19 // load return value as approval
20 load 0
21 return
```

Listing 4.2: Contrato contador con clave de tipo *byte*

```
1  #pragma version 4
2
3  // read global state
4  byte "counter"
5  dup
6  app_global_get
7
8  // increment the value
9  int 1
10 +
11
12 // store to scratch space
13 dup
14 store 0
15
16 // update global state
17 app_global_put
18
19 // load return value as approval
20 load 0
21 return
```

Ambos utilizan la misma interfaz:

```
{
    "methods": [
      {
        "opcode": 0,
        "name": "constructor",
        "required": {
                "arguments": {},
                "accounts": {},
                "applications": {},
                "assets": {}
        },
        "reserved": {
                "arguments": [],
                "accounts": [],
                "applications": [],
                "assets": []
        }
      }
    ]
}
```

Para las interfaces hechas a medida, `VeriTEAL` requiere que exista un método específico en el arreglo de métodos llamado "constructor". Este método se invoca previamente al *harness* principal. En este caso, como toda invocación del contrato, para cualquier argumento, ejecuta el mismo flujo, solo se define el "constructor" únicamente.

Luego se definen sus pre y post condiciones:

```
{
  "constructor": {
    "preconditions": ["true"],
    "postconditions":
            ["Global[42] < 5"]
  }
}
```

```
{
  "constructor": {
    "preconditions": ["true"],
    "postconditions":
            ["Global['counter'] < 5"]
  }
}
```

Asumiendo que las condiciones son correctas, es decir, que representan correctamente la especificación del contrato, puede verse a simple vista que existe un error en el contrato, ya que a la quinta invocación, el valor en el almacenamiento global será igual a 5.

A continuación vemos una versión simplificada del código Boogie sobre el listado 4.1 producido por `VeriTEAL`:

En primer lugar se definen los procedimientos basados en la interfaz provista. Estos preparan los argumentos necesarios para ejecutar cada flujo de control específico de su método y luego, llaman al procedimiento `contract`, que representa al contrato inteligente escrito en Algorand. Por último, se define el procedimiento principal `verify`, el cual `Corral`analizará e intentará encontrar contra ejemplos para las afirmaciones.

El siguiente listado muestra una simplificación del output de `Corral` para el programa `Boogie` del listado previo.

En este se puede ver que `Corral` encuentra un *True Bug* y provee la traza de ejecución para llegar al contraejemplo que rompe la afirmación.

```
1  procedure constructor();
2  implementation constructor_(){
3          call contract();
4  }
5
6  procedure contract();
7  implementation contract(){
8          var scratch_0 : int;
9          ...
10         var scratch_63 : int;
11         var local_0 : int;
12         var local_1 : int;
13         var local_2 : int;
14         var local_6 : int;
15         var local_3 : int;
16         label_0:
17                 local_0 := 42;
18                 local_1 := Global[local_0];
19                 local_2 := 1;
20                 local_3 := local_2 + local_1;
21                 scratch_0 := local_3;
22                 Global[local_0] := local_3;
23                 local_6 := scratch_0;
24                 return_variable := local_6;
25                 goto label_exit;
26         label_exit:
27                 return;
28 }
29
30 procedure verify();
31 implementation verify(){
32         assume (forall key: int :: Global[key] == 0);
33         assume true;
34         call constructor();
35         assert Global[42] < 5;
36         while (true){
37                 havoc choice;
38                 if ((choice) == 0) {
39                         assume true;
40                         call constructor();
41                         assert Global[42] < 5;
42                 }
43         }
44 }
```

```
1  Single threaded program detected
2  Verifying program while tracking: {assertsPassed, Global}
3  Program has a potential bug: True bug
4  PersistentProgram: error PF5001: This assertion can fail
5  Execution trace:
6    (CALL constructor)
7      (CALL contract)
8      (RETURN from contract)
9    (RETURN from constructor)
10   (CALL constructor)
11     (CALL contract)
12     (RETURN from contract)
13   (RETURN from constructor)
14   (CALL constructor)
15     (CALL contract)
16     (RETURN from contract)
17   (RETURN from constructor)
18   (CALL constructor)
19     (CALL contract)
20     (RETURN from contract)
21   (RETURN from constructor)
22   (CALL constructor)
23     (CALL contract)
24     (RETURN from contract)
25   (RETURN from constructor)
26 (1,0)  (ASSERTION FAILS assert Global[42] < 5;
```

4.2. Caso de estudio: Tinyman

Tinyman es un intercambio descentralizado (DEX) que opera en la blockchain de Algorand. Permite a los usuarios intercambiar *tokens* de manera descentralizada sin necesidad de intermediarios. Al igual que otros DEX, como Uniswap en Ethereum, Tinyman utiliza contratos inteligentes para facilitar los intercambios y proporcionar liquidez. Los usuarios pueden agregar liquidez a los *pools* de Tinyman y, a cambio, recibir una parte de las comisiones de transacción generadas por el intercambio. Tinyman es conocido por sus transacciones rápidas y de bajo costo gracias a la eficiencia de la blockchain de Algorand. El primero de enero de 2022, algunos de los *pools* de Tinyman fueron atacados. El atacante explotó una vulnerabilidad desconocida en el código del contrato inteligente para extraer aproximadamente $3 millones en *tokens* de los *pools* del contrato. La vulnerabilidad fue rápidamente emparchada a los pocos días del ataque.

Según Rob Henke [2], la explotación de la vulnerabilidad en Tinyman se debió a un fallo en el código del contrato inteligente del proyecto. Cuando un usuario llama al método *burn* del protocolo, debería recibir dos *tokens* diferentes a cambio. Las cantidades de cada *token* dependen de la cantidad almacenada dentro del protocolo.

El atacante explotó una vulnerabilidad en el código del contrato de los *pools* de Tinyman que les permitió recibir el mismo *token* dos veces en lugar de dos diferentes. Esto fue ventajoso para el atacante, ya que le permitió extraer el doble de *gobtc* (una *wrap* de Bitcoin en Algorand), en lugar de una mezcla de *gobtc* y *ALGO*. Dado que *gobtc* es mucho más valioso que *ALGO*, esto permitió al atacante obtener una ganancia significativa y drenar aproximadamente $3 millones en *gobtc* y *goeth* del *pool* de Tinyman en múltiples transacciones. Estos *tokens* luego fueron intercambiados por monedas estables y retirados.

Un usuario (*Pooler*) puede agregar liquidez a un *Pool* contribuyendo con valores equi-

valentes de cada activo en un par a la cuenta del *Pool* de ese par. A cambio, el *Pool* recompensa al usuario con *tokens* de liquidez en proporción a la cantidad de activos que haya contribuido.

Cualquier usuario que posea *tokens* de liquidez puede "quemarlos" (algunos o todos) en cualquier momento intercambiándolos con el *Pool* por la proporción correspondiente de los activos del *Pool*. Si un *Pool* tiene un saldo de 10 activos A, 100 activos B y ha emitido 10 *tokens* de liquidez en circulación, 1 token de liquidez puede ser quemado por 1 activo A y 10 activos B. El *Pool* tendría entonces un saldo de 9 activos A, 90 activos B y habría emitido 9 *tokens* de liquidez en circulación.

El proceso de *burn* de Tinyman está dividido entre Cuentas contrato (*LogicSigs*) y una Aplicación. Las Cuentas contrato son de particular interés, dado que éstas representan los *Pools* y contienen los activos.

La transición de *burn* es una transacción grupal compuesta de 5 sub transacciones:

0. Payment de Fees (firmado por Pooler)
1. ApplicationCall (firmado por Pool/LogicSig)
2. AssetTransfer/Payment (firmado por Pool/LogicSig)
3. AssetTransfer/Payment (firmado por Pool/LogicSig)
4. AssetTransfer/Payment (firmado por Pooler)

La primera transacción es el pago de las tarifas del protocolo. Éstas deben ser pagadas por el *Pooler*. La segunda es el llamado a la Aplicación, la cual se encarga de realizar algunas validaciones y actualizar los estados del *Pooler* y del *Pool*. La tercera y la cuarta son la transferencia de los activos desde del *Pool* al *Pooler*, en teoría los activos deberían ser distintos. La quinta y última transacción es la transferencia de los *tokens* de liquidez del *Pooler* hacia el *Pool*.

De la versión del *Pool* vulnerable se extrajo el código relevante a la vulnerabilidad y se generaron las siguientes post condiciones:

Esta especificación se generó manualmente basándose en los comentarios en el método *burn*. El código fuente puede encontrarse en [3]. En retrospectiva es fácil de ver que en ningún momento se requiere que los activos sean distintos.

Como es de esperar, `Corral` no encuentra ningún problema en el contrato.

```
1  Program has no bugs
2
3  Boogie verification time: 5.11 s
4  Time spent reading-writing programs: 3.19 s
5
6  Time spent checking a program (13): 4.10 s
7  Time spent checking a path (36): 5.50 s
8
9  Number of procedures inlined: 35
10 Number of variables tracked: 19
11 Total Time: 10.4012089 s
12 Total User CPU time: 10.29 s
```

```
{
  "constructor": {
    "preconditions": ["true"],
    "postconditions": [
      "GroupSize == 5",
      "GroupTransaction[0].Sender != Sender",
      "GroupTransaction[0].Receiver == Sender",
      "GroupTransaction[0].Amount >=
        (
          GroupTransaction[1].Fee +
          GroupTransaction[2].Fee +
          GroupTransaction[3].Fee
        )",
      "GroupTransaction[1].Accounts[1] != Sender",
      "GroupTransaction[1].Accounts[1] == GroupTransaction[2].AssetReceiver",
      "(GroupTransaction[1].Accounts[1] == GroupTransaction[3].AssetReceiver) ||
        (GroupTransaction[1].Accounts[1] == GroupTransaction[3].Receiver)",
      "GroupTransaction[2].Sender == Sender",
      "GroupTransaction[2].AssetReceiver == GroupTransaction[4].Sender",
      "GroupTransaction[3].Sender == Sender",
      "(GroupTransaction[4].Sender == GroupTransaction[3].AssetReceiver) ||
        (GroupTransaction[4].Sender == GroupTransaction[3].Receiver)",
      "GroupTransaction[4].Sender != Sender",
      "GroupTransaction[4].AssetReceiver == Sender",
    ]
  }
}
```

Agregando la siguiente condición a la especificación:

```
GroupTransaction[2].XferAsset != GroupTransaction[3].XferAsset
```

`VeriTEAL` encuentra un error en el programa:

Listing 4.3: Simplificacion de la salida traza de `Corral`

```
Program has a potential bug: True bug
PersistentProgram(9318,1): error PF5001: This assertion can fail
veriteal/artifacts/output.bpl: error PF5001: This assertion can fail

(1,0) (ASSERTION FAILS assert XferAsset_2 != XferAsset_3;)

Boogie verification time: 5.30 s
Time spent reading-writing programs: 3.33 s

Time spent checking a program (13): 4.19 s
Time spent checking a path (37): 5.73 s

Number of procedures inlined: 35
Number of variables tracked: 19
Total Time: 10.9014467 s
Total User CPU time: 10.9 s
```

5. CONSIDERACIONES Y LIMITACIONES

La decisión de no implementar ciertas características en la herramienta prototipo fue impulsada por la necesidad de equilibrar la ambición con la viabilidad dentro de las limitaciones del proyecto. Si bien estas características prometen mejorar las capacidades de la herramienta en el futuro, su implementación requiere una planificación cuidadosa, investigación y esfuerzos de desarrollo más allá del alcance de la línea de tiempo actual del proyecto. En un futuro, se podrá continuar explorando oportunidades para incorporar estas características en iteraciones posteriores.

5.1. Consideraciones específicas

Operaciones de manipulación de *strings*: Boogie no soporta nativamente *strings*, por lo que las secuencias de bytes se representan con números enteros. Por este motivo, los programas que manipulan *strings* no pueden ser analizados por la herramienta.

Completitud de opcodes: Además de los operadores de *strings*, existen otras operaciones no implementadas debido a que no fueron consideradas necesarias para la prueba de concepto de la herramienta, como por ejemplo, los operadores de enteros grandes (`mulw`, `addw`, `divw`), los operadores de creación de transacciones internas (`itxn_begin`, `itxn_submit`, `itxn_field`), los operadores de *boxes* y *storage* local y los operadores para subrutinas (`proto`, `frame_dig`, `frame_bury`) entre los más destacables.

***LocalStorage* y *Boxes*:** La funcionalidad de almacenamiento local y cajas de almacenamiento ilimitado no han sido integrados en esta versión del prototipo.

InnerTransactions: Las transacciones internas no están soportadas en esta versión del prototipo. El único tipo de transacción interna que presenta interés es la transición tipo llamado de aplicación ya que puede retornar valores. Su modelado puede realizarse de varias maneras: Si se conocen invariantes de la aplicación llamada, el resultado de la llamada puede ser retornado asumiendo el éxito de la transacción interna. Este proveería un análisis más certero. Caso contrario, también podría asumirse que el resultado de esta llamada puede tener cualquier valor, lo cual podría generar falsos positivos.

Naturaleza prototípica: Siendo un prototipo, es posible que la herramienta presente fallos significativos.

6. TRABAJOS RELACIONADOS

El campo de la verificación formal ha sido ampliamente estudiado, con contribuciones significativas que han moldeado nuestra comprensión y enfoque actual. En el marco de Algorand existen unos pocos trabajos que se enfocan en este campo. El objetivo de esta sección es examinar el estado del arte en el campo de la verificación formal para Algorand. Al examinar las metodologías, hallazgos y conclusiones de estudios previos, buscamos establecer una conexión clara entre la investigación pasada y los objetivos de esta tesis.

En primer lugar, volver a mencionar a `VeriSol` [10], ya que `VeriTEAL` herramienta está fuertemente inspirada en ésta.

En [12] introducen un método para mejorar la seguridad de los contratos inteligentes utilizando "verificación gradual", combinando técnicas de verificación estática y dinámica. Este enfoque equilibra precisión y flexibilidad, proporcionando verificación incremental y optimizando el uso de recursos. Al implementar este método para los contratos inteligentes de Algorand utilizando el lenguaje `pyTEAL`, los autores afirman proteger efectivamente contra vulnerabilidades, incluyendo ataques de reentrada. El método de verificación gradual garantiza *soundness* y reduce la sobrecarga en tiempo de ejecución, haciendo que la verificación sea más accesible para los desarrolladores. Los autores presentan un enfoque muy similar a `VeriTEAL`. La herramienta transforma *pyTEAL* a un lenguaje de representación intermedio C0, una variante de la cadena de herramienta Viper, y utiliza este para realizar un análisis estático en base a pre y post-condiciones provistas por el desarrollador. Además los autores desarrollaron un modulo llamado Weaver que realiza verificación dinámica para cubrir los casos en los que el verificador estático no pudo resolver las condiciones.

En [17] introducen `Panda`, un marco de análisis estático extensible diseñado para detectar automáticamente vulnerabilidades comunes en contratos inteligentes de Algorand.Similar al primer enfoque explorado en esta tesis, `Panda` realiza ejecución simbólica modelando la AVM en *Python* y utiliza `Z3` para encontrar trazas de ejecución donde una transacción es aprobada. Luego de haber hallado una traza de ejecución válida, `Panda` invocará a la componente de análisis para intentar detectar vulnerabilidades. A diferencia de `VeriTEAL`, `Panda` busca vulnerabilidades comunes (*RekeyTo* no validado, borrado arbitrario, entre algunas) en vez de errores en la implementación. Una ventaja de este enfoque es que no requiere una especificación del contrato. Los autores notan que las vulnerabilidades en las aplicaciones de Algorand están muy esparcidas. Notablemente, 4,008 (4.04 %) aplicaciones en la blockchain y 150,676 (27.73 %) aplicaciones fuera de la blockchain están marcadas como vulnerables.

7. CONCLUSIONES

Esta tesis ha explorado la verificación formal de contratos inteligentes en la *blockchain* de Algorand, con un enfoque en la traducción del lenguaje `TEAL` al lenguaje intermedio `Boogie`. Al aprovechar la herramienta de verificación `Corral`, se desarrolló un enfoque sólido que puede mejorar la seguridad y la fiabilidad de los contratos inteligentes desplegados en Algorand. Este trabajo contribuye al campo de la seguridad en *blockchain* al proporcionar un método sistemático para asegurar la corrección de la lógica de los contratos, reduciendo así el potencial de vulnerabilidades y pérdidas financieras.

Los resultados de nuestra investigación demuestran que la verificación en base a la traducción a `Boogie`, seguida de la verificación formal utilizando `Corral`, es tanto factible como efectiva. Este método no solo refuerza el marco de seguridad para Algorand, sino que también muestra un modelo potencial para adaptar estrategias de verificación similares a otras plataformas.

Si bien nuestro enfoque aborda muchos de los desafíos asociados con la verificación de contratos inteligentes, no está exento de limitaciones. La complejidad de los contratos y la naturaleza en evolución de la tecnología *blockchain* presentan desafíos continuos que requieren la adaptación continua de las técnicas de verificación. La investigación futura podría centrarse en mejorar la escalabilidad del proceso de verificación y ampliar su aplicabilidad a contratos más complejos.

En conclusión, el método de verificación formal propuesto en esta tesis representa un paso hacia la mejora de la confianza de los sistemas *blockchain*. Al garantizar que los contratos inteligentes en Algorand funcionen según su especificación, este trabajo contribuye a la adopción y éxito general de las aplicaciones descentralizadas, y a la adopción de técnicas de verificación formal automatizadas.

BIBLIOGRAFíA

[1] URL: https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/IR-trans1.pdf.

[2] URL: https://www.halborn.com/blog/post/explained-the-tinyman-hack-january-2022.

[3] URL: https://github.com/tinymanorg/tinyman-contracts-v1/blob/99da0d210f98c535ce4b0a22 contracts/pool_logicsig.teal.tmpl.

[4] *Algorand consensus - Algorand Developer Portal.* URL: https://developer.algorand.org/docs/get-details/algorand_consensus/. (accessed: 2023-09-15).

[5] *Algorand Transaction Calling Conventions.* URL: https://github.com/algorandfoundation/ARCs/blob/main/ARCs/arc-0004.md. (accessed: 2023-11-23).

[6] *Beaker: A framework for writing Smart Contracts on Algorand.* URL: https://github.com/algorand-devrel/beaker.

[7] «Boogie: A Modular Reusable Verifier for Object-Oriented Programs». En: (2005). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/krml160.pdf.

[8] «Corral: A Solver for Reachability Modulo Theories». En: (2016). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/techreport-1.pdf.

[9] Algorand Developers. *Global and Local State.* 2023. URL: https://www.youtube.com/watch?v=RT_rweOUWk0 (visitado 21-08-2024).

[10] *Formal Specification and Verification of Smart Contracts in Azure Blockchain.* URL: https://www.microsoft.com/en-us/research/uploads/prod/2019/05/1812.08829.pdf. (accessed: 2023-11-23).

[11] *Full Technical Report on Attacks.* URL: https://tinymanorg.medium.com/full-technical-report-on-attacks-18e3c5e89c5f. (accessed: 2023-10-09).

[12] «Gradual Verification for Smart Contracts». En: (). URL: https://arxiv.org/pdf/2311.13351.

[13] Akash Lal y Shaz Qadeer. «Powering the Static Driver Verifier Using Corral». En: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 2014.

[14] K. Rustan M. Leino. «Dafny: An Automatic Program Verifier for Functional Correctness». En: *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer Berlin Heidelberg, 2010.

[15] John R Levine. *Lex & yacc.* Sebastopol, CA : O'Reilly & Associates, Inc., 1992.

[16] Patricio Palladino. «ContractorJ: validando el comportamiento de clases de Java». En: (2017). URL: https://lafhis.dc.uba.ar/users/~diegog/licTesis/2017-05-15-Palladino-Patricio.pdf.

[17] «Panda: Security Analysis of Algorand Smart Contracts». En: (). URL: https://www.usenix.org/system/files/usenixsecurity23-sun.pdf.

[18] *PLY (Python Lex-Yacc)*. URL: https://www.dabeaz.com/ply/ply.html.

[19] *The Algorand Virtual Machine (AVM) and TEAL.* URL: https://github.com/algorand/go-algorand/tree/master/data/transactions/logic#the-algorand-virtual-machine-avm-and-teal. (accessed: 2023-09-15).

[20] «This is Boogie 2». En: (2016). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml178.pdf.

[21] Wikipedia contributors. *Static single-assignment form — Wikipedia, The Free Encyclopedia.* https://en.wikipedia.org/w/index.php?title=Static_single-assignment_form&oldid=1224604107. [Online; accessed 11-June-2024]. 2024.