Universidad Zaragoza
1542

Trabajo Fin de Grado

DETECCIÓN DE VULNERABILIDADES EN CÓDIGO FUENTE A TRAVÉS DE REDES DE PETRI

DETECTION OF VULNERABILITIES IN SOURCE CODE THROUGH PETRI NETS

AUTORA

DOMINIQUE SALOMÉ REA ÁVILA

DIRECTORES

RICARDO J. RODRÍGUEZ

RAZVAN RADUCU

ESCUELA DE INGENIERÍA Y ARQUITECTURA

GRADO EN INGENIERÍA INFORMÁTICA

DICIEMBRE 2024

Resumen

En la actualidad, el desarrollo de software está en continuo cambio y mejora, lo que provoca que nuevas vulnerabilidades se añadan a las ya conocidas. Esto incrementa el riesgo para los sistemas y los datos sensibles que gestionan, haciendo necesario un desarrollo de software seguro. El ciclo de desarrollo de software engloba varias fases, desde el análisis y diseño inicial, pasando por la implementación y pruebas, hasta el despliegue y mantenimiento. Cada una de estas etapas puede presentar vulnerabilidades diferentes, que hacen fundamental contar con herramientas para enfrentarlas en cualquiera de ellas.

Por tanto, este Trabajo de Fin de Grado ofrece una nueva herramienta para detectar vulnerabilidades en código fuente, con un enfoque específico en la vulnerabilidad TOCTOU (*Time Of Check To Time Of Use*). La herramienta obtiene el Árbol Sintáctico Abstracto a partir de la compilación de un fichero en lenguaje C/C++, que se procesa para transformarlo al modelo formal de redes de Petri. Finalmente, se analiza la red obtenida para identificar los patrones que indican la posible presencia de esta vulnerabilidad.

Con el propósito de verificar el funcionamiento y la eficiencia, se ha evaluado la herramienta con distintos ejemplos y se ha realizado una prueba que mide el tiempo de ejecución del proceso. Los resultados muestran que la herramienta cumple el objetivo planteado con un tiempo de ejecución reducido.

Abstract

Currently, software development is constantly changing and improving, which causes new vulnerabilities to be added to those already known. This increases the risk for the systems and sensitive data they manage, making secure software development necessary. The software development cycle encompasses several phases, from initial analysis and design, through implementation and testing, to deployment and maintenance. Each of these stages can present different vulnerabilities, which make it essential to have tools to address them in any of them.
Therefore, this Final Degree Project offers a new tool to detect vulnerabilities in source code, with a specific focus on the TOCTOU (*Time Of Check To Time Of Use*) vulnerability. The tool obtains the Abstract Syntax Tree from the compilation of a source code in C/C++ language, which is processed to transform it into the formal Petri net model. Finally, the network obtained is analyzed to identify the patterns that indicate the possible presence of this vulnerability.
In order to verify operation and efficiency, the tool has been evaluated with different examples and a test has been carried out that measures the execution time of the process. The results show that the tool meets the stated goals with a reduced execution time.

Índice general

Lista de Figuras

Lista de tablas

Lista de códigos de ejemplo

Capítulo 1

Introducción

En lo que se refiere al desarrollo de sistemas informáticos, la seguridad es uno de sus atributos fundamentales. Con el avance tecnológico y el creciente uso de sistemas digitales en sectores clave como el económico y el gubernamental, el impacto de una vulnerabilidad de software es cada vez mayor [1] . Las brechas de seguridad pueden ser explotadas por terceros de manera que se comprometa la disponibilidad, integridad y confidencialidad de los sistemas [2].
En este contexto es donde el diseño y desarrollo seguros son importantes para minimizar las vulnerabilidades y ayudar al software a mantenerse funcional ante cualquier ataque. El uso de herramientas de análisis y detección de vulnerabilidades junto al de buenas prácticas de seguridad y las pruebas exhaustivas, ayudan a reducir el impacto que pueden causar los fallos en etapas avanzadas del desarrollo o incluso después del despliegue del sistema [3].

1.1 Motivación y objetivos

Dentro de todo el conjunto de vulnerabilidades conocidas, TOCTOU[1] *("Time-Of-Check to Time-Of-Use"*) es una de las más relevantes porque posee la capacidad de dañar gravemente la integridad, confidencialidad y disponibilidad de los sistemas. Esta vulnerabilidad ocurre cuando hay una ventana de tiempo entre el momento en que se verifica un recurso y el momento en que se utiliza, que permite que un atacante modifique el estado del sistema. El motivo de centrarse en la detección de este tipo de vulnerabilidades reside en que es difícil de prever manualmente ya que suele depender de patrones concretos de comportamiento. Se entiende como *patrón de comportamiento* a la serie de llamadas a operaciones que siempre están presentes cuando se materializa esta vulnerabilidad.

El objetivo principal de este proyecto es desarrollar una herramienta que detecte los patrones de este tipo de vulnerabilidad mediante la transformación de un código en C/C++ en una red de Petri. La elección de utilizar redes de Petri como modelo se basa en su capacidad para representar de manera formal y visual los flujos de ejecución, lo que facilita la identificación de patrones específicos que puedan indicar la presencia de vulnerabilidades. Este enfoque es útil para modelar tanto la concurrencia como la secuencia de operaciones, características fundamentales para detectar vulnerabilidades como TOCTOU. Para ello se persigue identificar dichos patrones de elementos de código, transformar un modelo intermedio a un modelo de patrones, analizar la red final obtenida para detectar el patrón de la vulnerabilidad mencionada, y evaluar la eficiencia de la herramienta propuesta.

1.2 Organización del documento

Este documento se encuentra dividido en siete capítulos y un anexo. En el Capítulo 2 se introducen los conceptos necesarios para comprender el trabajo, como las redes de Petri y la vulnerabilidad TOCTOU. En el Capítulo 3 se expone el estado del arte, presentando herramientas y estudios que utilizan las redes de Petri en el contexto de detección de vulnerabilidades. En el Capítulo 4 se detalla la metodología y el entorno tecnológico empleados en el presente proyecto. En el Capítulo 5 se describe la base desde la que se ha construido este trabajo y se describe la herramienta desarrollada. En el Capítulo 6 se presentan casos de estudio donde se refleja el funcionamiento de la herramienta. En el Capítulo 7 se realiza un análisis de rendimiento para evaluar la eficiencia. Finalmente, en el Capítulo 8 se ofrecen las conclusiones del proyecto y se sugieren futuros trabajos de mejora. Al final del documento se encuentra el Anexo, donde se muestra cómo se ha distribuido el tiempo invertido en este trabajo.

[1] *https://www.packetlabs.net/posts/what-are-tocttou-vulnerabilities/*

Capítulo 2

Conceptos previos

En este capítulo, se presentan una serie de definiciones que permiten entender las bases teóricas de los conceptos empleados en el desarrollo del proyecto, introduciendo el estándar de modelado, así como los modelos concretos que se han empleado en el diseño de este proyecto. Posteriormente, se define el análisis de código fuente y la relación de las redes de Petri con los sistemas concurrentes. Finalmente, se aborda la definición de la vulnerabilidad TOCTOU.

2.1 Modelos de diseño

Para el diseño del software se ha optado por el uso de *Unified Modeling Language (UML)*. UML fue creado para unificar el modelado de software de modo que pueda ser comprendido de manera universal entre los desarrolladores de software. En UML, se pueden diseñar diferentes tipos de diagramas para representar diversas vistas del sistema, como diagramas de clases, secuencia, actividades y casos de uso, entre otros. El modelado consolida el punto de partida para crear un sistema software que cumpla con lo que se necesita, así como con la escalabilidad, seguridad y robustez [4].

En concreto el diagrama de clases permite la representación de manera estática de un sistema, enfocado principalmente en la programación orientada a objetos. Está formado por las clases, elementos principales que definen un conjunto de objetos que comparten características; constan con nombre de clase, atributos y funciones. Otra parte son las relaciones, que identifican la dependencia entre dos o más clases, o de una clase hacia sí misma; la información que tienen es su multiplicidad y su nombre de asociación. Por último, se encuentran las interfaces que declaran atributos, funciones y obligaciones que no pueden ser instanciadas [5].

La ejecución de un proceso se refleja en un diagrama de actividades, que se usa para representar de manera dinámica el funcionamiento del sistema. Se compone de actividades, flujos de control y nodos iniciales y finales; elementos esenciales para la compresión de la funcionalidad plasmada [6].

2.2 Análisis de código fuente

El análisis de código fuente es un proceso que facilita la identificación de errores y vulnerabilidades para favorecer un desarrollo de software más robusto y eficiente. El código fuente se define como el conjunto de archivos que contienen las instrucciones escritas en un lenguaje de programación, ya sea compilado o interpretado, que forman la base de un programa [7]. Por ello, hay diferentes maneras en las que se puede abordar el análisis dependiendo de los aspectos que se quieran examinar.

Por un lado, el análisis estático es el más utilizado y trata de examinar el código fuente sin ejecutarlo. Permite identificar errores, malas prácticas de programación y vulnerabilidades [8]. Por otro, el análisis dinámico concede la posibilidad de observar cómo el sistema maneja los datos y los recursos en tiempo de ejecución. Puede detectar vulnerabilidades de bibliotecas que utilice el código [9].

Por su parte, el análisis formal analiza el código mediante modelos matemáticos que representan el comportamiento. Puede reconocer vulnerabilidades procedentes de patrones de ejecución específicos y la manera en la que las variables y los recursos son utilizados [10].

Según qué tipo de vulnerabilidades se quieran detectar, cada uno de estos enfoques tiene sus ventajas y desventajas. Por ejemplo, el análisis estático es rápido y efectivo para encontrar problemas de sintaxis en código o de pérdida de memoria, pero puede producir falsos positivos. En cambio, el análisis dinámico es efectivo en analizar la lógica y comportamiento de un programa, pero tiene limitaciones a la hora de detectar defectos de calidad del código [11]. Y, por otro lado, el análisis formal supone una mejora del método de búsqueda de entradas que no cumplan las propiedades que el sistema estaba destinado a cubrir, permitiendo así encontrar fallos [12].

2.3 Concurrencia y Redes de Petri

Se entiende como *concurrencia* al hecho de la ejecución simultánea de procesos o hilos; por subsiguiente, un sistema concurrente es aquel en el que diferentes componentes y/o procesos trabajan de esta manera. Estos sistemas pueden operar en base a memoria compartida o mediante paso de mensajes [13]. Para esta clase sistemas es más preciso el uso de redes de Petri como medio de detección de vulnerabilidades de condición de carrera como TOCTOU, debido a que ofrecen un arquetipo semántico que permite representar eventos concurrentes.

Las redes de Petri [14], formalizadas en los años sesenta por Carl Adam Petri [15], se trata de un grafo bipartito orientado con lugares, transiciones y arcos que conectan estos elementos entre sí. De esta manera, permiten el modelado de sistemas que representan sincronización, concurrencia y conflictos. El grafo es de tipo dirigido, donde los lugares tienen un marcado con el que se indica el estado en el que el sistema modelado se encuentra; estas marcas se llaman *tokens*.

Este tipo de estructura de grafo ofrece facilidades para el análisis de propiedades, como la ausencia de bloqueos, la accesibilidad de estados y el alcance de los recursos [16].

2.4 Condición de carrera: TOCTOU

Dentro de los tipos de vulnerabilidades que pueden darse en los sistemas informáticos destacan las conocidas como condiciones de carrera (*Race Conditions*). Una condición de carrera ocurre en sistemas concurrentes cuando dos o más procesos acceden y modifican un recurso simultáneamente. Esta situación es posible cuando no se han implementado mecanismos de sincronización por lo que no se asegura la consistencia de las operaciones. El impacto de esta vulnerabilidad puede ocasionar el acceso indebido o la obtención de privilegios no autorizados, la corrupción y robo de datos o la denegación de servicios [17].

TOCTOU es una de las clases perteneciente a las vulnerabilidades de condición de carrera. Aparece cuando un atacante explota el tiempo entre que se verifica una condición y se usa un recurso. Este tiempo se denomina ventana de vulnerabilidad, y se utiliza con el fin de ganar un acceso no autorizado sobre dicho recurso o para ejecutar operaciones maliciosas. En este escenario, las llamadas al sistema que se usan para verificar y acceder a los recursos no son atómicos, lo que genera esta ventana de vulnerabilidad [18].

Como ejemplo, el Código 1 ilustra el conjunto de instrucciones típico en la aparición de esta vulnerabilidad. La línea 7 es el inicio de la ventana vulnerable, ya que es donde se realiza la comprobación de permisos del fichero de entrada a través de la función *access()*. Una vez se ha validado que el fichero tiene permisos de escritura, se abre con la función *fopen()* como se ven en la línea 8. El problema reside en que estas funciones comprueban permisos de usuarios distintos, efectivo y real. Si el fichero de entrada tiene permisos de escritura para cualquier usuario y pertenece al atacante, *access()* será validado ya que el programa vulnerable tiene como usuario efectivo *root.* Sin embargo, *fopen()* comprueba que el usuario que ha iniciado el proceso tiene permiso de apertura, por lo que antes de que se complete esta operación, un atacante puede modificar el sistema de ficheros y lograr que *fopen()* abra un fichero para el que no tiene permisos.

Esto se puede explotar por ejemplo con un enlace simbólico a un fichero con acceso restringido a *root*, el cual se podrá abrir ya que *fopen()* se verifica sobre el fichero de entrada que pertenece al atacante.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  void main(int argc, char *argv[]){
5          char *filename = argv[1];
6
7          if(!access(filename, W_OK)){ // Time of Check
8                  FILE* file = fopen(filename, "a+"); // Time of Use
9                  fclose(file);
10         } else {
11                 printf("[+] ERROR\n");
12         }
13 }
```

Código 1 Ejemplo con vulnerabilidad TOCTOU

Capítulo 3

Estado del Arte

Actualmente, en el campo de la ciberseguridad existen múltiples herramientas para realizar el análisis de código fuente. En este capítulo, se ha procedido con la búsqueda de herramientas y enfoques ya presentes en la literatura, así como aquellas que se centran en el uso de redes de Petri como medio para detectar vulnerabilidades.

3.1 Enfoques y herramientas de detección

En esta sección se presentan los principales enfoques que se utilizan para detectar vulnerabilidades en código fuente, junto a las herramientas usadas más relevantes actualmente.

Por un lado, para el análisis estático que puede estar basado en patrones, flujo de datos o de control [19], destaca la herramienta SonarQube[2]. De código abierto, esta herramienta ofrece un enfoque en la calidad del código y su seguridad. Sin embargo, está limitada en la detección de patrones concurrentes complejos. Otra herramienta es Fortify[3], que cuenta con una base de datos de reglas de seguridad y una interfaz para crear reglas personalizadas.

Por otro lado, el análisis dinámico puede tener lugar mediante la generación aleatoria de entradas, la monitorización de la ejecución para observar el uso de los recursos o la detección de fallos en la memoria [20]. Entre diversas herramientas, Valgrind[4] cuenta con módulos que permiten realizar este análisis y en ciertos escenarios es útil para detectar condiciones de carrera. Asimismo, ThreadSanitizer[5], programa desarrollado por Google con base en Valgrind, facilita la detección de problemas de concurrencia que pueden pasar desapercibidas en el análisis estático. En lo referente a herramientas que permitan construir grafos y analizarlos, se puede nombrar a CodeSonar[6].

3.2 Redes de Petri y el análisis de sistemas concurrentes

En esta sección se van a presentar diferentes estudios que han tenido como foco el uso de las redes de Petri para analizar sistemas concurrentes, junto a herramientas que realizan dicho análisis.

Respecto a herramientas de representación existentes para editar, simular y analizar redes de Petri, particularmente un tipo de redes de Petri llamadas coloreadas, destaca CPN Tools[7]. Por otro lado, TAPAAL[8] se centra en el análisis de redes temporizadas para modelar eventos temporales y analizar cómo se sincronizan los recursos utilizados.

---

2 https://lc.cx/cTHYPt
3 https://www.opentext.com/es-es/productos/fortify-static-code-analyzer
4 https://hardfloat.es/blog/2022/03/11/herramientas-deteccion-errores-valgrind-sanitizers.html
5 https://code.google.com/archive/p/data-race-test/wikis/ThreadSanitizer.wiki
6 https://codesecure.com/our-products/codesonar/
7 https://cpntools.org/
8 https://www.tapaal.net/

En cambio, si se habla propiamente del estudio de los sistemas concurrentes con redes de Petri se puede mencionar [21], que utiliza redes de Petri no deterministas para modelar sistemas. En dicho estudio, las arquitecturas actuales son tan complejas que se diseñan para ser no deterministas, por tanto, crean las *NT-Petri Nets*, redes de Petri coloreadas extendidas. Este framework facilita la construcción y ejecución con este tipo de redes en las que cada transición está formada por un conjunto de funciones. De esta manera, se puede optimizar un programa resolviendo el número máximo de hilos útiles en su ejecución.

Respecto a la detección de vulnerabilidades, se puede mencionar el estudio [22] para el modelado de escenarios de ataque complejos que usa redes de Petri temporizadas coloreadas. Basándose en probabilidades estocásticas, se expone un mecanismo para el modelado sistemático, la simulación y la explotación de vulnerabilidades complejas multinivel y multi-agente que se presentan en redes de trabajo y sistemas distribuidos.

Por último, conviene destacar la herramienta *Evinrude* [23]. Se trata de una herramienta que toma un código fuente en C y lo transforma en una red de Petri para realizar análisis dentro del contexto de los sistemas de detección de intrusiones.

Capítulo 4

Metodología

En este capítulo se presenta una descripción del proceso de desarrollo que se ha llevado a cabo. Asimismo, se especifica el entorno tecnológico con el que se ha trabajado.

4.1 Descripción

Se parte de la necesidad de obtener todos los constructos del código fuente a transformar para así plasmar el carácter jerárquico del código y conservar las estructuras principales. Por ello, se opta por el uso del Árbol Sintáctico Abstracto (AST), ya que viene incorporado en el compilador de *Clang* y ofrece una estructura fácil de leer e interpretar.

Una vez se ha decidido que se trabaja con el AST, se ha llevado a cabo la recopilación de los requisitos que se quiere que cumpla el sistema, acorde a las necesidades que se busca cubrir en cuanto al análisis de código fuente. Seguidamente, se ha creado el diseño de la estructura interna que se necesita para la implementación. Dicho arquetipo se verá reflejado en los diagramas UML.

El siguiente paso es plasmar lo diseñado en la implementación, creando cada clase y fichero que sea necesario. Aquí es donde se empieza con la transformación del AST en red de Petri. Tal y como se ha descrito anteriormente las redes de Petri son una estructura que facilita el análisis de sistemas concurrentes. Dado que el objetivo es examinar el código fuente en busca de la vulnerabilidad TOCTOU, esta representación es la más adecuada para tener una visión completa de los casos donde se pueden presentar este fallo de seguridad.

Cada constructo del código se puede representar por un conjunto de objetos lugar/transición que constituyen un patrón. Por ello, para la implementación se ha tomado un enfoque individual donde se ha ido transformando un patrón a la vez. Este método es escalar, ya que permite definir las estructuras grandes combinando las más pequeñas y simples. Es decir, se empieza por la implementación de los patrones de declaración de variables, funciones y referencias para seguir con las de estructuras de control.

Para comprobar que el resultado obtenido es el esperado, por cada implementación de un patrón se han ido realizando pruebas con ficheros de muestra y se ha verificado que se visualizaban de manera correcta. Cuando se ha completado la transformación del conjunto de patrones de estructuras se procede a la fase de validación de la herramienta.

En esta fase se presentan tres casos de estudio que tienen como propósito mostrar la capacidad de la herramienta para la transformación del código fuente en una red de Petri. El primer caso se ha obtenido en base a un código fuente sin vulnerabilidad, y los dos siguientes cuentan con dicha vulnerabilidad para así observar el patrón que se genera. Así mismo, se ha realizado un testeo de la eficiencia de la herramienta en cuanto al tiempo de procesamiento, medido mediante una serie de ejecuciones con diferentes ficheros de complejidades distintas.

4.2 Entorno tecnológico

Como parte de este proceso se han utilizado diferentes herramientas que facilitan el trabajo presente. Se ha escogido trabajar con ficheros en lenguaje C/C++, puesto que son generalmente usados en aplicaciones críticas, y tanto C como C++ son los lenguajes más usados hoy en día [24].

El compilador que se utiliza es Clang[9] de LLVM (*Low-Level Virtual Machine*) ya que facilita la parte del procesado y transformación al Árbol Sintáctico Abstracto. El Árbol Sintáctico Abstracto (*Abstract Syntax Tree, AST*) proporciona una representación jerárquica de un código fuente. Transforma cada construcción, semántica o sintáctica, del programa en nodos omitiendo aquellos elementos que no afectan directamente a la ejecución del código. Es decir, se enfoca en los elementos estructurales principales y necesarios [25]. El AST se obtiene en formato JSON, puesto que ofrece fácil lectura y comprensión de los datos [26].

Para el diseño de los diagramas se ha usado *Visual Paradigm*[10] en su versión 17.0. Este entorno incluye herramientas que facilitan la creación de diagramas UML. En cuanto a la fase de implementación se ha optado por utilizar el lenguaje de programación Python en su versión 3.11.9, ya que es soportado en diferentes sistemas operativos.

Una vez que se termina con la transformación a red de Petri, el fichero final que se devuelve se escribe en PNML, ya que es un lenguaje creado para este tipo de redes [27]. La visualización de dicho fichero se realiza en el entorno llamado PIPE5 [28][29]. PIPE está basado en Java para la construcción y análisis de redes de Petri estocásticas y proporciona un editor completo con las funcionalidades necesarias para este tipo de representación.

9 https://clang.llvm.org/index.html
10 https://www.visual-paradigm.com/

Capítulo 5

Análisis, diseño e implementación

En este apartado se expone el modelo que ha servido como punto de partida para el trabajo expuesto en este documento y que constituye la primera fase del desarrollo. Por otro lado, se explica la segunda fase que consta del análisis, diseño e implementación misma de la herramienta.

5.1 Modelo como punto de partida

Como base importante que sustenta este trabajo, resulta imprescindible nombrar el estudio de [30]. Con el propósito de realizar la transformación del código fuente en redes de Petri, se realiza una lectura y un tratamiento del AST. El AST obtenido de la compilación con *Clang* de un fichero en C/C++ se compone de los nodos de estructuras relevantes, referentes a estructuras de control, declaraciones y definiciones de funciones y variables, llamadas y referencias a funciones como se puede observar en la Tabla *1*.

Constructo de programación	Nodo del AST
Declaraciones	
Estamento de Declaración	DeclStmt
Declaración o definición de variable	VarDecl
Declaración o definición de funciones	FunctionDecl
Declaración o definición de parámetros de función	ParmVarDecl
Struct, union o clase	RecordDecl
Campo de un struct, union o clase	FieldDecl
Llamadas	
Llamada a función	CallExpr
Operadores	
Operadores binarios	BinaryOperator
Operadores unarios	UnaryOperator UnaryExprOrTypeTraitExpr
Estructuras	
If	IfStmt
Return	ReturnStmt
Bucle For	ForStmt
Bucle While	WhileStmt
Bucle Do-while	DoStmt

Tabla 1 Nodos relevantes del AST

En este TFG se ha desarrollado un script, denominado *Model.py* [31], y programado en Python que utiliza internamente *Clang*. De esta manera, deja un AST en formato JSON que se usará en la segunda fase del desarrollo. A su vez, simplifica el árbol para dejar las entradas al mismo nivel y facilitar el acceso a los mismos. El flujo de ejecución, donde se va leyendo cada nodo y guardándolo si es relevante, se puede observar en la Figura 1.

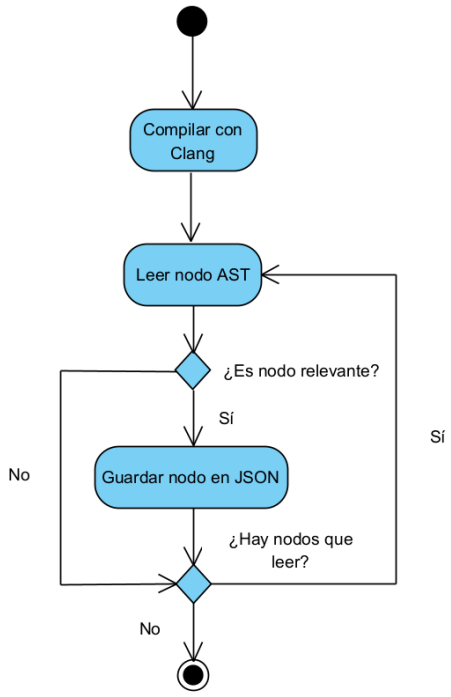

Figura 1 Diagrama de Actividad UML de Model.py

Una vez termina el proceso se obtiene un diccionario que corresponde a los nodos del AST relevantes de cada estructura existente en el código inicial. Cada entrada de este diccionario es un nodo padre que consta con su propio *"id"* y *"kind"* junto a información destacada, ya sea su nombre o sean los identificadores de sus nodos hijos, si existen. Por ejemplo, un nodo padre puede ser una estructura *If* que tiene como hijo un Operador Binario.

A modo de ejemplo se muestran dos tipos de nodos. En el Código 2 a la izquierda, se puede observar el nodo padre *"ReturnStmt"* que se corresponde a una construcción de *"return"*. Este nodo consta de su propio identificador en el par clave-valor, *"id"*: *"0x131e060"*. También, se contempla cómo se incorpora la etiqueta de su padre bajo la clave *"parent"* que se corresponde con la función que almacena este retorno. Esta información es útil para poder realizar operaciones con las relaciones entre los diferentes nodos. Otro dato importante es la clave *"inner"* que refleja los identificadores de los nodos hijos. Como se puede observar en el *Código 2* a la derecha, el nodo es un *"IntegerLiteral"* cuyo padre es el nodo "ReturnStmt". Con respecto a entradas de diccionario, ambos se encuentran al mismo nivel y no se pierde su relación, pues el hijo siempre tiene el identificador de su padre.

El nodo *"IntegerLiteral"* cuenta con los mismos datos que el nodo *"ReturnStmt"*. Sin embargo, por su tipo de estructura, posee una clave nueva denominada *"value"* que muestra el valor devuelto, o una clave *"type"* que indica el tipo de dato devuelto (por ejemplo, en este escenario, es un entero de valor 0).

Este tipo de relación se da en diferentes estructuras de programación. Por ejemplo, pueden aparecer operadores binarios dentro de otros. Sea cual sea la combinación, se verá reflejada en la relación de identificadores de cada nodo.

```
"0x131e060": {
      "parent": "0x131e070",
      "id": "0x131e060",
      "kind": "ReturnStmt",
      "range": {
          "begin": {
             "offset": 360,
             "line": 27,
             "col": 2,
             "tokLen": 6
          },
          "end": {
             "offse": 367,
             "col": 9,
             "tokLen": 1
          }
      },
      "inner": [
         "0x131e040"
      ]
},
```

```
"0x131e040": {
   "parent": "0x131e060",
   "id": "0x131e040",
   "kind": "IntegerLiteral",
   "range": {
     "begin": {
         "offset": 367,
         "col": 9,
         "tokLen": 1
      },
      "end": {
         "offset": 367,
         "col": 9,
         "tokLen": 1
       }
   },
   "type": {
       "qualType": "int"
   },
   "valueCategory": "prvalue",
   "value": "0"
}
```

Código 2 Nodos AST de ejemplo, “ReturnStmt” a la izquierda e “IntegerLiteral” a la derecha

5.2 Requisitos del sistema

Requisitos funcionales (RF)

Con los requisitos siguientes se busca definir como debe comportarse la herramienta a desarrollar.

RF1. La aplicación realizará una transformación del código fuente a AST, y del código AST a una red de Petri.
RF2. La aplicación realizará un análisis estático de vulnerabilidades en códigos fuente.
RF3. La aplicación detectará vulnerabilidades de tipo TOCTOU.
RF4. La aplicación informará del resultado del análisis estático de vulnerabilidades al usuario.

Requisitos no funcionales (RFN)

Definiendo estos requisitos se pretende plasmar las características y propiedades que se necesita que posea la herramienta.

RNF1. La aplicación se desarrollará para múltiples plataformas.
RNF2. La red de Petri de salida estará en formato PNML.
RNF3. La aplicación incorporará políticas de *logging*.
RNF4. La aplicación será robusta ante fallos.

Restricciones

R. La aplicación utilizará *Clang* para obtener el AST de códigos fuente de lenguajes C y C++.

5.3 Diseño del sistema

5.3.1 Flujo de ejecución

Una vez se definen los requisitos que se espera que tenga el diseño, se procede a realizar un diagrama de actividad. La Figura 2 refleja la funcionalidad del proceso que se va a programar para tener una visión inicial.

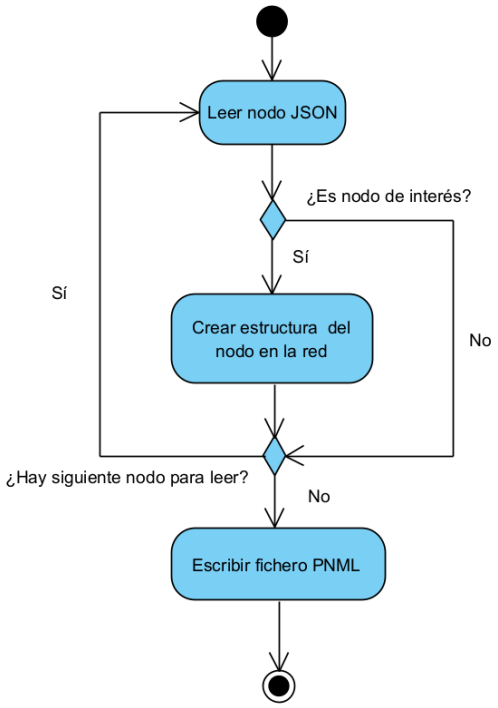

Figura 2 Diagrama de Actividad UML herramienta

Tal y como se observa, el proceso se inicia con la actividad "Leer nodo JSON" que lee cada nodo del AST de entrada obtenido en la fase previa. Si es un nodo de interés, se crea su estructura respectiva para la red de Petri que se está construyendo. En caso contrario, se procede a leer el siguiente nodo, si existe. Cuando ya no hay más nodos que leer, se realiza la escritura entera del fichero PNML final.

5.3.2 Diagrama de Clases

Con el flujo de ejecución establecido, es necesario determinar cuáles serán las clases de objetos esenciales para estructurar la herramienta. Para ello, se deben considerar las características clave que se busca cubrir y, de esta manera, elegir el patrón de diseño que mejor se adapte a los mismos:

- Modularidad en el formato de salida para que en caso de que se quiera cambiar o se quiera añadir uno nuevo, no se tenga que modificar toda la lógica interna.
- Reflejo de la distinción del proceso de transformación del AST y la generación de la salida de la red.
- Soporte de diferentes plataformas y posibilidad de que abarque más lenguajes de programación.

Por tanto, un patrón de diseño que brinda unas especificaciones afines es el *Strategy*. Este patrón permite definir, encapsular y hacer intercambiable un conjunto de algoritmos [32].

En vista de que es un diseño que debe transformarse en una red de Petri con sus lugares y transiciones, se necesita contar con los mismos objetos en el funcionamiento. La Figura 3 plasma el diagrama de clases creado.

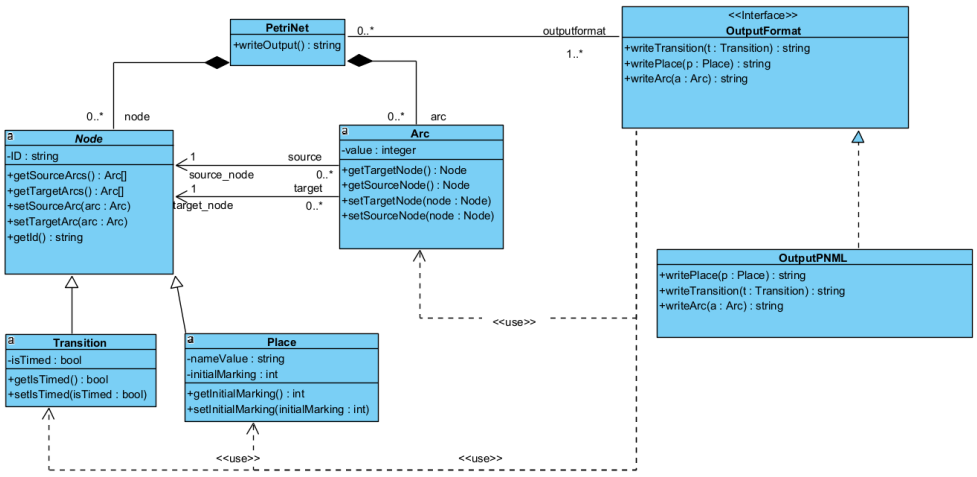

Figura 3 Diagrama de Clases UML herramienta

En este diagrama, se observa que *PetriNet* es una de las clases principales, representa la construcción de la red de Petri final y se encarga de devolver dicha red mediante su función *writeOuput()*. Junto a esta clase, se encuentran *Node*, que representa los nodos disponibles para dibujado, y *Arc* los arcos que establecen la relación entre estos componentes.

Esta clase tiene una relación de asociación de composición con *Node* y *Arc*, lo que indica que la red puede estar compuesta por múltiples elementos de estos tipos, como indica su numeración de *0 …**. Por otro lado, cuenta con una relación de agregación con la interfaz *OutputFormat*, indicando que la red podrá tener diferentes formatos de salida.

Node contiene el atributo <ID> donde se almacenará el identificador del nodo relevante leído. Además, tiene funciones para crear y obtener arcos, generando de esta manera las conexiones necesarias. De esta clase base heredan las otras dos llamadas *Transition* y *Place***.** La primera tiene un atributo para especificar si es temporizada o no y unas funciones para modificar y obtener este dato. En cambio**,** la segunda tiene variables para guardar el nombre del lugar (si tiene una marca inicial) y los métodos con los que se compone sirven para operar con esta marca. A su vez**,** *Arc* almacena el valor que pueda llegar a tener en la relación con los otros elementos y posee métodos para poder gestionar qué nodos son el origen y su destino.

Aparte de estas clases, una clase que implementa *OutputFormat* es *OuputPNML*. Esta es la que exporta la red en el formato estándar de PNML.

5.4 Implementación

En la implementación de la transformación a red de Petri, se han tenido que tomar una serie de decisiones con el fin de hacer el proceso lo más simple y fácil posible.

Primeramente, se ha decidido qué tipos de nodos del AST se van a representar en cada uno de los objetos que componen una red de Petri. Las transiciones son los sucesos que cambian los estados de la red (véase la Sección 2.3), así que representan los procesos de un programa [33]. Por ello, son un buen objeto para modelar las estructuras de control que determinan el flujo de ejecución del código fuente mediante las condiciones que se evalúan en sus guardas. Lo mismo sucede con las funciones, porque reflejan un cambio al ejecutar ciertos bloques de instrucciones. Así pues, se establece que los nodos de funciones, la declaración de variables, las estructuras de control y los operadores [30] serán representados por *Transition.*

Al mismo tiempo, otro elemento que será representado por una *Transition* serán las transiciones que aparecerán entre los patrones de constructos secuenciales. Así, los patrones que representan constructos anidados no contarán con esta transición intermedia. Por otra parte, para la construcción de los patrones de estructuras de control, también aparecerán objetos *Transition* estándar que son comunes en todos los modelos. Por ejemplo, sirven para marcar que se entra en el bloque de actividad de un bucle iterativo. Estas transiciones características de los patrones de dibujado se pueden observar en [30].

Por otro lado, los lugares representan aquellas condiciones que provocan que una transición se ejecute [33]. Estas condiciones, en el código fuente, varían según se modifiquen o usen variables. Por ejemplo, la evaluación de una variable en la condición de un *if* dictamina que se ejecute o no un bloque de instrucciones. De este modo, los nodos de tipo variable, parámetro, campo de *struct*, enteros, caracteres y cadenas serán representados por el objeto *Place*.

Cada estructura por tratar se divide en un modelo que tiene su entrada y salida [30]. Estas conexiones que sirven para unir diferentes patrones son los *inputs* y *outputs* que están representados también por lugares como se puede ver en la Figura 4.

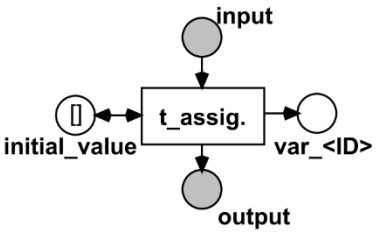

Figura 4 Patrón de definición de variable

Una parte importante en este tipo de redes es el marcado de los *tokens* que reflejan los recursos [28]. Por consiguiente, se ha establecido que, aparte del lugar inicial, llevarán marca los lugares que representan parámetros, variables, literales y lugares que se unen a la comprobación de la condición en las estructuras de control.

La manera en la que se ha ido implementando la herramienta ha sido en orden jerárquico. En este modelo de trabajo, se han desarrollado primero las funciones de tratamiento de los componentes base, tales como variables y su declaración, parámetros, literales y declaración de funciones. De este modo, se encontraban ya añadidas para la incorporación de las estructuras complejas de control. En cambio, el modo en que la herramienta procesa la creación de la red sigue un enfoque progresivo; es decir, se lee cada nodo y, en caso de que sea uno de los nodos que interesa (Tabla 1) se transforma en su objeto respectivo.

Una vez que todos los nodos están transformados, se vuelca la escritura en PNML de cada elemento en el fichero final para que pueda ser leído y visualizado correctamente. Para esta parte se ha utilizado la biblioteca ElementTree[11] de Python que implementa métodos para parsear y crear datos en formato XML. Con el fin de asegurar que se plasman todas las etiquetas y valores necesarios para PIPE, se ha realizado un estudio de la estructura de los ficheros que genera. Se ha creado una red sencilla directamente en su interfaz y se ha obtenido el fichero generado donde se plasman las etiquetas de cada elemento. Así se ha podido replicar con más precisión en la herramienta desarrollada.

Licencia

El código está disponible en GitHub [31], bajo la licencia pública general de GNU versión 3 (GNU GPLv3).

[11] *https://docs.python.org/3/library/xml.etree.elementtree.html*

Capítulo 6

Casos de estudio

En este capítulo se presentan un conjunto de casos de estudio con los que se busca mostrar la capacidad de la herramienta para plasmar los patrones del código fuente. Primeramente, se muestra un caso de una estructura sin patrón TOCTOU y seguidamente se expone dos casos que contienen dicha vulnerabilidad.

6.1 Primer Caso

El primer caso que se expone es el fragmento de un código (obtenido de [31]) que presenta una declaración de variable, un bucle *Do-While* como estructura principal y una devolución de entero (Véase el Código 3).

```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main(){
5           int i = 0;
6           do{
7                   i++;
8           } while(i < 10);
9
10          return 1;
11  }
```

Código 3 Ejemplo de bucle do-while

Para tratar el caso, lo primero que hay que realizar es la tarea de compilación y limpieza. Al terminar se obtiene el AST que se usa como entrada para la herramienta. Dicho AST se lee para construir la red de Petri correspondiente que se puede ver en la Figura 5.

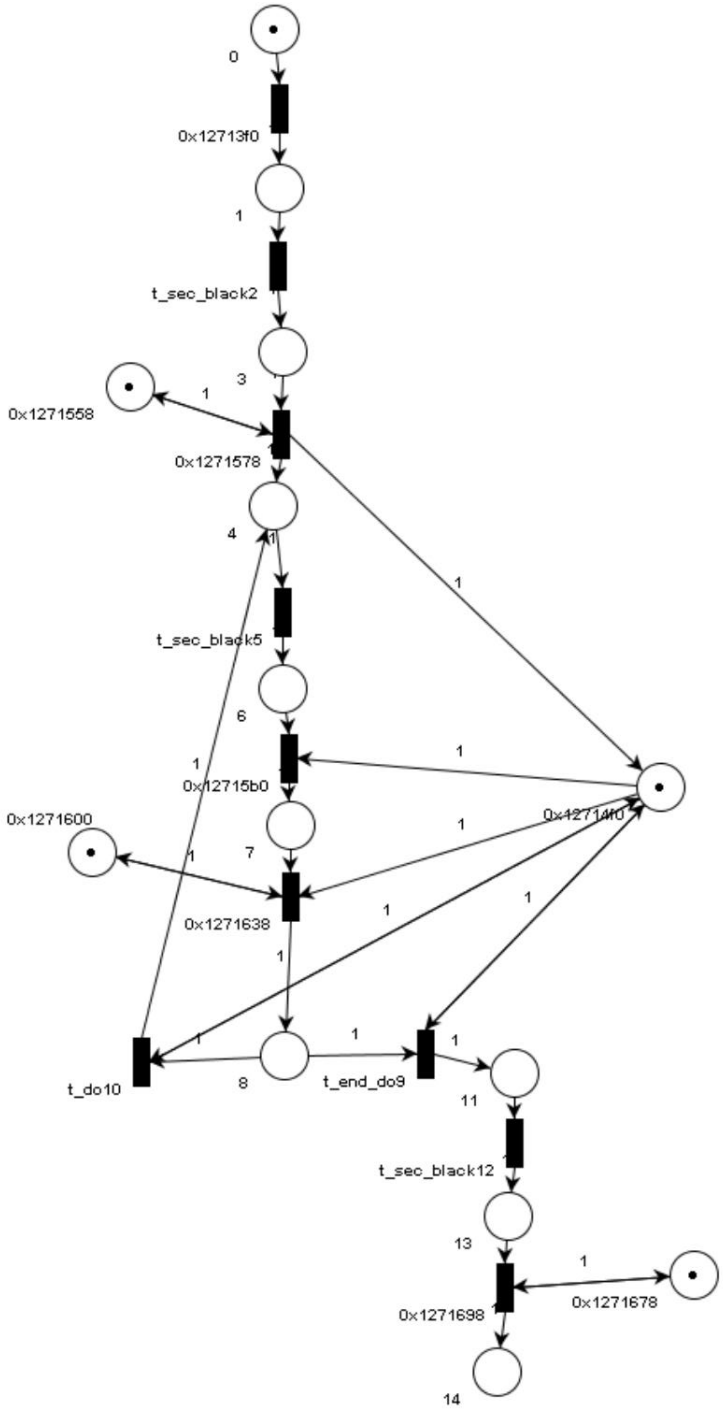

Figura 5 Visualización de red de Petri resultante de código do-while

Se observa cómo la línea 5 del Código 3 genera una transición que inicializa la variable *"i"* con el valor 0, ambos representados por un lugar en la Figura 5.

Por otro lado, la estructura de control *do-while* está formada por:

- Para la línea 7, una transición que representa la operación de aumento conectada al lugar de "i".
- Para la línea 8, una transición que cuenta con un arco de entrada desde el lugar de "i" y desde el lugar que representa el valor de "10". De esta manera se asocia la evaluación.
- Y transiciones que modelan el flujo de repetición (*t_do*) o finalización (*t_end_do*) del bucle.

Por último, una transición conectada al lugar del valor "1" representa la línea 10 del Código 3.

De esta manera, a través de la red de Petri obtenida se puede observar el flujo de ejecución del código fuente. Por ejemplo, se plasma que el bucle se ejecuta al menos una vez antes de evaluar la condición.

6.2 Segundo Caso

En esta ocasión se analiza el Código 1 (obtenido de [31]) que se ha explicado previamente en la Sección 2.4. Dicho código cuenta con una definición de variable y una estructura de control condicional donde se hace presente la vulnerabilidad TOCTOU. Después de haber procesado el fichero para obtener su AST, se construye la red de Petri que se observa en la Figura 6.

La definición de la variable *"filename"*, se transforma en una transición con un arco hacia el lugar para dicha variable, un arco de entrada desde el lugar creado para el parámetro *"argv"* y otro bidireccional que lo conecta con el lugar del índice de *argv.*

Por otra parte, la estructura de control condicional que se ve claramente reflejada en la Figura 6, se compone de los siguientes elementos:

- Una transición que representa la llamada a *access(),* conectada a los lugares de sus parámetros de entrada.
- Transiciones de control que marcan el inicio de las ramas verdadera (*t_true*) y falsa (*t_false*).
- Una transición que se genera para la llamada de *fopen ()* y que conecta con los lugares de sus parámetros de entrada. A su vez, el conjunto de estos objetos está enlazado a la transición previa que declara la variable donde se guarda el descriptor devuelto.
- Una transición para la llamada *fclose()* con un arco de entrada desde el lugar del descriptor.
- Finalmente, una transición para la llamada a *printf()* conectada al lugar que representa el *string* a imprimir por pantalla.

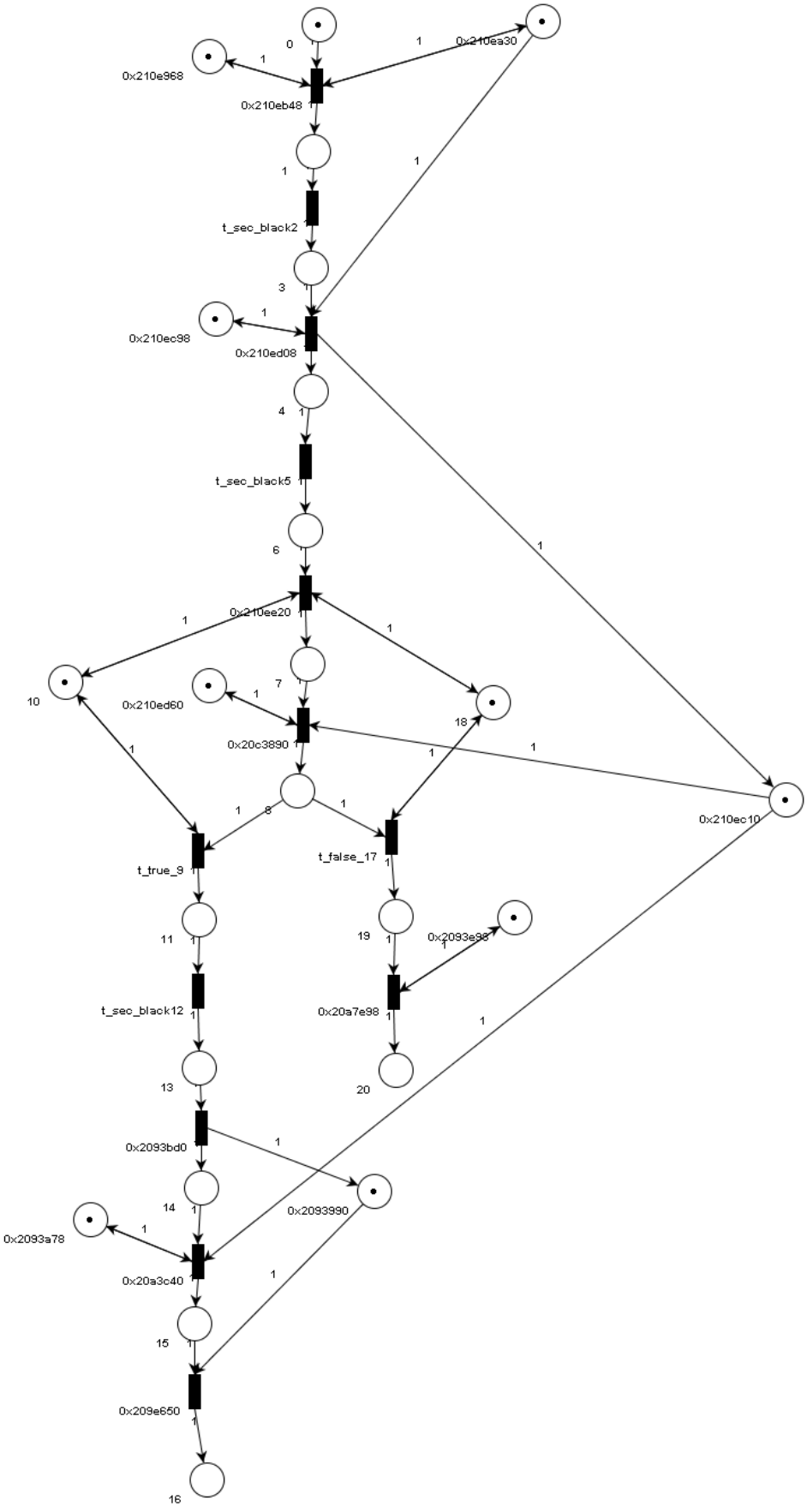

Figura 6 Visualización red de Petri resultante de código TOCTOU

Centrando la atención en las secciones que interesan para la vulnerabilidad TOCTOU, se plasma en la Figura 7 la parte de la condición donde se hace uso de *access()* para comprobar los permisos del fichero. Se está por tanto ante la parte *"time-of-check"* y la que marca el inicio de la ventana de vulnerabilidad.

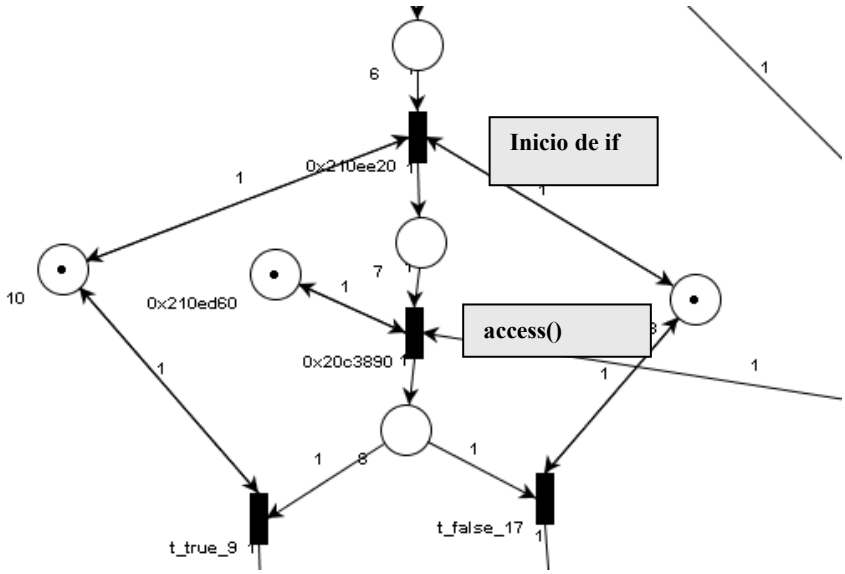

Figura 7 Sección "time of check" red de Petri TOCTOU

Por otro lado, en la Figura 8 se puede observar los dos caminos posibles que puede tomar la ejecución dependiendo de la evaluación de la condición. La rama que interesa es la verdadera ya que es donde se hace uso de la operación *fopen()*. Es aquí donde se representa el otro punto de la ventana de vulnerabilidad, el llamado *"time-of-use"*. Mediante el recorrido por las transiciones que se tienen que ejecutar para llegar a la operación *fopen()*, se puede visualizar la sección vulnerable y tener una concepción del tiempo que lleva efectuar estas operaciones.

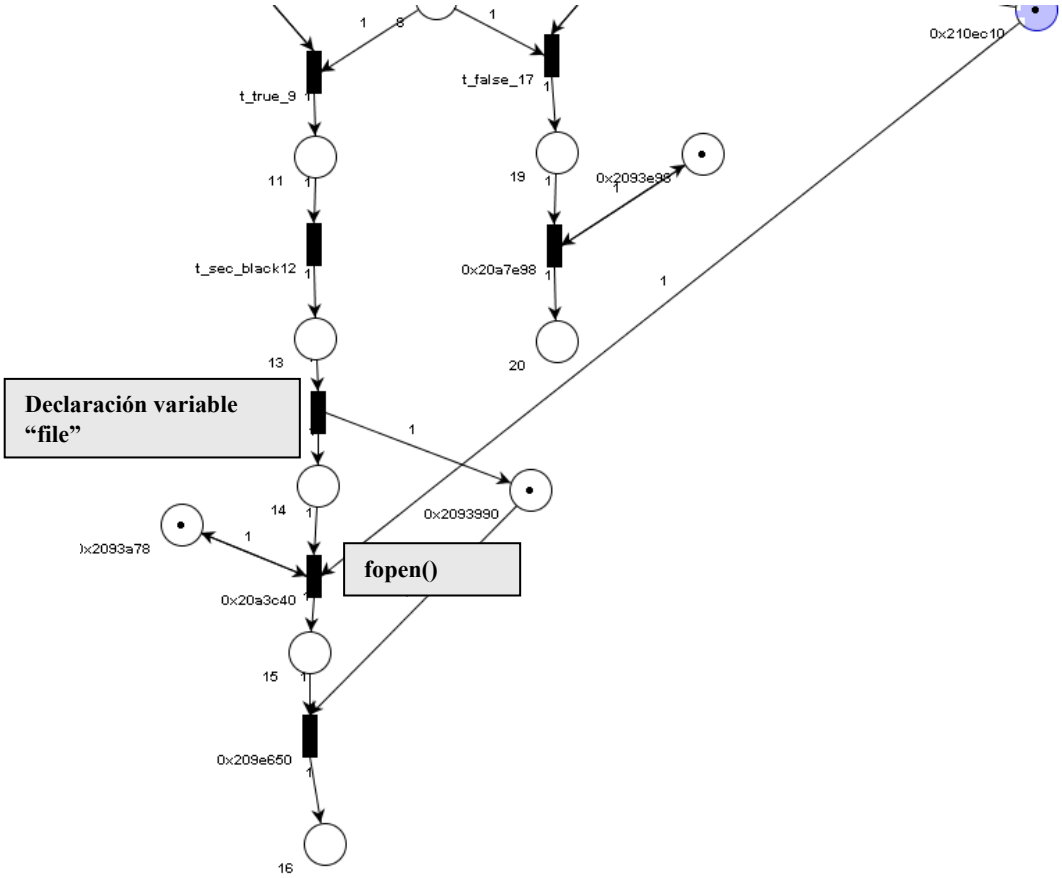

Figura 8 Sección "time of use" red de Petri TOCTOU

6.3 Tercer Caso

Para este último caso se presenta un fichero que de nuevo cuenta con una vulnerabilidad TOCTOU (adaptado de [31]), pero difiere en la función de verificación con respecto al caso de la Sección 6.2. En la línea 9 del Código 4, se observa que la función *stat()* obtiene la información del fichero a través de la variable *"filename"*. Si la operación termina con éxito se procede a validar el permiso de escritura a través de *"fileStat.st_mode"* en la línea 10. Este es el inicio de la ventana de vulnerabilidad que tiene lugar hasta que termine de ejecutarse la operación de apertura del fichero con *fopen()*.

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/stat.h>
4
5  void main(int argc, char *argv[]){
6            char *filename = argv[1];
7            struct stat fileStat;
8
9            if(stat(filename, &fileStat) = 0){ // Time of Check
10                     if(fileStat.st_mode & S_IWUSR){
11                              FILE* file = fopen(filename, "a+"); // Time of Use
12                              fclose(file);
13                     }
14                     else{
15                              printf("[+] ERROR\n");
16                     }
17           }
18           else {
19                     printf("[+] ERROR\n");
20           }
21 }
```

Código 4 Ejemplo con vulnerabilidad TOCTOU y uso de stat()

Tratado el AST obtenido de procesar el fichero con *Model.py*, se obtiene la red de Petri de la Figura 9.

Las instrucciones coincidentes con el Código 1, como la definición de la variable *"filename"* o el uso de *fopen(),* generan los mismos elementos de la red de Petri para este escenario. Por ello, no se van a volver a especificar y sólo se va a comentar la composición de las instrucciones diferentes.

- La línea 7 se compone de una transición que conecta con el lugar creado para la variable *"fileStat"*.
- La línea 9 presenta una transición para la función *stat()* unida a la transición definida para *"&"*. A su vez, está conectada a las transiciones de referencia a las variables *"fileStat"* y *"filename*".
- La línea 10 se representa con una transición para el operador *"&"* conectada a los lugares de la variable *"S_IWUSR"* y de la variable definida previamente, *"fileStat"*.

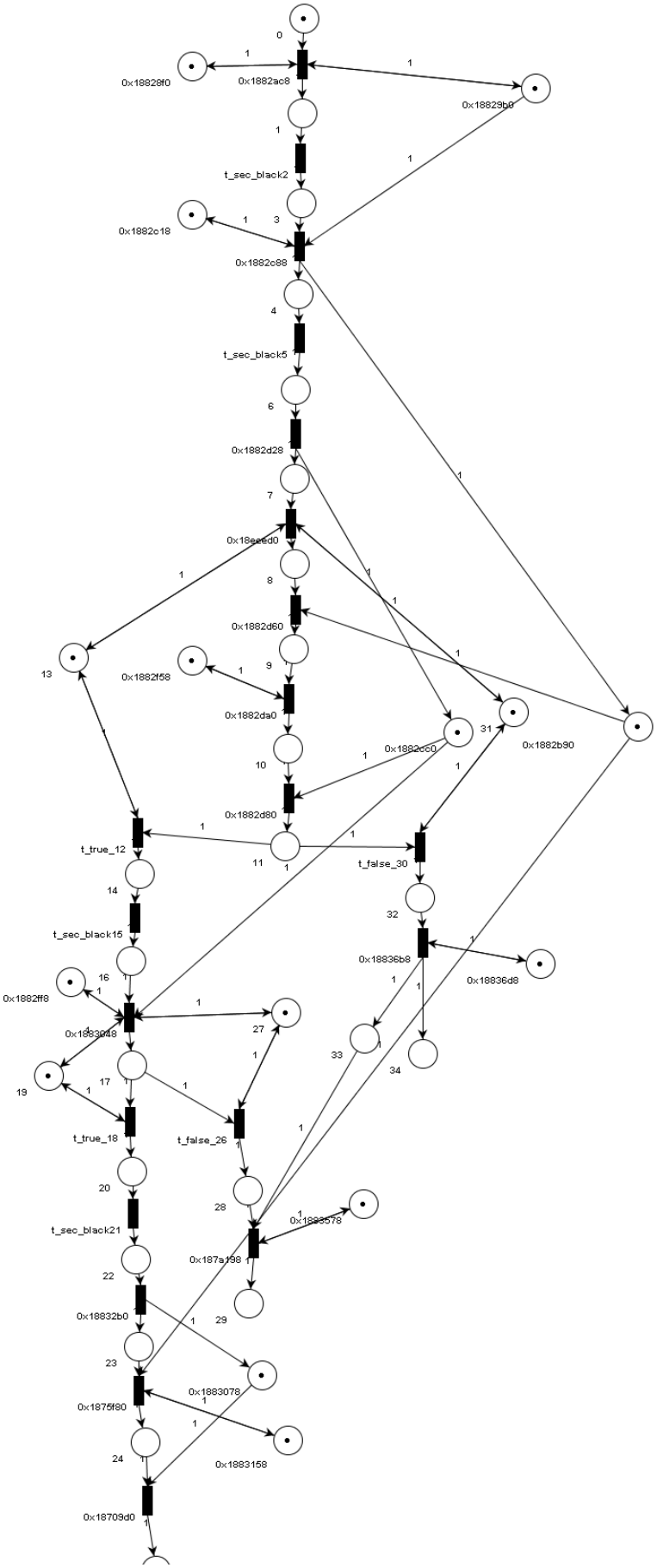

Figura 9 Visualización red de Petri resultante de TOCTOU con stat()

Ahora bien, el inicio de la ventana de vulnerabilidad (*"time-of-check"*) de TOCTOU se identifica con facilidad. Se compone del conjunto de transiciones donde se verifica la información con *stat()* y los permisos con *"fileStat.st_mode"*, como se ve en la Figura 10.

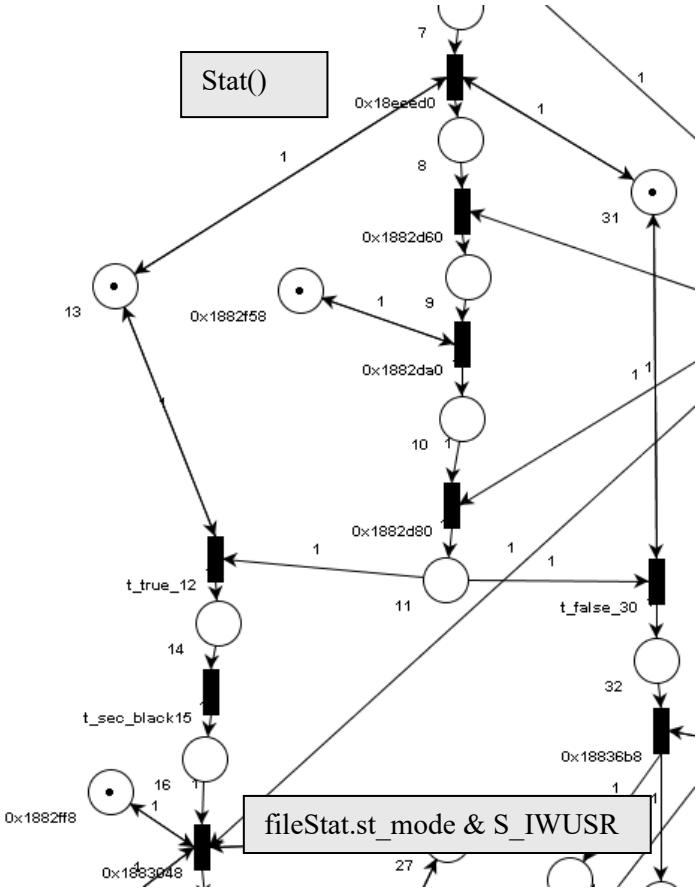

Figura 10 Sección "time of check" con stat() y fileStat.st_mode

Por otro lado, en la Figura 11 se puede observar los dos caminos que puede tomar la ejecución dependiendo de la evaluación de la condición. En la rama *true* es donde se hace uso de la operación *fopen()* por lo que se está ante el *"time-of-use"* de la ventana de vulnerabilidad. Se consigue así visualizar la sección vulnerable y tener una visión precisa del flujo de ejecución.

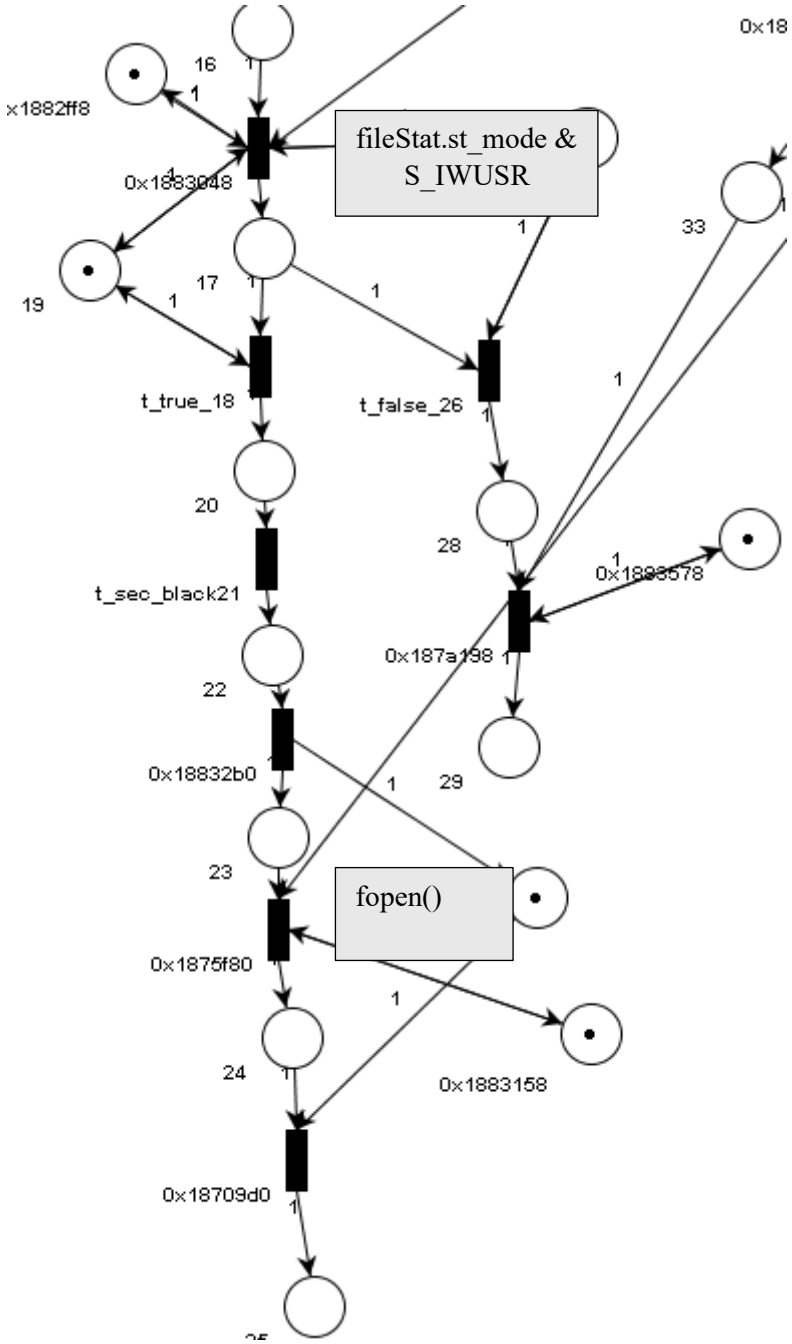

Figura 11 Sección “time of use” TOCTOU con stat()

6.4 Resultados

A la vista de los casos de uso que se han presentado puede afirmarse que la herramienta, es útil para modelar en red de Petri diferentes tipos de código fuente. Primeramente, se ha visto que para casos de códigos sin vulnerabilidades se plasma correctamente las diferentes instrucciones y relaciones entre cada elemento.

En cuanto al patrón de vulnerabilidad TOCTOU, se observa la separación de las instrucciones de verificación ("*Time-of-check"*) y uso ("*Time-of-use"*) del recurso. Las transiciones intermedias que permiten que el recurso se altere entre ambos momentos quedan adecuadamente reflejadas, lo que facilita la visualización de la ventana vulnerable.

Por ello, la representación permite analizar el origen y las causas del problema lo que abre la posibilidad de crear estrategias de mitigación, como el añadir mecanismos de sincronización o validaciones adicionales en el código.

Capítulo 7

Análisis de rendimiento

En este apartado se ha realizado un análisis de rendimiento enfocado en medir el tiempo de procesamiento a la hora de la transformación, a la vez que la escalabilidad en cuanto al tamaño de ficheros que puede procesar. Por tanto, primero se presenta el entorno y el modo en el que se ha llevado a cabo el análisis. Finalmente, se detallan los resultados obtenidos.

7.1 Definición del entorno

El entorno seleccionado para realizar estas pruebas de rendimiento consiste en un ordenador personal con un procesador Intel(R) Core (TM) i7-10510U a 2.30 GHz de 4 núcleos y 8 hilos. Cuenta con 16GB de memoria RAM y ejecuta como sistema operativo Windows 10 versión 22H2.

7.2 Ejecución

Además de medir el tiempo de ejecución, se ha tenido en cuenta también la complejidad de las estructuras a transformar. De esta manera el conjunto de pruebas que se ha establecido abarca diferentes estructuras tanto básicas como más complejas, incluyéndose también patrones de TOCTOU para comparar si tiene impacto en el tiempo de transformación.

El método se basa en registrar el tiempo de todo el proceso que incluye la transformación del AST para obtener la red y la escritura del fichero final en PNML. Para ello se ha decidido ejecutar la herramienta con cada fichero 100 veces para obtener una media de tiempo fiable.

		Tamaño archivo (líneas de código)	Tiempo medio total ejecución (s)
Complejidad baja	Declaración de variable	174	1.77
	Printf básico	224	1.78
	Get buffer	295	1.82
	Definición de variable	308	1.76
	Definición de función	308	1.77
	Struct	354	1.76
	Declaración y asignación de variable	362	1.76
	Scanf básico	370	1.79
	Definición e invocación de función	569	1.82
	Invocación de función dentro de otra	919	1.76

Complejidad alta	Bucle While	394	1.74
	Declaración y asignación de puntero	699	1.97
	Condicional If	715	1.76
	Bucle Do-while	395	1.94
	Condicional If anidado	1192	2.01
	Condicional If sin else	1240	1.76
	Bucle for	1673	1.85
	Operadores unarios y binarios anidaos	3649	1.83

Tabla 2 Tiempos de ejecución de ficheros sin TOCTOU

Tamaño archivo (líneas de código)	Tiempo medio total ejecución (s)
1181	1.83
1381	1.82
1395	1.76
1753	1.77
2707	1.85

Tabla 3 Tiempos de ejecución de ficheros con TOCTOU

7.3 Resultados

Los resultados obtenidos en las Tabla *2* y Tabla *3* muestran como el tiempo de ejecución se refleja de manera bastante consistente, lo que sugiere que la herramienta tiene un comportamiento eficiente y estable, independientemente del tamaño del archivo. Sin embargo, se nota una leve variación con respecto al aumento de la complejidad de los ficheros, pero sin un impacto significativo en cuanto al tiempo de ejecución. Por ello, se puede afirmar que sigue manteniendo estabilidad sin afectar gravemente el rendimiento.

La herramienta presenta un comportamiento eficiente, capaz de manejar archivos de diferentes complejidades incluyendo TOCTOU, sin grandes variaciones temporales. Se confirma así que la herramienta es adecuada para transformar a una red de Petri, código fuente. No obstante, para considerar la escalabilidad en cuanto a tamaño y complejidades grande, tendría que realizarse pruebas con ficheros de grandes tamaños y dependencias.

Capítulo 8

Conclusiones y trabajo a futuro

El trabajo realizado ha conseguido cumplir el objetivo planteado al iniciar este proyecto, que consiste en la transformación de código fuente en redes de Petri. Se ha comprobado que la herramienta puede modelar, de manera clara y precisa, código fuente con vulnerabilidad TOCTOU en una red de Petri.

Los resultados obtenidos demuestran que la herramienta es capaz de modelar el flujo de ejecución del código que a veces resulta difícil de visualizar con métodos tradicionales. Presenta eficiencia y consistencia en el tiempo de ejecución independientemente del tamaño y complejidad del fichero.

A pesar de lo logrado, existen varias maneras en las que expandir y mejorar la herramienta en futuros desarrollos. Una de las mejoras consiste en incorporar un algoritmo de dispersión para asegurar una mejor distribución de los elementos de la red de Petri generada en la herramienta de visualización.

Otra manera de expandir la herramienta es la incorporación de la detección de otras vulnerabilidades, así como la ampliación a otros lenguajes de código fuente de entrada y de otros formatos de salida.

Por último, en lo relativo a la interacción con el usuario final, una posible mejora es la creación de una interfaz gráfica en la cual tenga lugar todo el proceso y que ofrezca un entorno de visualización, simulación y análisis.

Bibliografía

[1] N. Kolesnikov, “50 Estadísticas Clave de Ciberseguridad para Noviembre de 2024”. techopedia.com. https://www.techopedia.com/es/estadisticas-ciberseguridad (Accedido el 10 de noviembre de 2024).

[2] M. Jiménez, “Ataques cibernéticos: causas, tipos y consecuencias”. piranirisk.com. https://www.piranirisk.com/es/blog/ataques-ciberneticos-causas-y-consecuencias (Accedido el 10 de noviembre de 2024).

[3] R.Trifonov, O.Nakov, G.Pavlova, S. Manolov, G. Tshochev, P. Nakov, “Analysis of the Principles and Criteria for Secure Software Development”, 2020 28th National Conference with International Participation (TELECOM), Sofia, Bulgaria, 2020. Accedido el 12 de noviembre de 2024. [Online]. Disponible: doi: 10.1109/TELECOM50385.2020.9299567.

[4] UML. “What is UML”, uml.org. https://www.uml.org/resource-hub.htm (Accedido el 27 de octubre de 2024).

[5] D. Bell, “The UML2 class diagram”, developer.ibm.com. https://developer.ibm.com/articles/the-class-diagram/ (Accedido el 27 de octubre de 2024).

[6] IBM, “Activity diagrams”, ibm.com. https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-activity (Accedido el 27 de noviembre de 2024).

[7] “Qué es Código Fuente”, arimetrics.com. https://www.arimetrics.com/glosario-digital/codigo-fuente (Accedido el 7 de Noviembre 2024).

[8] P. Thomson, “Static Analysis: An Introduction: The fundamental challenge of software engineering is one of complexity”, *Queue,* vol. 19, p.13. Jul-Aug 2021. Accedido: el 6 de noviembre de 2024. doi: 10.1145/3487019.3487021. [Online]. Disponible: https://doi.org/10.1145/3487019.3487021.

[9] T. Ball, “The concept of dynamic análisis”. *ACM SIGSOFT Software Engineering Notes*, 1999, vol. 24, no 6, p. 216-234. Accedido: el 6 de noviembre de 2024. doi: 10.1145/318774.318944. [Online]. https://dl.acm.org/doi/pdf/10.1145/318774.318944

[10] StudySmarter, “Verificación formal.”, studysmarter.es. https://www.studysmarter.es/resumenes/matematicas/logica-y-fundamentos/verificacion-formal/ (Accedido el 6 de noviembre de 2024).

[11] D.Silva, P. Samarasekara, R. Hettiarachchi, “A comparative Analysis of Static and Dynamic Code Analysis Techniques”, Sri Lanka Institute of Information Technology. (SLIIT), Sri Lanka, CC-BY 4.9, 2020. Accedido: el 14 de noviembre de 2024. [Online]. Disponible: doi: 10.36227/techrxiv.22810664.v1.

[12] P. Bjesse, “What is formal verification?”, *SIGDA Newsl*, vol. 35, no 24, Dec. 2005. Accedido: el 14 de noviembre de 2024. doi: 10.1145/1113792.1113794. [Online]. Disponible: https://doi.org/10.1145/1113792.1113794.

[13] Massachusetts Institute of Technology, “Reading17: Concurreny”, web.mit.edu. https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/ (Accedido el 6 de noviembre de 2024).

[14] T. Murata, "Petri nets: Properties, analysis and applications," IEEE, vol. 77, no. 4, pp. 541-580, April 1989, doi: 10.1109/5.24143.

[15] C. A. Petri, *Communication with automata*. Rome Air Development Centre,1966. Accedido: el 23 de noviembre de 2024. [Online]. Disponible: https://apps.dtic.mil/sti/tr/pdf/AD0630125.pdf

[16] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis. *Modelling with Generalized Stochastic Pet.* England, Wiley,1995.

[17] Imperva, “Race Condition”, imperva.com. https://www.imperva.com/learn/application-security/race-condition/#:~:text=A%20race%20condition%20vulnerability%20is,to%20unexpected%20and%20erroneous%20outcomes (Accedido 5 de noviembre 2024).

[18] Twingate, “What is Time-Of.Check to Time-of-Use?”, twingate.com. https://www.twingate.com/blog/glossary/time-of-check-to-time-of-use-attack (Accedido el 5 de noviembre de 2024).

[19] R. Dewhurst, “Static Code Analysis”, owasp.org. https://owasp.org/www-community/controls/Static_Code_Analysis (Accedido el 14 de noviembre de 2024).

[20] A., “Herramientas de prueba de seguridad de aplicaciones”, a2secure.com. https://www.a2secure.com/blog/herramientas-de-prueba-de-seguridad-de-aplicaciones-ast/ (Accedido el 14 de noviembre de 2024).

[21] M.Rawson, M. G.Rawson, *Petri Nets for Concurrent Programming* . Arxiv, 2022. Accedido: 7 noviembre 2024. [Online]. Disponible: https://arxiv.org/pdf/2208.02900

[22] O.M.Dahl, S. D. Wolthusen, “Modeling and Execution of Complex Attack Scenarios using Interval Timed Colored Petri Nets”, IEEE Computer Society , Norway, 2006.

[23] J. Voron, F. Kordon, “Evinrude: A tool to Automatically Transform Program's Sources into Petri Nets”, ResearchGate, París, 2008.

[24] P.J.Gallardo, “Los lenguajes de programación más demandados en 2024”, hackaboss.com. https://www.hackaboss.com/blog/lenguajes-programacion-mas-demandados (Accedido el 23 de noviembre de 2024).

[25] Universidad de la Laguna, “Árboles de Análisis Abstracto”, ull-esit-gradoii-pl.github.io. https://ull-esit-gradoii-pl.github.io/temas/syntax-analysis/ast.html (Accedido el 26 de octubre de 2024).

[26] Json, “Introducción a JSON”, json.org. *https://www.json.org/json-es.html* (Accedido el 26 de octubre 2024).

[27] Pnml, “Welcome on pnml.org”, pnml.org. https://www.pnml.org/index.php (Accedido el 27 de octubre 2024).

[28] P. Bonet, C.M. Llado, R. Puijaner and W.J. Knottenbelt, “PIPE v2.5: A Petri Net Tool for Performance Modelling” , Proc. 23rd Latin American Conference on Informatics (CLEI 2007), Costa Rica, Oct. 2007. Accedido: el 26 de octubre de 2024. [Online]. Disponible: https://www.doc.ic.ac.uk/~wjk/publications/bonet-llado-knottenbelt-puijaner-clei-2007.pdf

[29] N.J. Dingle, W.J. Knottenbelt and T. Suto, “PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets”, ACM SIGMETRICS Performance Evaluation Review (Special Issue on Tools for Computer Performance Modelling and Reliability Analysis), London, vol. 36(4), March 2009, pp. 34-39. Accedido: el 26 de octubre de 2024. [Online]. Disponible: https://www.doc.ic.ac.uk/~wjk/publications/dingle-knottenbelt-suto-per-2009.pdf

[30] R. Raducu, R. J.Rodríguez, P.Álvarez. “Model-Based Analysis of Race Condition Vulnerabilities in Source Code”, Universidad de Zaragoza, España, JNIC 2022.

[31] S. Rea, 2023, TFG-ModelBasedAnalysis, [SourceCode]. https://github.com/SalomeReav/TFG-ModelBasedAnalysis

[32] K. Ayeva, S. Kasampalis, *Mastering Python Design Patterns: A Guide to Creating Smart, Efficient, and Reusable Software*. Alemania: Packt Publishing, 2018. Accedido: el 28 octubre de 2024. [Online]. Disponible: https://www.google.es/books/edition/Mastering_Python_Design_Patterns/PshsDwAAQBAJ?hl=es&gbpv=1&kptab=overview

[33] M. Granda. “Redes de petri: definición, formalización y ejecución”. ctr.unican.es. https://www.ctr.unican.es/asignaturas/mc_procon/Doc/PETRI_1.pdf (Accedido el 26 de octubre de 2024).

Anexo

Tiempo invertido y diagrama de Gantt

El diagrama de Gantt en la Figura 12 refleja las diferentes fases del proyecto y la dedicación en el tiempo invertidos. Como se puede observar, se comienza con un periodo de investigación donde se adquiere conocimientos previos necesarios. Seguidamente comienza la etapa de diseño e implementación de la herramienta desarrollada. Finalmente, el periodo de elaboración de la memoria. Por otro lado, la Tabla 4 detalla el tiempo invertido en cada una de estas fases.

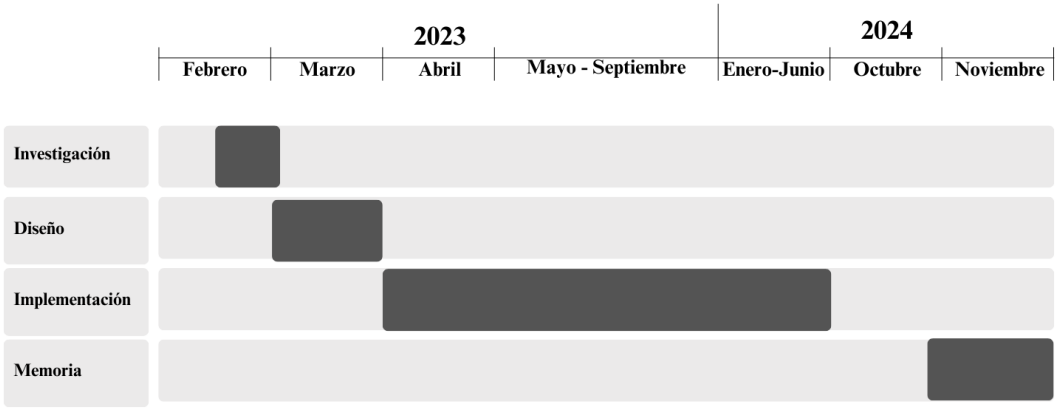

Figura 12 Diagrama de Gantt

Fase	Horas
Investigación	39
Diseño	20
Implementación	210
Reuniones	9
Memoria	100
Total	378

Tabla 4 Horas invertidas