UNIVERSIDAD DE DEUSTO

DETECCIÓN DE ERRORES EN SOFTWARE MEDIANTE VERIFICACIÓN DE MODELOS

IVÁN GARCÍA FERREIRA

Bilbao, Junio de 2015

UNIVERSIDAD DE DEUSTO

DETECCIÓN DE ERRORES EN SOFTWARE MEDIANTE VERIFICACIÓN DE MODELOS

Tesis doctoral presentada por
IVÁN GARCÍA FERREIRA
dentro del Programa de Doctorado en
INFORMÁTICA Y TELECOMUNICACIÓN

Dirigida por el
Dr. IGOR SANTOS GRUEIRO
y por el
Dr. PABLO GARCÍA BRINGAS

Autor | Co-director | Co-director

Bilbao, Junio de 2015

A toda mi familia, en especial a mi abuelo Ángel, que me contagió su pasión por la lectura y el auto-aprendizaje que tanto me han ayudado.

Abstract

The incorporation of computers and the Internet in our lives has facilitated many processes that used to take hours, days or even months, being accomplished in less time, improving our lives in many ways. But it is prone to misuse.

Attacks against countries, espionage, invasions, and many other war-flavoured events are only the tip of the iceberg of all that happens on the Internet. Many of these attacks are aimed at known programs that have some type of vulnerability, causing them to be the gateway to any system or network.

Because of this, the search for vulnerabilities is becoming a new profession, highly paid and valued by both nations and companies on the one hand, and by mafias and black markets on the other hand.

For this reason, this dissertation seeks to improve the processes of finding software bugs, and hence the following hypothesis is formulated: *«There exists a method of representation of the memory that is able to detect known errors, both on the heap and stack»*.

In this context, this work focuses on a new algorithm to find vulnerabilities using model checking to analyse binaries. Based on our notation is possible to create formulas that represent errors and make the algorithm to return the possible execution flows that would expose the system to the discovered vulnerabilities.

In addition, our algorithm is developed in a language that validates the generated code, Coq, thus ensuring that the code itself has no errors.

The validation of the algorithm and the code has been verified in two ways: validating vulnerabilities that have been already found by other security experts and finding new vulnerabilities that have not been yet discovered.

Finally, through this research, it has been released a framework to find vulnerabilities in binaries, which will hopefully be able to help the community to develop more secure software.

Resumen

La introducción del ordenador e Internet en nuestras vidas ha hecho que muchos procesos que antes costaban horas, días, incluso meses, ahora puedan ser llevadas a cabo en mucho menos tiempo, mejorando nuestras vidas en numerosos aspectos. Como ha sucedido tantas veces en la historia, cualquier invento puede acabar siendo utilizado con fines maliciosos.

En Internet también se llevan a cabo ataques a estados, espionaje, invasiones, y otros aspectos tan característicos de cualquier guerra. Muchos de estos ataques están dirigidos a programas conocidos que presentan algún tipo de vulnerabilidad, haciendo que éstos sean la puerta de entrada a cualquier sistema o red.

A causa de esto, la búsqueda de vulnerabilidades se está convirtiendo en una nueva profesión, muy bien pagada y valorada tanto por naciones y empresas, por un lado, como por mafias y un gran mercado negro, por el otro.

Por esta razón, la presente tesis doctoral busca mejorar los procesos de encontrar errores en el software, y por esta razón se formula la siguiente hipótesis: «*Existe un método de representación de la memoria que es capaz de detectar errores conocidos, tanto en el heap como en la pila*».

En este contexto, nos centramos en crear un nuevo algoritmo para buscar vulnerabilidades utilizando *model checking* para el análisis de binarios. Nuestra notación, facilita la creación de fórmulas que representen errores

y el algoritmo devuelve los posibles flujos de ejecución que provocarían que una vulnerabilidad se llevase a cabo.

Además, la programación de nuestro algoritmo se ha realizado en un lenguaje que valida el código generado, Coq, asegurando de esta forma que el propio código que busca errores no contenga ninguno en sí mismo.

La validación del algoritmo y del código ha sido corroborada de dos modos: i) encontrando nuevas vulnerabilidades que no habían sido descubiertas y ii) comprobando código inseguro que ya ha sido publicado por otros expertos en seguridad.

Finalmente, tras este estudio, se ha liberado una plataforma para la búsqueda de vulnerabilidades que puede ayudar a la comunidad a realizar un software más seguro.

Agradecimientos

Esta tesis no hubiera sido posible sin la colaboración de muchas personas que me han ayudado o apoyado durante todo este trayecto, y a las que me gustaría agradecer.

En primer lugar quisiera dar mi agradecimiento a mi director de tesis Pablo García Bringas, por darme la oportunidad de trabajar en el S3Lab, lo cual ha cambiado mi vida por completo, y por esas charlas sobre el Señor de los anillos que solo él me seguía.

A mi amigo y también director de tesis, Igor Santos. Está claro que sin tus consejos, reprimendas y los apuntes en cuadernos no hubiera llegado tan lejos. Conservaré esos «memes» para siempre.

Tampoco puedo dejar pasar la oportunidad de agradecer a DEIKER y a Deustotech por haber depositado la confianza en mi estos años y hacer que esta tesis se hiciera posible.

Lógicamente tengo que agradecer a todo el S3Lab (los que están y los que han estado) por todos los grandes momentos pasados dentro y fuera del laboratorio, la ayuda prestada, los ánimos y la amistad que todos me habéis dado. No os nombro a todos, pero de cada uno de vosotros me llevo algo positivo.

Querría hacer una reseña especia dentro del laboratorio a mi *binary team*, Xabi e Iskander, por ser los únicos que comprendíais mi locura. También a Iker, Jorge y Sete, por todas las risas que hemos pasado dentro y fuera del laboratorio.

A mi «cuadrilla», Roberto, Iñigo y Javi, que habéis estado pendientes de mi progreso día a día, demostrando que siempre estáis dispuestos a ayudarme y a apoyarme. Espero que nuestros jueves sigan por muchos años.

Finalmente, no puedo dejarme a los que son los cimientos de toda mi vida, mi familia, que siempre han estado a mi lado. Especialmente a mis padres que tanto me han apoyado estos años y comprendían que no tenía mucho tiempo para ir a verles, llenaría esta tesis solo con cada cosa que tengo que agradecerles. A mi hermano, todo lo que diga sobre él es poco, ha estado ahí en mis momentos más difíciles y siempre es mi gran apoyo. Y no puedo dejarme a mis tíos: Jose, mi segundo hermano, gracias por dejarme meter horas con el Spectrum cuando sólo tenía seis años; y Ángel, que me entregó aquella libreta de apuntes de MS-DOS, lo que desató mi amor por las pantallas negras y las lineas de comandos. Está claro que vosotros iniciasteis el proceso que desembocó en esta tesis.

Índice general

Índice de figuras

Índice de tablas

«Al pasar por un periodo difícil, recuerda: aunque hayas perdido grandes batallas, has sobrevivido y estás aquí. Eso es una victoria. Demuestra tu alegría y demuestra tu capacidad para seguir adelante.»

Paulo Coelho
El manuscrito encontrado en Accra

«En ciertos casos, aprendemos más buscando la respuesta a una pregunta y no hallándola que conociendo esa respuesta.»

Lloyd Alexander (1924–2007)

CAPÍTULO

1

Introducción

Los errores en el software se han convertido en un grave riesgo en muchos sectores de nuestro mundo. Esto se debe principalmente a la implantación de programas informáticos dentro de casi todos los elementos de nuestra sociedad. Actualmente, este problema se está extendiendo incluso más debido al crecimiento de lo que se ha denominado *Internet of things*, lo que ha provocado que los errores en el software se estén generalizando. Evitar fallos en partes tan importantes como en el software de los automóviles, aviones, o cualquier otro elemento crítico puede hacer que se salven vidas y grandes cantidades de dinero.

Detectar estos errores antes de que el software llegue a los usuarios finales es uno de los principales problemas a intentar resolver dentro del desarrollo del software. Por ello, diversos grupos de investigadores se dedican a crear algoritmos que sean capaces de detectar estos fallos, tanto en el código fuente, en tiempo de compilación, como que el fichero binario ya halla sido creado.

El primer error informático del que se tiene constancia ocurrió en el año 1945, cuando Grace Hopper descubrió que el mal fun-

cionamiento del ordenador Mark II era debido a que una polilla se había introducido entre los componentes del ordenador. Por esta razón ella apunto en su bitácora «*First actual case of bug being found*» (primer caso de insecto que se encuentra)[1]. Desde de ese momento, se acuñó el término *bug* para referirse a los errores informáticos. En aquella época, los servidores ocupaban habitaciones enteras, y las labores de programación y depuración del código eran unas tareas muy complicadas, aunque eso fue cambiando poco a poco con el paso del tiempo.

Un gran cambio para la informática se produjo en 1976. En este año, Steve Wozniak creo el primer ordenador personal. Este hecho hizo mucho más sencilla la incorporación de estas máquinas en nuestras vidas, y con el paso del tiempo cada vez era más fácil encontrar ordenadores personales en las casas. En gran parte este cambio fue debido a que el precio era más asequible para un hogar medio y porque su tamaño era muy pequeño en comparación con lo que había en aquel momento. A su vez, en la segunda mitad del siglo XX también han mejorado ostensiblemente las técnicas de programación, con lenguajes mucho más fáciles para el desarrollador, tanto para programar como para depurar. Estos dos acontecimientos han hecho que se hayan incrementado mucho en los últimos años la cantidad de usos tanto de los ordenadores como de las aplicaciones que se programan para éstos.

Por tanto, cualquier error informático que se pueda producir tiene una gran relevancia en nuestras vidas. Mucha culpa de este aumento de interés se ha debido a los *crackers*, que se aprovechan de ellos para penetrar en sistemas; o a que los *virus* utilizan estos fallos para infectar máquinas, con grandes perdidas de horas de trabajo en empresas e incluso, en el peor de los casos, porque pueden llegar a costar vidas humanas.

[1] http://www.history.navy.mil/photos/images/h96000/h96566kc.htm

1.1 Motivación

Como hemos visto, la informática se ha ido introduciendo en nuestra sociedad para ir quedándose poco a poco. Esto ha hecho que a día de hoy todos tengamos un ordenador en casa y gran parte de nuestras vidas sea controlada por ellos. Son los ordenadores quienes controlan el tráfico, (ya sea de coches, mediante semáforos; de trenes, manejando los cambios de vías; de aviones, controlando despegues y aterrizajes, etc.), las transacciones bancarias, las herramientas médicas para realizar operaciones, y muchas más aplicaciones que, aunque no lo lleguemos a apreciar, están continuamente funcionando y prestándonos servicios.

Al tener tantas partes de nuestra sociedad controlada por estos ordenadores, es vital que no se produzcan errores que puedan ocasionar riesgos para la población. Por ello, realizar verificación del código con el que se desarrollan los programas, es una parte esencial del ciclo de vida del software, más aun si pensamos en sistemas críticos, como por ejemplo el control de vuelo de un avión.

Hoy en día, existen algoritmos preparados para verificar el código fuente de la aplicación que se está desarrollando, pero no cuentan con un método para revisar el ensamblador que sale después de ser optimizado por el compilador. Éste introduce optimizaciones y, como cualquier otro programa, también puede contener fallos. Un error que es producido por el compilador no se detecta hasta etapas muy finales del desarrollo del software. Este tipo de problemas solo es detectado si la empresa que está programando tiene un equipo de pruebas haciendo comprobaciones continuamente sobre cada parte del código que se va desarrollando. Aunque en muchas ocasiones este tipo de comprobaciones ha demostrado ser muy eficaz, el factor humano puede hacer que estos errores lleguen hasta el usuario final. Además el coste de dicho equipo supone una gran cantidad de dinero a las empresas que deciden hacer este tipo de comprobación. Los sueldos de los expertos en seguridad son muy costosos para las empresas, debido al alto nivel de especialización que tienen, y a los pocos expertos que existen en el mundo.

1.2 Situación actual

Los errores en el software pueden suponer un peligro a la seguridad de los usuarios, tanto en el mundo físico como en el digital. Los atacantes suelen usar estos fallos para diversos fines, como por ejemplo extender la propagación de un virus, conseguir nuevos ordenadores para una *botnet*[2] o enviar grandes cantidades de *spam*[3]. Con esto, se ha conseguido despertar un interés en los principales sistemas operativos del mercado e incluso en fabricantes de hardware, que se han preocupado por la seguridad haciendo productos más seguros.

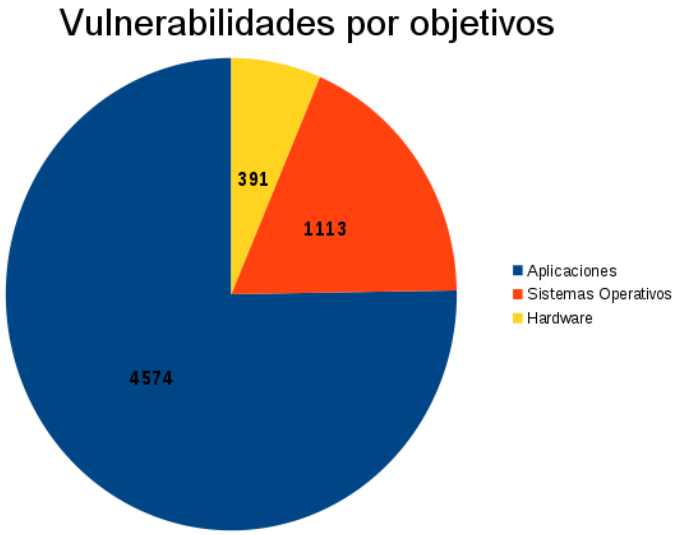

Figura 1.1: Los objetivos de la vulnerabilidades según Cristian Florian en GFI[4].

Como se puede observar en la Figura 1.1, son muy pocos los fallos que se producen en hardware o en los sistemas operativos. De un 75 % de las vulnerabilidades que aparecen, éstas se dan para aplicaciones creadas por terceros. Aunque el sistema operativo y

[2]Red de ordenadores controlados por un ordenador central, y que son utilizadas para realizar tareas conjuntas.

[3]Correo no deseado

[4]Adaptada de la página web: http://www.gfi.com/blog/top-vulnerable-applications-operating-systems-2010/

el hardware intenten proteger este software, es muy difícil que se pueda salvaguardar completamente.

En muchos casos, estas aplicaciones son usadas por miles de personas, lo que pone en peligro la seguridad de muchos usuarios. Un ejemplo claro de este tipo de vulnerabilidades son las que suelen ocurrir en el software que se ejecuta sobre Microsoft Windows. Aunque las protecciones de este sistema operativo han mejorado mucho la mala programación de los programas que corren sobre esta plataforma han ocasionado múltiples problemas de seguridad.

1.3 Un problema en busca de solución

Los errores en el software son un problema que todavía no tiene una solución clara, pero es un asunto por el cual las empresas pueden llegar a perder cantidades importantes de dinero cuando no se llegan a detectar a tiempo. Los problemas que pueden llegar a ocasionar los fallos en el software son varios. Pasamos a describirlos a continuación.

El primer problema es la seguridad de la persona que esté utilizando el software. Un usuario que posea un programa con vulnerabilidades está expuesto al ataque de cualquier atacante malicioso, o *malware*. Las personas que se benefician de esto pueden llegar a realizar varios tipos de ataques. Desde un simple ataque de denegación de servicio hasta la instalación de un troyano que robe toda la información del usuario, pasando por convertir el ordenador del usuario en un zombie[5] para realizar un ataque DDOS (*Distributed Denial Of Service*).

Otro problema que se da por los errores en el software es el mal funcionamiento del programa. Muchas veces, la vulnerabilidad descubierta no es explotable, pero también es posible que simplemente produzca que el programa se cierre, haciendo que el usuario normal no pueda trabajar. Esta opción puede parecer poco preocupante si pensamos en los programas que tenemos en los ordenadores de casa, pero hay que tener en cuenta las infraestructuras críticas (como

[5] Ordenador controlado totalmente por un usuario malicioso

por ejemplo hospitales). Se podrían nombrar muchos ejemplos en los cuales un fallo en el software ha repercutido en grandes problemas, pero creo que los ejemplos más claros de este tipo de errores los pudimos observar en la explosión del cohete Ariadne 5[6] y en el fallo de la de la navegación de los misiles Patriot que provoco decenas de muertos[7].

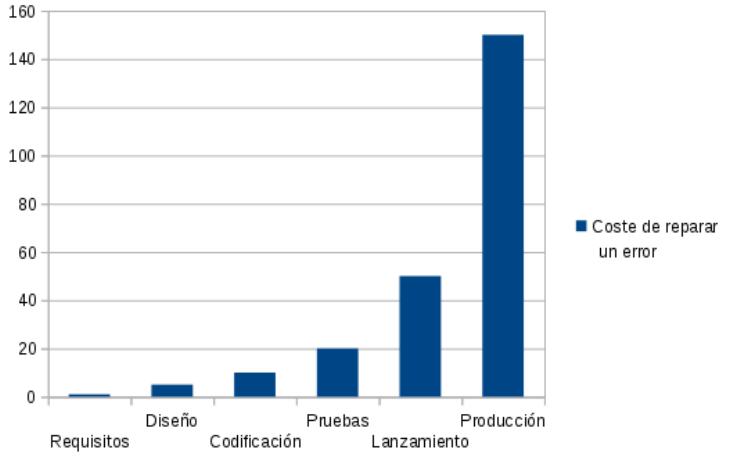

Figura 1.2: Coste de reparar un fallo en el software según Barry Boehm[8].

El tercer problema, es el aspecto económico. Como podemos observar en la Figura 1.2, el coste de la reparación de un error cuando ya ha salido a la calle es mucho más caro que el coste que tiene encontrar el fallo en cualquier momento de la producción del software. Estos costes pueden venir por diversos motivos, pero en ellos no se tienen en cuenta la cantidad de clientes que se pueden perder si un fallo en el software les impide trabajar. En este caso, además de pérdidas económicas, la empresa tendría una perdida de credibilidad que podría hacer que la gente no volviera a confiar en el producto.

[6]http://www.ima.umn.edu/ãrnold/disasters/ariane5rep.html

[7]http://sydney.edu.au/engineering/it/ alum/patriot_bug.html

[8]Gráfico adaptado de la página web: http://es.scribd.com/doc/20893084/Advanced-OOP-and-Design-Patterns

En la figura anterior, también podemos observar como los costes de arreglar un error en el software van aumentando según se va avanzando en las etapas de desarrollo. Como vemos, corregir una vulnerabilidad una vez que el producto está lanzado es hasta 150 veces más costoso que hacerlo durante las primeras fases del proyecto.

Además de estos problemas, el mercado de las vulnerabilidades hace que la búsqueda de fallos en el software se incremente mucho más. El incremento del valor por fallos en aplicaciones informáticas, ha conseguido que se desarrolle un mercado que llega a pagar 250.000 dólares por por cada error que se descubre en el sistema operativo IOS. Este dato lo podemos observar en la Figura 1.3 donde hemos adaptado un gráfico mostrado por Forbes[9] donde se revelaban los precios pagados por vulnerabilidades en una entrevista a un experto en seguridad. Las empresas se están dedicando a crear competiciones entre grupos (creando un sistema de puntos) para conseguir que los investigadores encuentren más errores.

Existen empresas que incluso ofrecen ventajas para aquellos que descubran más vulnerabilidades en un periodo de tiempo.

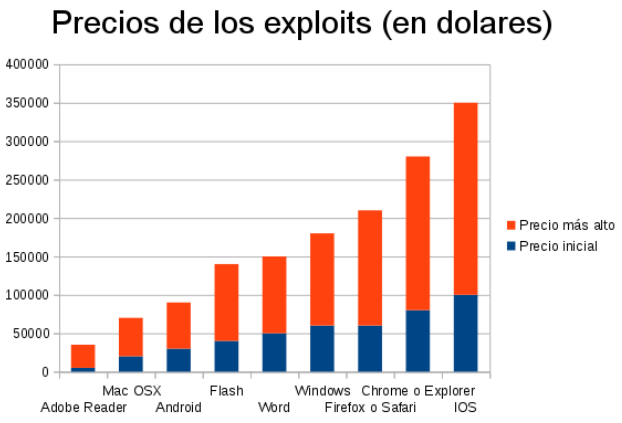

Figura 1.3: Precios ofrecidos por las vulnerabilidades.

Además contamos con el mercado negro, que compra vulnerabi-

[9] http://www.forbes.com/sites/andygreenberg/2012/03/23/shopping-for-zero-days-an-price-list-for-hackers-secret-software-exploits/

lidades para la dispersión de virus o troyanos bancarios, con lo que los precios podrían llegar a dispararse mucho más.

1.4 Zero-days

Nicolas Waisman [Wai08] realizó un estudio en 2008, en el cual saco a la luz la verdad sobre las vulnerabilidades descubiertas y los parches realizados en las empresas dedicadas al desarrollo del software.

Uno de los detalles más sorprendentes de la investigación de Waisman fue el tiempo que tardan las empresas en parchear el software una vez que se ha detectado una vulnerabilidad y está ha sido enviada por el investigador. En la Tabla 1.1 podemos observar los tiempos que se mantienen en activo las vulnerabilidades. Como se puede ver el ciclo de vida es enorme, y si además pensamos que a la empresa le cuesta mucho más en términos monetarios, nos podemos dar cuenta del gran problema que causan los errores en el software.

Tabla 1.1: Duración en días de la reparación de los errores en el software.

Duración de los Zero-days	Días
Corta	90
Medía	348
Larga	1080

1.5 Software malicioso

También conocido como *malware* (del inglés *malicious software*), este tipo de programa tiene mucho que ver con la seguridad en sistemas informáticos. Este tipo de programas se aprovechan de las

vulnerabilidades en aplicaciones informáticas para conseguir extenderse e infectar redes enteras. Si no existieran errores en el software sería suficiente con educar a los usuarios para no ejecutar nada sospechoso. Así ningún usuario resultaría infectado, pero al aprovecharse de estos fallos, en algunos casos, dicho *malware* no necesita interacción con el usuario en ningún momento.

El gusano Conficker[10] fue capaz de infectar un 5 % de los ordenadores del mundo (entre 600.000 y 700.000) en el año 2009, y esta gran infección fue debida a un error en la programación del servicio `Server` de Windows[11].

La rápida propagación de este gusano fue debido a un error en el servicio RPC de Windows[12]que afectaba a todas las versiones de XP y superiores. Muy poca gente había parcheado esta vulnerabilidad, y el gusano sólo tenía que explotarla para conseguir plenos accesos al ordenador atacado.

1.6 **Ciberguerra**

En los últimos se años está hablando mucho de la *ciberguerra*. Este término también tiene mucho que ver con las vulnerabilidades en el software. Los atacantes de otros países estudian el software utilizado por su enemigo, con lo cual se pueden preparar ataques que van enfocados sólo a destruir cierta parte vital de la infraestructura de éste.

Un ejemplo de este ataque lo tenemos en otro *malware*, Stuxnet[13]. Stuxnet fue creado para entrar en las centrales nucleares de Irán, aprovechándose de un fallo producido por los sistemas PLC de Siemens, insertando un *rootkit*[14] y haciéndose con el control de

[10]http://blogs.eset-la.com/laboratorio/2009/01/25/estadisticas-conficker/

[11]http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4250

[12]http://technet.microsoft.com/en-us/security/bulletin/ms08-067

[13]http://www.symantec.com/connect/blogs/stuxnet-using-three-additional-zero-day-vulnerabilities

[14]Programa que permite a un atacante mantener el acceso a la máquina con privilegios elevados, ocultando todo rastro de sí mismo para que el usuario normal no se dé cuenta.

dichas centrales, para espionaje o sabotaje.

Por suerte, los programadores de Stuxnet cometieron el error de no calcular bien las IPs de Irán, provocando que se pudiera detectar en otros países (fue descubierto por unos analistas de virus en Bielorrusia[15]), como se puede ver en la Tabla 1.2.

Tabla 1.2: Países infectados por Stuxnet ordenados por el numero de ordenadores infectados.

País	Ordenadores infectados
Irán	62.867
Indonesia	13.336
India	6.552
Estados Unidos	2.913
Australia	2.436
Reino Unido	1.038
Malasia	1.013
Pakistán	993

Según O'Murchu (autor del artículo de Symantec que hablaba más profundamente sobre el caso), Stuxnet se valía de tres 0days sin publicar, que hacían que su trabajo fuera muy efectivo, pudiendo llegar a colarse en sistemas vitales. Aunque a día de hoy nadie ha reclamado la autoría de Stuxnet, se cree que lo hizo un país que está muy preparado tecnológicamente y que tiene medios como para pagar a programadores muy expertos por el desarrollo un *malware* de este tipo.

1.7 Un problema que no cesa

Puede parecer que la situación de la búsqueda de fallos en el software ha podido disminuir en los últimos años con la mejora de los

[15] http://www.pcworld.com/article/205827/was_stuxnet_built_to_attack_irans_nuclear_program.html

compiladores y de algunas técnicas nuevas de mitigación de errores, pero esto no ha sido así. Las vulnerabilidades encontradas por los investigadores independientes han aumentado en los últimos años.

Aunque las protecciones aplicadas por Microsoft en los últimos años sí que hayan aumentado, los atacantes siempre consiguen nuevos métodos de saltárselas.

Según podemos ver en en los informes que realiza la empresa McAfee, en tanto en 2011[16] y 2013[17] (últimos años que ha creado este informe con estadísticas respecto a otros años) se ha visto un incremento de las vulnerabilidades aparecidas desde 2008, además los fallos descubiertos en Windows 7 superaron a los de XP en 2013. A pesar de ser solo es la gráfica de las vulnerabilidades en Microsoft, la situación no es muy diferente en otros desarrolladores de software.

1.8 Errores en el software

Existen diversos tipos de fallos que pueden suceder a la hora de desarrollar aplicaciones informáticas. Muchos de ellos pueden llevar a la ejecución de código malicioso por parte de un atacante con conocimientos necesarios para llevar a cabo tal actividad.

Aunque existen muchos errores en el software que pueden ocasionar que éste funcione erróneamente o incluso que llegue a dejar de funcionar, son pocos los que son considerados peligros potenciales para la seguridad por dos de las mayores empresas dedicadas a el control de estos errores: CWE [18] y SANS[19].

Gran parte de los errores críticos, tienen su procedencia en lenguajes que no utilizan una *sandbox*[20] como respaldo, que pueda cu-

[16] https://community.mcafee.com/community/business/blog/2011/09/14/ms-patch-tuesday-briefing–september-2011

[17] https://blogs.mcafee.com/business/security-connected/ms-patch-tuesday-briefing-april-2013

[18] http://cwe.mitre.org

[19] http://www.sans.org

[20] Un entorno seguro en el cual los errores pueden controlarse sin poner en

brir los errores que se producen por una mala programación. Un típico ejemplo de un lenguaje con esta protección es Java. En este caso la *sandbox* realiza tres operaciones importantes, la verificación del *bytecode*[21] generado por el proceso de compilación, la carga de clases y la gestión de seguridad.

Gracias a esta protección java se asegura de que sus programas sean correctos. El funcionamiento del verificador de código comprueba varias cosas; las principales son las siguientes:

- Controla que los saltos se hagan entre los límites de memoria permitidos.
- Todas las instrucciones que puedan variar el flujo de control vayan a lugares donde exista código fuente y no a la memoria.
- Ninguna instrucción puede modificar o acceder a una variable local que sea mayor o igual que el número de variables que el método indica que puede manejar.
- El código no puede ir más allá del final de la región dedicada al código.
- Por cada manejador de excepciones, el comienzo y final del código protegido deberá estar en la sección de código. El manejador de excepciones deberá empezar por una instrucción válida y no podrá ser un *opcode* que ha sido modificado por otra instrucción.
- Los argumentos de una función siempre tienen los tipos que la función espera.
- No se puede producir ningún desbordamiento de pila.

Estas normas fueron puestas por Cohen [Coh97] en 1997. Aunque las directivas a seguir estaban claras desde finales de los años noventa, ya existían trabajos que realizaban muchas de las labores

peligro al resto del sistema operativo

[21] Código intermedio entre el código fuente y el ensamblador

que Cohen había marcado como importantes para este lenguaje, como los algoritmos verificadores del *bytecode* de java realizados por Gosling [Gos95] y Yellin [Yel95], en 1995. Actualmente se sigue trabajando en la seguridad de la sandbox de Java, debido a que siguen surgiendo fallos de programación. Cabe destacar el trabajo de Lindholm con Yellin [LYBB13], que desde 1997 vienen desarrollando el estándar del lenguaje de especificación de Java, mejorándolo y creando nuevas versiones periódicamente.

En el otro lado nos encontramos los lenguajes de programación que no tienen ningún tipo de *sandbox*. El caso más común y más famoso de estos lenguajes es C, el cual es conocido por los errores que se dan con programadores poco experimentados. Aunque el lenguaje es mucho más poderoso que el resto de sus competidores (por la posibilidad de controlar hasta un nivel bastante bajo), la dificultad de programar en este lenguaje de programación es muy elevada, razón por la cual han aumentado mucho las vulnerabilidades en este lenguaje de programación. Gran parte de estos fallos son debido al tipado muy permisivo, al trabajo con punteros o la poca posibilidad de controlar los errores en tiempo de ejecución.

Normalmente los fallos en el lenguaje C se engloban dentro de unos errores de programación muy conocidos, y es mínima la posibilidad de que éstos sean completamente nuevos. En las siguientes secciones explicaremos más detalladamente las vulnerabilidades más conocidas dentro del lenguaje de programación C.

1.8.1 Desbordamiento de la pila

Una de las primeras vulnerabilidades que aparecieron relacionadas con los errores software fue el desbordamiento del *buffer*[22]. Uno de los primeros en documentar esta problemática fue Elias Levy [One96], también conocido como Aleph One dentro de la escena *underground*.

Esta vulnerabilidad está en el «Top 3» de las vulnerabilidades más peligrosas dentro del «Top 25» que realiza la página web «Com-

[22] Espacio reservado para guardar datos.

mon Weakness Enumeration» [23]. La razón por la cual se encuentra entre los primeros puestos de esta categoría es porque un atacante con conocimientos puede ser capaz de apoderarse de un sistema completo, consiguiendo los mismos permisos que tiene el programa que contiene el error.

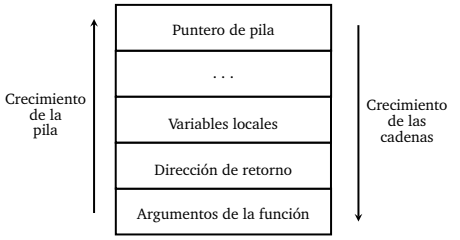

Figura 1.4: Estructura de una llamada a una función en la pila.

La vulnerabilidad de *buffer overflow* consiste en una variable que está mal acotada, y es posible escribir más allá de sus límites. Esta escritura sucede en la memoria reservada para la pila. La estructura de las llamadas a funciones en la pila la podemos ver en la Figura 1.4. Como vemos, las variables locales de la función se encuentran justo antes de la dirección de retorno a la que tiene que volver la función. Un dato a tener en cuenta y que podemos observar en la imagen, es que aunque la pila crezca en un sentido, las cadenas de datos crecen en la dirección contraria. En consecuencia, si un valor en una variable local escribe más allá del limite asignado por el ordenador puede sobrescribir la dirección de retorno.

Al sobrescribir una dirección de retorno, el atacante puede hacer que la ejecución del programa a partir de ese momento vaya hacia donde él quiera. Esto puede llevar a ejecutar cualquier tipo de código dentro del ordenador atacado. Si además este tiene permisos de administrador, el atacante podrá acceder a cualquier dato o programa que se encuentre dentro sin ningún tipo de límite.

El problema del *buffer overflow* es uno de los grandes errores que existen en programación, y cuya explotación es relativamente fácil. Actualmente existen miles de páginas que enseñan a cualquier

[23] http://cwe.mitre.org/top25/index.html

persona interesada a encontrar y explotar este error. Este problema siempre es debido a que el programador no controla correctamente los valores con los que está llevando a cabo las operaciones (normalmente cadenas), y esto lleva a que se produzca el desbordamiento del *buffer*. Aunque un buen hábito de programación podría llevar a que se eviten muchos de los problemas que existen con este error, aun en nuestros días es uno de los errores más comúnmente explotados.

Stack Cookies

Una de las primeras protecciones que surgieron para evitar los desbordamientos de pila fueron las Stack cookies. Esta característica se añadió en los compiladores para que al principio de cada función, si el compilador tenia activada la opción /GS, se incrustará un valor aleatorio en la pila y al final de la función se realizara la comprobación de este. En el caso de que este valor no fuera el mismo que al principio de la función significaría que la pila ha sido sobrescrita de alguna forma y se saldría del programa. A pesar de que no es una solución muy elegante, el control del programa puede conseguir que la aplicación se cierre. Algo que no puede ser muy problemático en muchos programas, excepto si se trata de un programa que ofrece unos servicios críticos.

DEP o Data Execution Protection

Nació como un intento de evitar que se pudiera ejecutar código en páginas de memoria de datos, las cuales no se deberían ejecutar en unas condiciones de programación normales. Este sistema de protección de la memoria fue introducido en los sistemas operativos de Microsoft por primera vez en el SP2 de Windows XP. Aunque existen algunos programas que utilizan estas zonas para ejecutar datos, no es lo común, y deberían marcarlas explícitamente para que tuviesen permisos de ejecución. En el caso de no realizarse, el programa acabaría mostrando un error indicativo de que no se puede ejecutar la memoria.

El DEP es capaz de funcionar en dos modos bien diferenciados. El primer modo esta basado en software y está diseñado para hacerlo funcionar en aquellas máquinas que no lo tienen implementado por hardware. Actualmente, este modo de funcionamiento está prácticamente en desuso, ya que casi todos los procesadores conocidos ya lo implementan por hardware, lo cual facilita la protección de la memoria en máquinas antiguas y que no están preparadas para ello. Este modo tiene que ser desarrollado por los programadores a la hora de compilar el programa, ya que al no estar implementado por hardware, el sistema operativo no controla directamente la ejecución de esta protección. También hay que destacar de este modo, que depende de aplicaciones de terceros ya que Microsoft nunca ha implementado DEP mediante software. Tener una protección de este tipo por parte de terceros, hace que el usuario esté expuesto a que la compañía tarde en sacar parches o actualizaciones si surge un problema en el producto.

El segundo modo de funcionamiento es la verdadera forma del DEP y es la basada en hardware, que implementa bit NX (*no execution*). Este sistema existe desde hace relativamente poco tiempo, y es el sistema operativo, Microsoft Windows en este caso, el que lo configura en el arranque. En la Tabla 1.3 observamos los modos de funcionamiento:

Tabla 1.3: Modos de funcionamiento del DEP

Modo	Valor	Descripción
AlwaysOff	0	Inactivo para todos los procesos
AlwaysOn	1	Activo para todos los procesos
OptIn	2	Protege las aplicaciones de Windows y de terceros que lo soliciten
OptOut	3	Protege todos los procesos excepto aquellos que lo soliciten específicamente

La opción por defecto de Windows es OptIn. Esto es porque

cuando Microsoft se planteó la opción de incluir el DEP en el sistema operativo, se dio cuenta que podría tener muchos problemas de compatibilidad con programas de terceros. Estos programas hacían uso de la memoria de forma incorrecta, ejecutando partes del código en ella; de esta forma, en caso de ser activado los programas fallarían.

En los sistemas a servidores como Windows 2003 o Windows 2008, la opción por defecto es OptOut, para tener un grado mayor de seguridad en todos los posibles procesos que se podrían ejecutar dentro del sistema operativo. Lógicamente, en los sistemas operativos programados para actuar como servidores la seguridad tiene que ser mayor que en aquellos destinados a los usuarios normales.

ASLR o Address Space Layout Randomization

Esta es una característica que traen los sistemas operativos más recientes. Windows empezó a hacer uso de él en la versión denominada *Vista*. ASLR cambia las posiciones en las que se encuentran la pila, las librerías, PEB[24], TEB[[25], el *heap* e incluso la dirección base en la memoria. De esta manera, se aseguraba que aunque un atacante fuese capaz de explotar una vulnerabilidad, no pudiese poner los valores fijos en el *exploit*[26] debido a su variación en cada reinicio de la máquina. Esta protección ha conseguido evitar posibles vulnerabilidades que, sin ella, hubieran sido explotadas.

Uno de los problemas que surgieron al intentar colocar las librerías de un programa en posiciones aleatorias, fue que las *dlls* no se pueden cargar en un trozo de memoria diferente para cada proceso que las cargue. Las librerías del sistema son compartidas entre diferentes programas, y no es posible que cada uno las cargue en posiciones aleatorias. Para solucionar esto, se creó un mapa de bits

[24] *Process Environment Block*, estructura de datos interna de Windows para el control de los procesos.

[25] *Thread Environment Block*, estructura de datos que almacena los datos del hilo que se está ejecutando.

[26] Código que se aprovecha de los errores en el software para conseguir permisos en la máquina que lo ejecuta.

global a todo el sistema, en el cual cada bit representa 64KB de memoria. El nombre de este mapa de bits es `_MiImageBitmap` y por cada dll del sistema, se marca en este mapa de bits, con lo que se conoce la dirección en la que está siendo mapeada dicha dll. Este mapa de bits tiene un tamaño de 0x2800 bits y es capaz de representar las dlls cargadas entre las direcciones de memoria 0x50000000 y 0x78000000. La posibilidad de que un atacante consiga acertar con el valor aleatorio en la que se encuentra la dll es de 1/254, algo que puede ser adivinado fácilmente usando sistemas de fuerza bruta.

Uno de los grandes programas de Microsoft para la seguridad de los binarios en ejecución dentro de los sistemas Windows, EMET[27], implementa esta protección como parte de su sistema para la evitar el uso de exploits por parte de atacantes.

SafeSEH

Otra defensa para combatir el desbordamiento del *buffer* es SafeSEH. Esta característica protege las aplicaciones en tiempo de compilación; es decir, aquellos programas que no hayan sido compilados con esta características no serán protegidos. Este es uno de los grandes problemas con los que se encuentra este mecanismo de seguridad. Cuando un programa se compila con la opción /SAFESEH el compilador genera una tabla con todos los manejadores de excepciones válidos. Con esta tabla el sistema operativo sabe cuándo se ha producido un overflow en memoria, ya que el valor que se haya sobrescrito no está en dicha tabla. Cuando ocurre una excepción y el SafeSEH está activado, el sistema operativo verifica que la dirección del manejador de excepciones que va a lanzarse es un manejador válido, comprobando que existe en dicha tabla. En el caso de que un atacante haya conseguido sobrescribir el SEH y cambiado la dirección del segundo campo del SEH para que apunte a un código ejecutable escogido por el atacante (que normalmente será una secuencia de códigos operacionales `pop/pop/ret`) el sistema operativo comprobará que este valor no se encuentra en la tabla de manejadores de excepciones y finalizará el programa.

[27] https://support.microsoft.com/en-us/kb/2458544.

SafeSEH tiene un gran problema: sólo funciona con aquellos programas que previamente han sido compilados con la opción /SAFESEH activada; lo cual sigue poniendo en peligro a muchos programas, haciéndoles todavía vulnerables a que se sobrescriba el SEH.

Structured Exception Handler Overwrite Protection

Como la solución realizada mediante el SafeSEH no era válida del todo, se necesitaba alguna característica que hiciera que los programas hechos por terceros, aunque hubieran sido compilados con la opción /SAFESEH, estuvieran protegidos. Con esa idea en mente nace SEHOP(*Structured Exception Handler Overwrite Protection*), una característica que protege a la SEH en tiempo de ejecución. La idea básica es la de funcionar como una Stack cookie, pero para la SEH, para lo cual se conserva la configuración original de la cadena del sistema de excepciones, es decir, el campo *Next* de cada estructura apunta a la siguiente. La diferencia está en la parte final de la cadena que, en vez del final anterior con 0xFFFFFFF, ahora contiene un puntero al manejador de excepciones y la siguiente estructura será la que apunte a 0xFFFFFFFF.

1.8.2 Desbordamiento del heap

Al igual que sucedía en el caso del desbordamiento del *buffer*, si no se acotan bien los límites de una variable reservada dinámicamente puede suceder que sobrescriba otros datos alojados en la memoria.

Los datos en el *heap* tienen una cabecera con diversa información que si es sobrescrita puede hacer que el comportamiento de la memoria reservada cambie, o simplemente puede cambiar los datos que están guardados para provocar comportamientos aleatorios o arbitrarios dentro del programa.

La estructura del *heap* es muy distinta a la de la pila que hemos visto en el punto anterior. Cada bloque de memoria que se reserva para una aplicación contiene, además de los datos guardados por el programa, una pequeña cabecera. Ésta es la que contiene información de su situación, y el estado de un bloque en el *heap*. Uno de los

datos más importantes de los que se informa desde esta cabecera es el estado del bloque (libre u ocupado).

Para gestionar dichos bloques, mejorar el rendimiento y evitar la fragmentación de la memoria el *heap manager* divide esta tarea en dos partes:

Front-end allocator La primera linea de actuación para gestionar el heap. Sólo responde a tamaños determinados.

Back-End Allocator Se usa cuando el *Front-End Allocator* no puede cumplir una tarea o cuando no está activado.

Aunque esta es la base de funcionamiento, esto ha ido variando mucho entre los sistemas operativos de Windows. Si nos fijamos en Microsoft encontramos dos maneras bastante diferentes de hacer una gestión del *heap*:

Look Aside List (LAL) Actualmente solo funciona en XP

Low Fragmentation (LF) Vista y superiores

La `Look Aside List` (Figura 1.5) es la responsable de las asignaciones de memoria por debajo de 1024 bits y se encuentra en el offset 0x580 de la estructura _HEAP. El `Front-End Allocator` con `Look Aside List` no es mas que un *array* de 128 listas de una capacidad de 30 bytes cada una. La función de éstas es contener bloques de memoria libres que pueden ser usados por la CPU para asignar memoria a un proceso. Cada lista en este *array*, da servicio a un tamaño determinado, empezando por un tamaño de 16 bytes y sumando 8 bytes mas en cada lista dentro del *array*, hasta llegar a la posición 127 con un tamaño de 1024 bytes. Cada uno de estos bloques contienen una cabecera de 8 bytes la cual contiene unos metadatos que se usan para el control del bloque de memoria y por ello el tamaño mínimo de bloque para la memoria es de 16 bytes, de los cuales 8 son para la cabecera y 8 para los datos que se vayan a guardar en la memoria. En el caso de que no hubiera ningún bloque libre para un tamaño el puntero de *array* de la `Look`

Aside List seria NULL, lo cual haría la solicitud pase al *Back-End Manager* intentando ampliar la memoria con alguna de las técnicas que explicaremos más adelante.

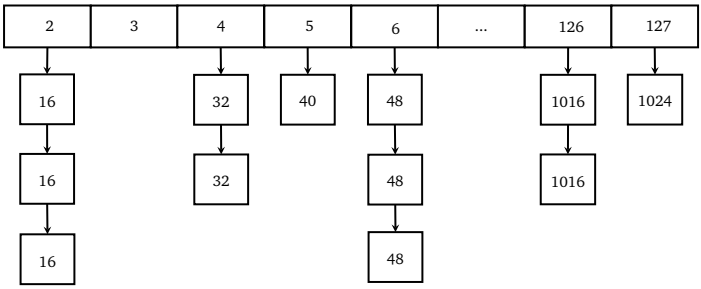

Figura 1.5: Look Aside List.

Cuando la primera parte no puede completar una asignación es cuando entra en juego el Back-End Allocator, el cual contiene un *array* de 128 listas y cada una de ellas corresponden a un determinado tamaño. En este caso a esta estructura se le llama FreeList.

Como podemos observar en la figura 1.6 la forma del Back-End Allocator es muy parecida a la del Front-End Allocator. Al igual que en éste la posición dos de la FreeList también es la que se ocupa de los bloques libres de tamaño 16, el bloque 3 se ocupa de los bloques de tamaño 24 y así sucesivamente hasta llegar al 127 que tendrá un tamaño de 1024. Otro detalle que marca la diferencia es que cada bloque esta doblemente apuntado, con dos punteros llamados Flink y Blink, que apuntan al bloque anterior y posterior respectivamente.

Lo que sí cambia respecto al Front-End Allocator, es que la posición 0 de la FreeList se ocupa de los bloques mayores de 1024 (1016 bytes para asignar datos y 8 bytes para la cabecera) y menores que el limite de asignación de la memoria virtual, ordenados de mayor a menor. Cada bloque de la posición cero de la FreeList contiene también los dos punteros Flink y Blink. El puntero Flink apunta al anterior de menor tamaño y el Blink señala al bloque posterior de mayor. Con esto se consigue que en cada posición de la FreeList exista una cadena doblemente apuntada,

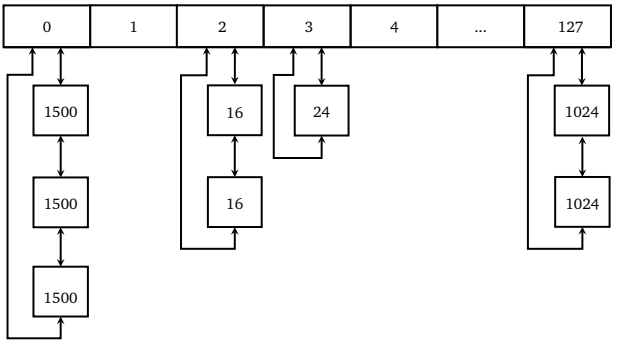

Figura 1.6: Back End Allocator.

y que los bloques estén ordenados por tamaño.

Heap Cookie

Uno de los mecanismos para proteger el *heap* es la `Heap Cookie`, que fue introducido en Windows XP SP2 y que tiene la misma filosofía que la Stack Cookie para la pila. La cookie va en el *heap* y sólo tiene un byte para poder crear un valor aleatorio. Esto le da al atacante una posibilidad de 1/256 de acertar la *cookie*. Aunque no es un valor muy alto, esto dificulta algo más la posible explotación del *heap*. En XP la comprobación del valor del *heap* solo se realiza cuando el bloque de memoria va a ser liberado, lo cual también es un posible problema de seguridad, ya que durante el tiempo que el trozo este asignado, no se realiza ninguna comprobación, haciendo posible que un atacante modifique el valor en el *heap*, sin que la ejecución de la aplicación se dé cuenta hasta que libere dicho trozo de memoria. En Vista y versiones superiores de Windows esto ha sido mejorado y la comprobación del valor de la *cookie* se realiza en varias funciones, haciendo más difícil la explotación.

Safe Unlinking

Safe Unlinking es una protección que también surgió en XP, para que no se pudiese sobrescribir la memoria. Éste se asegura de que un bloque de memoria que va a ser liberado pertenezca a la `FreeList`.

Para ello comprueba que el valor de `Blink` en el bloque apuntado por `Flink`, sea la dirección del trozo de memoria que se quiere liberar. Lógicamente, también realiza la misma operación con el bloque apuntado por el puntero `Blink`. Cuando esta comprobación falla, el proceso que se está ejecutando en XP llama a la función `RtlpHeapReportCorruption()`, que advierte de la corrupción, mientras que en Vista y superiores el `Safe Unlinking` llama a la función `RtlpLogHeapFailure()`, el cual comprueba el valor de `HeapEnableTerminateOnCorruption` y si este valor está activado termina el proceso actual, haciendo que la explotación no sea posible.

Antes de la llegada del `Low Fragmentation Heap`, la única manera de saber que la cabecera de un trozo de memoria en el *heap* estaba sobrescrita era mediante la Heap Cookie, pero esto cambió a partir de Windows Vista. La codificación se realiza sobre los cuatro primeros bytes de la estructura `HEAP ENTRY`. La codificación se realiza mediante una XOR de los tres primeros bytes y se guardan en la variable `SmallTagIndex`. Luego se hace un XOR con los cuatro primeros bytes junto con el valor que se encuentra en `HEAP->Encoding`, el cual se crea cuando se llama a la función `RtlCreateHeap()`, que genera un valor aleatorio.

1.8.3 Format strings

Este tipo de error saco a la luz múltiples vulnerabilidades a principios de 2000. Aunque ahora ya casi no se encuentran de este tipo, no está mal conocerlas ya que siempre se pueden presentar de múltiples formas y aparecer en cualquier momento. Esta vulnerabilidad es muy sencilla de entender ya que se basa en un fallo en la programación cuando se van a mostrar datos en la pantalla con el comando `printf` o similares, y permite una variación del flujo del programa.

La forma correcta de programar un `printf` es esta:

```
char nombre[20];
printf("Introduce tu nombre:\n");
scanf("%s", nombre);
printf("Tu nombre es:\n");
printf("%s",nombre);
```

Y la manera incorrecta y que es donde se encuentra la vulnerabilidad de *format string* es:

```
char nombre[20];
printf("Introduce tu nombre:\n");
scanf("%s", nombre);
printf("Tu nombre es:\n");
printf(nombre);
```

Como podemos ver, el cambio entre los dos códigos sólo es la ultima linea de la función *main*, que pasa de ser un: **printf("%d",array)** a un **printf(array)**. Aunque en ambos casos la salida se muestre correctamente, al intentar realizar un desbordamiento del *buffer* comprobamos que éste no existe. La vulnerabilidad es más sencilla de lo que pensamos, observando la Figura 1.7 se puede entender más fácilmente como el `printf` se comporta en la pila.

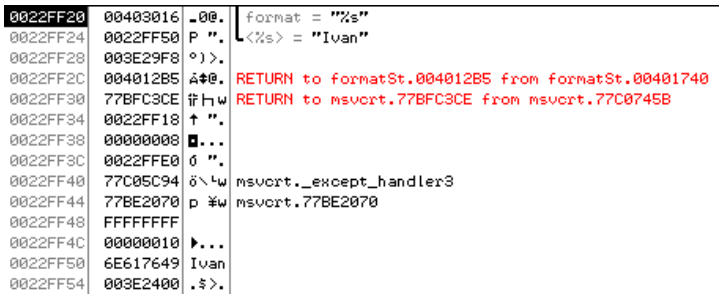

Figura 1.7: Retorno de la pila.

Si ponemos especial atención, podemos observar que la pila muestra los dos argumentos que toma la función `printf` para esta llamada en particular. El primer argumento es la cadena de formato, que en este caso sólo muestra un «%s», aunque podría haber más símbolos de formato o solamente una cadena de texto. El segundo

argumento esta íntimamente relacionado con el primero; como el primer argumento es un «%s», el segundo argumento tiene que ser una cadena que tiene que mostrar por pantalla. En este caso, el valor del segundo argumento apunta a la dirección 0x0022FF50 en donde se encuentra la cadena «Ivan» y será la que muestre cuando se ejecute el `printf`. En la Figura 1.8 podemos observar cómo se comporta el programa vulnerable.

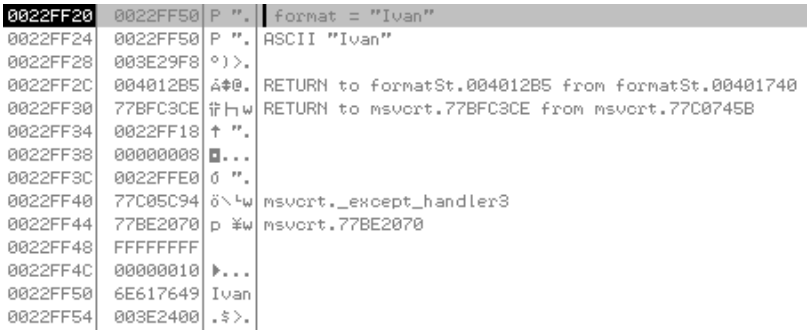

Figura 1.8: Retorno de la pila en el `printf` vulnerable.

Observamos que el primer argumento ya apunta hacia la cadena «Ivan» que se encuentra en la posición 0x0022FF50 de la pila. Y aunque el segundo dato pasado a esta función, que es el que tenía que tener el valor de la cadena de formato, también apunta hacia la posición 0x0022FF50, en este caso la función `printf` no lo va a tener en cuenta ya que en el primer argumento no existe ningún carácter de formato para que `printf` use el segundo valor pasado a la función.

Ahora que ya comprendemos como funciona el `printf`, y cómo maneja los argumentos que se le van pasando, vamos a explicar su vulnerabilidad. Imaginemos que cuando el programa nos pide que introduzcamos un nombre, nosotros en vez de introducir un nombre introducimos una cadena de formato, como por ejemplo "%s%s%s%s". Lo que sucede cuando esto ocurre lo podemos ver en la Figura 1.9, donde se observa lo que sucede en la pila justo antes de ejecutar el `printf` que nos muestra por pantalla cual es nuestro nombre.

Como podemos ver, `printf` coge más argumentos de los que

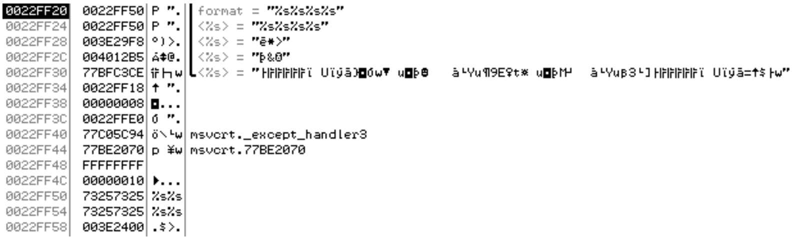

Figura 1.9: Retorno de la pila.

debería, y esto es porque al pasarle una cadena con cuatro especificadores de formato, `printf` piensa entonces que tiene cuatro argumentos. Si comparamos las Figuras 1.8 y 1.9 vemos cómo lo que ha hecho `printf` es coger los siguientes valores de pila para mostrarlos por pantalla, con lo cual nos va a mostrar por pantalla el contenido al que apuntan esas direcciones de memoria, es decir, todos los caracteres hasta que encuentre el carácter de final de cadena 0x00. Si en vez de poner "%s" ponemos "%x" podemos ver las direcciones apuntadas por la pila con el consiguiente peligro para la seguridad del programa ya que esa información es en la que se basan los escritores de malware para hacer *exploits* que ejecuten código aleatorio gracias a esta vulnerabilidad.

1.8.4 Integer overflow e integer underflow

En el mundo de la informática llamamos entero a una variable en la cual se guardan datos numéricos. En *C* o *C++*, las variables de tipo `int` (vienen del ingles *integer*, que se traduce como entero) tienen signo. Las variables `int` suelen tener la misma capacidad que la que tienen los punteros en el sistema que son compilados, es decir, en los sistemas x86 los enteros tendrán un tamaño de 32 bits, mientras que en los sistemas x64 los enteros tendrán un tamaño de 64 bits. Por simplicidad a la hora de seguir con la explicación de los *integer overflows*, pensaremos siempre que el tamaño de los enteros es de 32 bits. La forma de conocer el signo de un valor almacenado en una variable de tipo entero es fijándose en su bit mas significativo, si éste está a uno significa que el valor almacenado en la variable es

negativo, mientras que si el valor del bit mas significativo es cero significa que el valor almacenado es positivo. Mientras que los valores positivos son desde el 0x00000000 hasta el 0x7FFFFFFF, los valores negativos van desde el 0x80000000 hasta el 0xFFFFFFFF. Los números en negativo se calculan mediante la tecnica de complemento a uno, por lo que podemos decir que los números decimales que podemos albergar en las variables de entero con signo van desde el -2.147.483.648 hasta el 2.147.483.647.

En el caso de que un valor sobrepase este valor máximo se hará la operación de módulo con número 0x100000000, con lo cual obtendremos un número que entre en los 32 bits reservados para el valor. Esto mismo sucedería para cualquier otra variable numérica, como por ejemplo `short` o `double`.

Ahora que conocemos cómo son representados los enteros nos será más fácil entender los *integer overflow/underflow*. Vamos a ver un ejemplo muy simple para comprender cómo se producen:

```
#include <stdio.h>

int main(void)
{
        int i;
        i=2147483647;
        printf("%d\n",i);
        i=i+1;
        printf("%d\n",i);
}
```

En este ejemplo se le introduce un valor a la variable i el valor máximo que puede almacenar un entero con signo; después de esto lo mostramos por pantalla para que podamos ver el valor; acto seguido sumamos uno al valor que esta almacenado en i y lo mostramos por pantalla. Se podría llegar a pensar que si i es igual a 2147483647 entonces:

$$\text{i} = 2147483647 + 1 = 2147483648$$

Pero si ejecutamos el programa veremos que la realidad es bien diferente. El resultado del primer printf nos devuelve correctamente

el número 2147483647, pero el segundo nos devuelve el número -2147483648, ya que si hacemos la misma suma en hexadecimal veremos que:

$$\text{i} = \text{0x7FFFFFFF} + \text{0x00000001} = \text{0x80000000}$$

Y como hemos visto antes, el número 0x80000000 es tomado por el sistema como un numero negativo, deñ que si hacemos su complemento a uno, vemos que:

$$\text{0x80000000} + \text{0x00000001} = \text{0x80000001} = 2147483648$$

Y como al tener el bit mas significativo puesto a uno sabemos que es negativo, con lo cual es el -2147483648.

Aunque esta función de ejemplo parece poco peligrosa veamos qué sucede si lo extrapolamos a una función que hace un *malloc*[28]:

```
#include <stdio.h>
#include <string.h>
#define TAMANO(x, y) ((x)>=(y) ? (1):(0))

int main(void)
{
  int letras;
  char *mem, nombre[20];
  printf("Longitud de tu nombre:\n");
  scanf("%d",&letras);
  printf("Escribe tu nombre:\n");
  scanf("%s",nombre);
  if(TAMANO(letras,strlen(nombre)))
  {
    printf("Reservando espacio para: %d\n",letras);
    mem = (char *)malloc(letras*sizeof(char));
    memcpy(mem,nombre,strlen(nombre));
    printf("Se ha guardado en memoria %s",mem);
  }
  else
  {
    printf("Error: nombre demasiado largo\n");
    return 0;
  }
}
```

[28]Reserva dinámica de memoria

Si ejecutamos este programa vemos que es un simple programa que te pide el tamaño de tu nombre; más tarde reserva la memoria y lo guarda. En el caso de que intentemos hacer un *heap* overflow poniendo un tamaño menor que el nombre que vamos a introducir el programador ha hecho una pequeña macro que nos impide hacer esto, pero que nos va a permitir hacer un *interger overflow*.

Aunque en un principio este simple programa puede parecer que está bien programado evitando el *heap overflow*, tiene un grave problema que se encuentra en la macro y en la comparación que efectúa. La macro hace la comparación sin tener en cuenta el signo, lo cual hace que el *integer overflow* sea posible.

Figura 1.10: Código ensamblador de la vulnerabilidad.

En el código en ensamblador de la Figura 1.10 vemos que después del `scanf` se hace la comparación en la línea 0x00401303, entre lo que actualmente se encuentra en EAX (que después de la llamada al `strlen` es el tamaño del nombre que se ha introducido) y lo que hay en EBP-C, que es la variable local en la que guarda el tamaño que nosotros hemos introducido. Justo después de eso, en la linea 0x00401306 es donde realiza el salto con la orden JB. La especificación de Intel para el código ensamblador indica que se corresponde a un código operacional que salta cuando el valor es inferior, pero que no tiene en cuenta el signo, que es lo que permite que, si introducimos un valor superior a 0x7FFFFFFF, éste lo tome como un valor positivo, con lo cual no saltará. Si cambiamos este código operacional por su equivalente que toma en cuenta el signo JL, comprobaremos que hará correctamente dicha comprobación y no como la está haciendo hasta ahora.

Cuando el programa pide el tamaño de nuestro nombre le podemos introducir un valor mayor de 2.147.483.648 y menor de 4.294.967.295, con el cual cuando se ejecute la llamada a `malloc` esta devolverá un puntero nulo (ya que no va a reservar ningún sitio

en memoria de tamaño negativo), y al seguir el programa y ejecutar la llamada a la instrucción `memcpy`, se producirá un error por tratar de escribir en una región no válida de memoria que este caso seria SS:00000000.

Aunque los *integer overflows* no son una vulnerabilidad en si misma como los *buffer overflow*, sí que puede hacer que se varíe el flujo del programa y esto puede ocasionar algún tipo de vulnerabilidad de la cual un atacante se podría aprovechar para tomar el control de la aplicación. Aunque en el ejemplo hemos introducido los valores directamente, en la mayoría de los programas esto no sucede tan fácil. Las operaciones que hacen posible los *integer overflow* son las de suma y multiplicación.

La protección contra este error no es otra que tener unas buenas prácticas a la hora de programar, evitando que el programador muestre por pantalla los valores de forma que propicie un error de este tipo. Aunque este error puede ser fácilmente detectable mediante análisis estáticos simples estos errores todavía se pueden encontrar a menudo en programas con bastante fama.

1.8.5 **Condición de carrera**

Las condiciones de carrera son un problema que surge con la computación multitarea. El error aparece cuando se intenta acceder a un recurso y son dos procesos los que acceden al mismo modificándolo o cerrándolo, con lo cual en el momento en el que un proceso intenta acceder a el recurso este ha cambiado, provocando la vulnerabilidad. Como siempre la mejor forma de analizar esta vulnerabilidad es a través de un ejemplo:

```
if(!access(nf, W_OK))
{
  printf("Letra que quieres buscar: ");
  buscar=getche();
  fichero = fopen(nf,"r");
  do
  {
    letra = fgetc(fichero);
    if(letra==buscar)
      contador++;
  }while(letra!=EOF);
  fclose(fichero);
  printf("\nHay %d letras en el fichero\n",contador);
}
else
  printf("Error el fichero no existe\n");
```

En este caso la solución de este problema no es muy complicada porque simplemente comprobando si la función `fopen` devuelve `NULL` controlaremos el posible error. La orden `access` actualmente no se utiliza, debido a que directamente intenta abrir el fichero y, de esta forma, queda bloqueado para cualquier otra operación que otro programa, usuario o el sistema operativo intenten hacer con él.

El problema de las condiciones de carrera es mucho más problemático de lo que se muestra con este ejemplo, ya que donde verdaderamente surge es en la programación multihilo, en la cual una aplicación puede acceder con varios hilos al mismo recurso siendo esto mucho más difícil de controlar por el programador. Hay que tener en mente que aunque a la hora de hacer código nos parezca que una operación es atómica, en el momento de la compilación y pasarlo a ensamblador esta operación puede tomar varias operaciones en este código, haciendo que la operación ya no sea atómica, con lo cual ya es vulnerable a sufrir una situación de condición de carrera.

Las causas principales que provocan situaciones de condición de carrera son el tratamiento de ficheros, señales y las operaciones de red.

También hay que pensar que encontrar esta vulnerabilidad es bastante problemático, ya que al no ocurrir siempre es difícil de encontrar; mucho mas hacer un código que la pueda explotar para demostrar dicha vulnerabilidad, ya que hay que reunir unas de-

terminadas condiciones en cada ejecución del programa lo que en muchas ocasiones es complicado. De la misma forma que es difícil encontrarla y explotarla también es complejo realizar un código para evitar que esto suceda, por esto todavía no existe ninguna protección oficial contra este error que proteja al usuario en tiempo de ejecución.

1.9 Hipótesis y objetivos

En la sección anterior hemos visto cuáles son los principales puntos de esta tesis. Basándonos en ello, la hipótesis de partida es:

> *«Existe un método de representación de la memoria que es capaz de detectar errores conocidos, tanto en el heap como en la pila.»*

El objetivo principal de esta tesis, como se puede comprobar después de todo el estudio realizado hasta el momento es el siguiente:

> *«Diseñar e implementar un algoritmo verificado capaz de detectar los errores en los binarios antes de su ejecución.»*

Los objetivos específicos son los que se relacionan a continuación:

Objetivo 1 *Diseñar e implementar un algoritmo de capaz de determinar cual ha sido el camino que ha tomado una ejecución hasta encontrar un error, que actúe como base para el siguiente objetivo.*

Objetivo 2 *Diseñar e implementar un método verificado, que no genere errores en su ejecución capaz de detectar los principales errores producido en la programación de software.*

Objetivo 3 *Diseñar e implementar un método capaz de detectar y evitar los errores en las fórmulas de nuestro algoritmo que han sido introducidas por el usuario.*

Objetivo 4 *Diseñar e implementar un método capaz de parsear el código ensamblador y adaptarlo al lenguaje del algoritmo.*

Objetivo 5 *Demostrar la eficacia de los métodos anteriores con programas reales.*

El primer objetivo se basa en la creación de lenguaje que ayude a predecir los errores que pueden llegar a ejecutarse en la memoria en tiempo de ejecución. Este algoritmo tiene que ser rápido y preciso, para que un usuario pueda ejecutarlo sin problemas. Éste será capaz de detectar cuando se va a producir un error en el programa por un error en la memoria y devolver su localización y ruta hasta él.

El segundo objetivo, y teniendo en cuenta el algoritmo generado anteriormente, consiste en crear un programa verificado que no contenga código en sí mismo. Así el programa será capaz de detectar errores sin generar nuevos problemas, y conociendo que el programa siempre funcionará correctamente.

El tercer objetivo se basa en detectar los errores que se pueden producir cuando el usuario utilice las fórmula para encontrar errores en nuestro algoritmo. Esta aplicación debe hacer un análisis léxico y sintáctico del lenguaje con el que se implementará el algoritmo.

En el cuarto objetivo se adaptará el código ensamblador a nuestra representación intermedia de este código para que este sea capaz de ser interpretado correctamente por nuestro algoritmo.

Además de estos objetivos principales, se tendrán en cuenta otros a la hora de llevar a cabo la tesis. Estos objetivos operacionales son:

Escalable *Diseñar un lenguaje que sea adaptarse y pueda ser mejorado sin mucha dificultad.*

Transportable *Diseñar un lenguaje que sea posible ejecutar entre los diferentes sistemas operativos sin realizar excesivos cambios.*

Independencia *Diseñar un lenguaje que no dependa de la máquina en la que se ejecute, para que el programador no se vea sometido a nuevos problemas.*

Seguridad *Minimizar el número de vulnerabilidades que pueden salir del entorno de producción.*

Para comprobar los resultados del lenguaje intermedio para la representación de la memoria, se evaluará mediante medios matemáticos como lo hacen otros lenguajes matemáticos como, por ejemplo, *dataflow analysis* o *abstract interpretation*. En este caso se utilizará la «calculo de las construcciones inductivas», uno de los métodos más utilizados por los investigadores en este campo.

1.10 Metodología de la investigación

Para la realización de esta tesis doctoral se ha elegido utilizar el método mc-14 [Edm94]. Aunque se barajó la posibilidad de utilizar otros métodos, como por ejemplo el conocido método hipotético-deductivo [But69], se ha creído que el método *MC-14* es más aconsejable para los objetivos que perseguimos y está mejor adaptado a la forma de trabajar en la especialidad en la que se va a basar esta tesis, la informática. El método hipotético-deductivo es muy similar si miramos la forma simplificada del *MC-14*, pero no se centra tanto en la parte de las conclusiones y el proceso de investigación, siendo su última fase la de experimentación.

Por esta falta de las secciones de conclusiones, y por la modernidad de *MC-14*, es por lo que hemos escogido este método para llevar a buen fin esta tesis. Aunque como bien sabemos las fases que componen esta metodología son catorce, se pueden resumir en cuatro puntos:

1. Observación e investigación
2. Hipótesis
3. Creación y realización de experimentos

4. Resultados y Conclusiones

Aunque los 14 estados que tiene este método científico que vamos a seguir son mucho más precisos, este resumen de fases es en el que nos basaremos para explicar como han sido las fases de nuestra investigación.

Este método esta basado en la formulación de una hipótesis después de una observación. Ésta tiene que ser lo más concreta posible, evitando todo tipo de generalización que pueda llevar a una tesis muy difusa y sin ningún objetivo realmente claro.

Observación e investigación

El primer punto con el que se inició la investigación fue la recopilación de un estado del arte completo y de la adquisición del conocimiento necesario para llevar a cabo los sucesivos experimentos. En nuestro caso se estudiaron las diferentes técnicas de *model checking* y de revisión estática de código hasta que se dio con una carencia dentro de este campo.

Hipótesis

En ese momento se definió expresamente la suposición de partida, y cuáles eran los problemas a los que se quería dar solución con esta investigación. Este es el punto más importante dentro de la tesis, ya que este apartado constituye la base sobre la que se va a construir el estudio posterior.

Realización de experimentos

En esta parte se desarrolló el algoritmo, se realizó la validación mediante la lógica matemática, y se implementó el programa. Ésta es la parte central de la investigación y sin la cual no se pueden llevar a cabo los experimentos.

Resultados y Conclusiones

Una vez que el programa estaba finalizado se llevaron a cabo los experimentos para comprobar que se ha validado la hipótesis de partida. Una vez finalizados, se realizó una comprobación de los resultados y las posteriores conclusiones de éstos.

1.11 Arquitectura del sistema propuesto

El sistema propuesto esta tesis, puede definirse en cuatro partes diferenciadas.

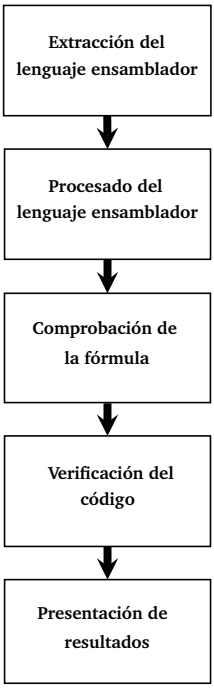

Figura 1.11: Arquitectura del sistema.

Extracción del lenguaje ensamblador En esta parte se extraerá el código ensamblador del binario que se vaya a estudiar. Para ello se utilizará IDA Pro[29]. Este programa es capaz de recono-

[29]https://www.hex-rays.com/products/ida/

cer muchas de las estructuras de datos dentro del binario. IDA Pro es capaz de identificar la funciones de la aplicación a examinar, con lo cual es posible analizar partes del un programa en vez de hacerlo con el programa completo.

Procesado del lenguaje ensamblador En está sección se procesará el código que hemos extraído en el apartado anterior para que cumpla los requisitos necesarios para se pueda hacer la verificación de la formula.

Comprobación de la fórmula En esta parte se comprueba que la fórmula introducida por el usuario sea correcta siguiendo las especificaciones del lenguaje, mediante un análisis sintáctico y léxico.

Verificación de código La comprobación de las fórmulas en el lenguaje ensamblador del binario se hará en esta sección. Para ello el programa recorrerá estáticamente cada posible ruta que puede tomar el ejecutable desde su punto de entrada.

Presentación de resultados Aquí se mostrarán los fallos que han sido encontrados aplicando nuestra comprobación de errores. Esta presentación deberá ser clara mostrando los resultados para que un usuario pueda entender.

1.12 Estructura del documento

El contenido de esta tesis doctoral se estructura de la siguiente forma:

Capítulo 2

Repasaremos el estado del arte dentro del marco de esta investigación, haciendo principal hincapié en las técnicas para encontrar errores que se han investigado hasta el momento. Veremos cuáles son los algoritmos existentes en la actualidad para la búsqueda de errores en el lenguaje ensamblador o para la comprobación de código fuente. Finalmente veremos cuáles son los programas existentes

para resolver problemas similares de los que trata este estudio.

Capítulo 3

En el capítulo 3 se mostrará el algoritmo matemático creado durante el estudio realizado para esta tesis. También se explicará la nueva operación matemática necesaria para que nuestro algoritmo funcione correctamente, y la explicación de cuál fue el problema que originó que se tuviera que crear. La definición de la estructura Kripke, que utilizamos para nuestro modelo, también será explicada aquí, así como la sintaxis y el léxico de nuestro nuevo algoritmo para buscar código.

Capítulo 4

En este capítulo veremos cómo se ha implementado y verificado la parte de código con la que se realizan las búsquedas en el software. Se explicará cuales han sido las equivalencias que hemos utilizado entre el lenguaje matemático y el usado para la realización del programa. También veremos como funciona el programa con el cual se han llevado a cabo las tareas de verificación, así como una explicación detallada de como se realizan. Se explicará como ha sido el proceso de creación y como se ha definido la representación intermedia del lenguaje ensamblador Intel x86, para acercarla a nuestra algoritmo. De esta manera se realiza más fácil el proceso de la comprobación de código ensamblador.

Capítulo 5

El capitulo cinco es el dedicado a la presentación y explicación de los resultados obtenidos mediante esta tesis. Además, se mostrará cual es la complejidad que tiene el algoritmo que hemos creado y como repercute esto con los problemas conocidos que tiene la técnica que utilizamos para el estudio y descubrimiento de vulnerabilidades.

Capítulo 6

En este capitulo se extraen las conclusiones obtenidas a partir de los resultados obtenidos. Además trataremos los problemas y li-

mitaciones que se han encontrado mientras se llevaban a cabo tanto los experimentos, como creación del algoritmo y de la implementación del mismo. También expondremos unas recomendaciones y el posible trabajo futuro con el que se puede seguir la investigación de esta tesis.

1.13 Conclusiones

Como hemos observado en este capítulo, las vulnerabilidades en el software suponen un serio riesgo que debe de ser tratado con la gravedad que se merece. Un error en un programa informático puede ser una grave amenaza para todo un sistema, comprometiendo totalmente su seguridad. Estos fallos no solo se producen por los malos hábitos de los programadores, sino también por una mala gestión de las optimizaciones por parte de los compiladores.

La gravedad de estos errores puede ser desde un simple cierre del programa que contiene el error, hasta que un atacante pueda llegar a conseguir el control de toda la máquina que ejecuta la aplicación. Este tipo de fallos en el entornos críticos puede llegar a comprometer incluso la seguridad física de las personas. Este tipo de fallos son los que se están llevando a cabo actualmente por diferentes países para realizar sus ataques en la ciberguerra global que se está llevando a cabo.

En la actualidad se están realizando muchos estudios para conseguir algoritmos que detecten errores en el software, ya sea examinando el código compilado dentro de un ejecutable, como en el propio código fuente. El objetivo de todos estos estudios es intentar reducir al mínimo el problema de los errores en el software y de esta manera tener programas más seguros, que no supongan ningún tipo de riesgo.

En el siguiente capitulo veremos cuáles han sido los avances hasta el día de hoy en la creación de algoritmos de búsqueda de errores, y las diferentes variantes que existen dentro de cada uno de los campos, centrándonos más en aquellos que están directamente relacionados con esta tesis.

«Sabemos muy poco, y sin embargo es sorprendente que sepamos tanto, y es todavía mas sorprendente que tan poco conocimiento nos de tanto poder.»

Bertrand Russell (1872–1970)

Capítulo

2

Estudio de la literatura

El rápido crecimiento de los sistemas informáticos ha llevado consigo que el incremento de la complejidad de los programas también haya aumentado. Esto ha ocasionado que se haya vuelto imprescindible la creación de algoritmos que resuelvan de manera eficaz la búsqueda de errores dentro de los programas. Por esta causa, diversos equipos de científicos en todo el mundo se han centrado en el asunto de la seguridad informática.

Los métodos formales proporcionaron una forma efectiva de verificación que reduce el tiempo para encontrar problemas en los programas, sobre todo en las fases de diseño. El reconocimiento de estás técnicas es tal que se encuentran en manual de buenas prácticas de organizaciones tan importantes como la Agencia Espacial Europea o la NASA, en cuyos casos es de vital importancia que el software no contenga ningún error.

2.1 Introducción

Muchos programadores tienen la creencia de que si programan el código correctamente, el binario que genere el compilador no contendrá errores, sin sin contemplar la posibilidad de que el compilador puede introducir fallos. Los errores en este tipo de software acaban influyendo en la forma en la que se genera el binario, haciéndolo menos eficiente e incluso, en algunos casos, incorrecto.

Los compiladores han avanzado mucho desde sus inicios y cada día se están convirtiendo en programas que ofrecen más posibilidades a los programadores. El avance en los procesadores también ha traído consigo que aumente la complejidad de los programas, haciendo que varios procesadores puedan ejecutar el mismo programa algo que también tiene que ser controlado para que no se produzcan errores. Los compiladores más modernos incluso permiten que se pueden programar una misma aplicación en varios lenguajes de programación, como sucede con Visual Studio[1].

Son estas características las que hacen que la complejidad de los compiladores se incremente exponencialmente, consiguiendo que estos sean más proclives a cometer errores en el proceso de la generación del binario. Los errores de este tipo son los más complicados de detectar en el proceso de desarrollo de la aplicación, motivo por el cual el programador puede no llegar nunca a darse cuenta que se está produciendo ese problema, ya que su código es correcto desde el punto de vista de un lenguaje de alto nivel. Por esta razón, se impone realizar un profundo análisis, no solo del código fuente de la aplicación, sino también del binario que es generado por cualquier tipo de compilador.

En el resto de este capítulo repasaremos los principales métodos para la detección de errores, comenzando en cómo es la situación actual en la búsqueda de errores y los tipos de análisis que se realizan para encontrar errores en el software.

Más tarde nos centraremos en los análisis estáticos, reflexionan-

[1]Como se puede observar en la web: https://msdn.microsoft.com/en-us/library/aa270930(v=vs.60).aspx

do sobre los algoritmos que más se utilizan dentro de este ámbito. Aquellos que han sido utilizados en los procesos de verificación del software y de la seguridad en múltiples aspectos, los cuales iremos viendo a lo largo de este capítulo.

Las lógicas temporales y el *model checking* también tienen un apartado muy importante en la verificación del sistemas. Estos dos elementos están íntimamente relacionados ya que el *model checking* utiliza la lógica temporal para hacer sus razonamientos. Estas técnicas han sido utilizadas para múltiples propósitos dentro de la verificación de código, e incluso en plataformas hardware para la creación sistemas sin errores en su diseño. Todas las técnicas y problemas que han surgido con su aplicación serán mostrados a continuación.

2.2 Sistemas actuales

En la actualidad los investigadores en el área de la seguridad informática realizan todo tipo de estudios para intentar resolver los problemas relacionados con la programación de aplicaciones. De todos los estudios que se realizan en este campo, los que alcanzan mayor repercusión en este ámbito son los que intentan solucionarlo basándose en el estudio del código. Hasta el día de hoy, se han implementado soluciones para los tipos más comunes de situaciones en los cuales se pueden detectar los errores: en tiempo de ejecución o de compilación.

En la Figura 2.1, vemos las diferentes técnicas que se emplean para la mitigación de errores en el compilador. En ella, podemos observar los dos grupos que hemos mencionado claramente diferenciados. Puede sorprender, que como estamos hablando de errores que se pueden abordar desde el compilador, estos mitigan errores en tiempo de ejecución. Esto es debido, a que el compilador generará el ejecutable con protecciones para los errores conocidos, y de esta forma evitará que se ocasionen errores cuando el programa esta funcionando como hemos visto en la introducción.

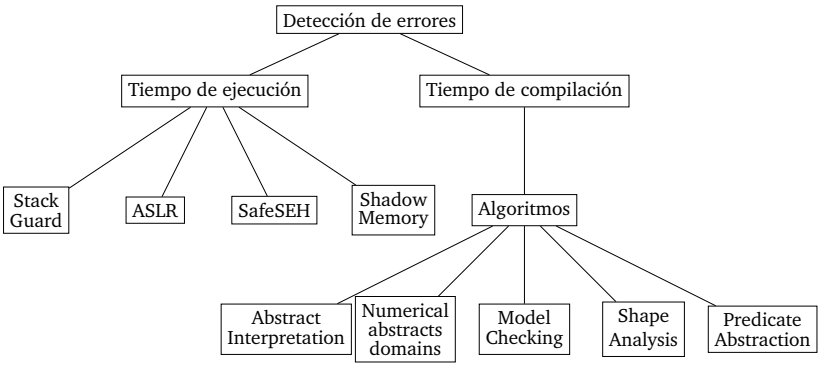

Figura 2.1: Detección de errores desde el compilador

Los investigadores se inclinan por dos posturas muy diferentes. Por un lado, tenemos los investigadores que abordan el problema desde un punto de vista más práctico de la informática, haciendo programas para aplicaciones muy concretas, como son los realizados en las protecciones en tiempo de ejecución. Esta rama suele estar desarrollada por profesionales de la seguridad informática realizando protecciones para cubrir un propósito muy especifico. Esta parte ya fue tratada en la introducción de esta tesis, mostrando las aplicaciones que han desarrollado empresas, como Microsoft, para luchar contra las vulnerabilidades, entre las que podemos encontrar el SafeSEH o ASLR, entre otros.

Los enfoques más teóricos suelen ser menos aplicables en entornos reales, ya que aunque demuestren resultados prácticos, en la mayoría de los casos suelen estar basados en errores demasiado específicos. Otro problema que puede llegar a darse en este terreno es que las publicaciones científicas no siempre vienen acompañadas del código fuente con el que se han realizado las pruebas, haciendo que la labor de comprobar los resultados obtenidos sea aun más complicada.

2.2.1 Historia de la búsqueda de errores

Desde incluso antes de la creación de los primeros ordenadores (tal y como los conocemos ahora mismo), ya existía una gran preocupa-

ción por los errores que se podían producir en la programación de aplicaciones. Los primeros en poner el foco en este problema fueron Goldstine y Von Neumann [GVNM+48] por un lado, y Church [Chu36], Gödel [Göd31] y Alan Turing [Tur49] por otro. Ellos fueron los pioneros en este campo, y sentaron las bases para todos los trabajos posteriores, ya que incluso a día de hoy se siguen consultando sus artículos para conocer los primeros pasos dentro de la corrección de errores.

En 1964, C.A.R. Hoare [Hoa69], empezó a trabajar en la *lógica de Hoare*, un sistema que trata de mitigar los errores, ya sea con una buena especificación antes de hacer el programa, o con un buen estudio de este una vez realizado. Está lógica la veremos con más detalle más adelante.

Poco después que Hoare, Manna y Pnueli [MP92] crearon la *Lógica de corrección total*. Sus estudios se basaron en el trabajo de Hoare, pero mientras éste usaba la corrección parcial (la precondición es cierta y el programa se ejecuta normalmente, entonces la postcondición será cierta), Manna y Pnueli intentaban conseguir la corrección total (la precondición tiene que ser cierta, el programa ejecutarse correctamente y la postcondición ser cierta también).

Mas adelante, Floyd [Flo67] y Naur [Nau66], siguieron con el trabajo de sus predecesores y crearon una serie de especificaciones para intentar probar la corrección del software basadas en las salidas de los programas. Su trabajo derivó en el método *Floyd-Naur*, también basado en la lógica de Hoare.

Ya en el año 1975, Dijkstra [Dij75] publicó un trabajo introduciendo la semántica de transformación de predicados. En su trabajo estableció el concepto de la "precondición más débil", que podemos ver en la ecuación 2.1, ()en donde S, es una secuencia de comandos a cumplir y R una postcondición que debe cumplirse, y lo que se intenta conseguir es el caso más débil de ejecución del programa con la salida deseada).

$$wp(S,R) \tag{2.1}$$

En año 1977, Patric Cousot [CC77] creó el método conocido co-

mo *abstract interpretation* para poder analizar los valores que adquieren los programas en cada momento de su ejecución y, de esta forma, determinar si se producen errores. Además de para analizar el comportamiento de los programas durante toda su ejecución, *abstract interpretation* también se utiliza para decidir optimizaciones y transformaciones dentro del compilador. Mediante *abstract interpretation*, se puede también analizar una aplicación en base a un dominio de valores abstractos, obteniendo una versión abstracta de la aplicación. De esta forma, no es necesario ejecutar el programa en ningún momento y conseguimos un visión más amplia del funcionamiento del mismo.

Un punto de inflexión en el problema de las vulnerabilidades, la marcó el primer gusano informático. En 1988, Robert T. Morris [Mar90] aprovechó una vulnerabilidad del servidor de correo Sendmail, para infectar a un 10 % de los ordenadores que se encontraban conectados ese año a Internet (unos 6.000). El gusano causó unos daños que fueron valorados entre 200 y 53.000 dolares de la época.

Este hecho alertó a todos los administradores de sistemas del mundo, quienes empezaron a interesarse por la seguridad (un campo inexistente hasta entonces). Desde ese momento, el interés por el malware y los ataques sobre el software se convirtió en un punto más a tener en cuenta.

La década de los noventa trajo consigo un aumento del interés de los investigadores sobre el *model checking* (centrándose en la búsqueda de errores). Manna et al. [MP92] fue uno de los impulsores de utilizar este método para buscar errores, seguido por técnicas como *simbolic model checking* [McM93] o *bounded model checking* [BCCZ99]. Ambos se detallarán más adelante.

Ya en 2003, Ding [DZ03] realizó un estudio para intentar conocer los accesos a la memoria analizando todas los posibles puntos de acceso a un programa para intentar mejorar su eficiencia. En su estudio, representa el acceso a los datos en forma de árbol binario, en el cual, cada nodo contiene los valores necesarios para hacer efectivo el cálculo de los accesos a memoria.

Al año siguiente, Balakrishnan [BR04] desarrolló otro sistema

para analizar los datos de los programas en la memoria, llamado *value-set analysis*. En este trabajo, creó un método que se basó en la análisis de estático de programas realizado por Cousot [CC77], capaz de analizar la memoria de los ejecutables solo mirando el código ensamblador que es generado por el compilador. A su vez, Balakrishnan, desarrolló CodeSurfer[2], una herramienta con la que consiguió demostrar el potencial de su método para la búsqueda de errores en binarios para las plataformas x86. Ésta herramienta realiza un estudio de los punteros, el cual determina las dependencias del programa a examinar y, mediante una interfaz gráfica logra que la interacción con el usuario sea excelente. Además, una muy trabajada API permitía que otros investigadores pudieran desarrollar *plugins* para mejorar esta herramienta.

Ya en el año 2012, Younan [YJPVdE12] diseñó un modelo para la gestión de memoria que aumentaba la seguridad separando la cabecera del *heap* de sus datos, situándolos en regiones de memoria diferentes. De esta manera aunque los datos en el *heap* fuesen sobrescritos por un atacante este ataque no podría sobrescribir la cabecera. Mediante esta protección se evitaría muchas de las vulnerabilidades producidas por el desbordamiento del *heap*. Para mayor protección , Younon inserta *guards* al principio y al final de la región, y separa los datos del propio trozo reservado de la información que el usuario quiere guardar, asegurándose de que un desbordamiento de esta cabecera no sobrescriba la información del usuario. Además, en el momento que ya no haya sitio para insertar nuevas cabeceras, se reservará otra región de memoria insertando también los *guards* para protegerla. Un inconveniente que surge de este método es que cada región tiene su propia protección independiente de las otras regiones, lo cual hace que el tiempo de computación para poner permisos es excesivo.

Recientemente, Song et al.[ST13b] crearon Pommade[3], una herramienta para buscar malware utilizando *model checking* basándose en las técnicas *pushdown model checking* y en CTPL de Kinder [KKSV05]. Una herramienta capaz de detectar 200 muestras de

[2]http://www.grammatech.com/research/technologies/codesurfer
[3]http://www.liafa.univ-paris-diderot.fr/~song/pommade/

malware creadas mediante NGVCK y VCL32. Algo que otros antivirus como Avira o Avast no fueron capaces de detectar.

2.2.2 Técnicas utilizadas

Un atacante experimentado suele ser capaz de encontrar la forma de explotar un error, por muchas protecciones que contenga el ejecutable. Tanto las *guards* [CPM+98], SafeSEH [4], como el *ASLR* [5] que hemos visto en el apartado anterior, no son técnicas perfectas. Lo que conlleva que un atacante con suficientes conocimientos pueda saltárselas al entender como funcionan y cuales son sus debilidades.

Un caso distinto es el de *shadow memory*, el cual no implementa una protección del mismo tipo que lo hacen las protecciones que hemos comentado antes. Esta técnica realiza un mapeo de la memoria, la que en principio era muy costosa en términos de rendimiento, ya que prácticamente doblaba la memoria que necesitaba un programa para funcionar, pero Nethercote y Seward [NS07] lograron superar este problema para implantarlo en Valgrind[6]. En su estudio desarrollan una técnica para mejorar el método por el cual los bytes son copiados en la memoria, creando la herramienta Memcheck para demostrar su funcionamiento. Con ello consigue evitar muchos errores de en la memoria de los procesos protegidos con esta herramienta. Además de esta herramienta, también desarrollaron ThreadCheck con el mismo sistema, aunque esta vez para detectar condiciones de carrera entre los diferentes hilos de ejecución.

Basado también en *shadow memory*, Serebryany creó Address sanitizer [SBPV12]. Esta herramienta fue incluida dentro del compilador LLVM desde la versión 3.1 del mismo, consiguiendo la protección del binario en tiempo de compilación. Gracias a este nuevo enfoque ha conseguido superar a Valgrind en rendimiento de la herramienta, haciendo que Address sanitizer funcione mucho más rápido que MemCheck.

[4] https://msdn.microsoft.com/es-es/library/9a89h429.aspx

[5] http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-in-windows-vista.aspx

[6] http://valgrind.org/

Otro método muy utilizado para proteger el software de las vulnerabilidades es desarrollar unas nuevas librerías de gestión de memoria. Ejemplos de estas librerías son: Libmib[7], Safestr[8] o Libsafe [TS02]. Estas librerías proporcionan seguridad adicional a la memoria en tiempo de ejecución introduciendo protecciones, y son muchos los programas que utilizan esta técnica con buenos resultados. Estas nuevas librerías hacen que sea más complicada su adopción por parte de los programadores, que se ven obligados a cambiar su forma de hacer reserva de memoria y, en muchos casos, complica la forma de programar haciendo que los ciclos de vida del software se vuelvan más lentos. Y aunque esta técnica se usa en muchos programas, tampoco permite saber en tiempo de compilación si estamos creando un programa peligroso, debido a que este tipo de librerías no están preparadas para ello.

Las técnicas basadas en tiempos de compilación hacen que los fallos no lleguen a producirse. Con esto, los programas son más seguros, ya que el programa tendrá un código correcto a la vez que podemos implementar las protecciones de tiempo de ejecución. Con todo esto, se conseguirá que, en el poco probable caso de que se encuentre un error en el programa, este sea difícil de explotar.

La metodología que se emplea en el análisis de software, es la obtención de resultados a través de cálculos usando métodos matemáticos. Los modos más utilizados en la actualidad van desde métodos semánticos, hasta las técnicas más actuales como *value-set analysis*. Para la comprobación de los resultados, se suelen apoyar en las comprobaciones por inducción matemática u otros sistemas similares [HJ03, NVB10].

2.3 Tipos de análisis

Para realizar pruebas que permitan la búsqueda de código erróneo se derivan principalmente tres tipos de análisis diferentes: dinámico (la búsqueda de errores y la comprensión del programa en tiempo

[7] http://www.mibsoftware.com/libmib/astring
[8] http://www.zork.org/safestr

de ejecución), estático (análisis del código fuente sin ejecutar el programa) y híbrido (mezcla de las dos técnicas anteriores).

En las siguientes secciones vamos a ver cada una de estas técnicas con detalle.

2.3.1 **Análisis dinámico**

Los procedimientos para realizar análisis dinámico de binarios ha mejorado mucho en los últimos años. Esto se debe en gran parte a las investigaciones que han sido desarrolladas por varios equipos de investigadores alrededor del mundo.

Los dos métodos principales utilizados en el análisis dinámico son *taint analysis* y *forward symbolic execution*. Uno de los más importantes avances en este campo fue desarrollado por NewSome y Song [NS05] utilizándose principalmente para la búsqueda de ataques que implicasen la sobrescritura de memoria, es decir *buffer overflows*. Para demostrar que esta teoría era funcional desarrollaron TaintCheck. Otro trabajo utilizando estas técnicas fue desarrollado por Brumley et al. [BNS+06], utilizándolo para la generación de firmas con el objetivo de buscar vulnerabilidades. Un año más tarde Yin et al. [YSE+07] utilizaron esta misma técnica para crear el software Panorama para la detección y análisis de *malware*.

Otro método bastante utilizado es monitorizar las llamadas a funciones, ya sea dentro del propio programa o al sistema operativo. Normalmente esta tarea se lleva a cabo mediante la monitorización o *hookeo* de la API de Windows [Neb00]. Estás técnicas también han sido muy utilizadas para el análisis de binarios, como el caso de Xu et al. [XSCM04] para detectar *malware* o Gliber et al.[GCEC12] para encontrar fugas de memoria.

El último método utilizado es el de controlar los parámetros que son introducidos en las funciones para intentar saber los valores que son pasados estas a la pila. Este método es muy similar al realizado por el análisis estático, pero en este caso se centra en conocer los valores reales introducidos en la pila. Este método fue usado por Bayer et al.[BMKK06] para la detección de patrones de binarios

maliciosos.

Aunque en los últimos tiempos el análisis dinámico está ganando adeptos, los altos costes computaciones que se necesitan para realizar este tipo de análisis hace que a día de hoy todavía se sigan utilizando los análisis estáticos para realizar las labores de inspección de código. En algunas ocasiones los resultados conseguidos mediante esta técnica no podrán ser utilizados hasta que los ordenadores actuales no aumenten mucho su potencia de calculo para poder realizar tareas más pesadas.

2.3.2 **Análisis estático**

El análisis estático es un tipo de inspección de código en el cual no es necesario ejecutar la aplicación para saber como esta puede llegar a funcionar (al contrario de lo que sucede con el análisis dinámico, que sí necesita ejecución). Una de los principales beneficios que tiene este tipo de análisis es que puedes llegar a inspeccionar todo el código de la aplicación, incluso aunque haya partes éste muertas que nunca se puedan ejecutar en condiciones normales.

La principal ventaja del análisis estático es que al no tener que ejecutar el código es más seguro, y en muchos casos más rápido que el análisis dinámico. El hecho de no tener la necesidad de ejecutar el código fuente también trae más seguridad al investigador. Este detalle es muy importante, sobre todo en el caso de los investigadores que trabajan con *malware*, los cuales ven reducido el riesgo a infectarse. En el caso del análisis dinámico siempre existe esa posibilidad, ya que algún tipo de *malware* es capaz de detectar si se está llevando a cabo un análisis dinámico del binario y de evitar estas medidas para infectar la máquina.

Otra ventaja del análisis estático es que analiza el código sin necesidad de disponer de un sistema entero con la misma plataforma sobre la que fue construido el binario que se va a analizar. Incluso dependiendo del tipo de análisis que se esté realizando es posible que se asignen valores a los datos dentro del código haciendo una ejecución simulada y detectando más errores que siguiendo simple-

mente el flujo de control. Las llamadas a las funciones o a la API de Windows pueden ser seguras o inseguras dependiendo de cuales son los parámetros que se le pasan a estas llamadas. En el análisis realizado por OWASP en 2013 [OWA13] en el apartado sobre el *Uso de Componentes con Vulnerabilidades Conocidas*, demuestra que es uno de los grandes peligros actuales dentro de la seguridad informática englobándolo dentro del «Top 10» de los riesgos de seguridad más importantes.

Debido a que es bastante fácil la búsqueda de patrones dentro del software se han creado diversos algoritmos para buscar estos patrones, ya se para la búsqueda de vulnerabilidades, *malware*, similitud de código y otras muchas aplicaciones [CJS+05, LS92, CJ06].

2.3.3 Análisis híbrido

Este tipo de análisis utiliza el técnicas dinámicas para localizar eventos dentro del programa, y técnicas estáticas para analizar como funcionan estos eventos.

Esta técnica ha sido utilizada por varios programas que por un lado realizan un análisis dinámico emulando una máquina, para luego realizar un análisis estático que realiza las comprobaciones más rápidamente. Un ejemplo de este tipo de análisis los podemos ver en DSD-Crasher [CSX08] (para encontrar vulnerabilidades en el código) o la plataforma utilizado por Rus [RRH03] para realizar análisis de la memoria.

2.4 Algoritmos de análisis estático

Debido a que esta tesis se centra en el análisis estático, nos vamos a centrar más en el estudio de esta técnica. Por esta razón veremos cuales han sido los principales algoritmos que han sido creados para realizar este tipo de análisis.

2.4.1 Lógica de Hoare

Uno de los primeros pasos en la mitigación de errores fue realizada por C.A.R. Hoare [Hoa69] el cual creo la *lógica de Hoare* con las contribuciones de Robert Floyd.

La lógica de Hoare se basa en la terna «{Q} S {R}». Q y R son la precondición y postcondición necesaria que se debe cumplir para que el programa S funcione correctamente. De esta forma, si se cumple que el programa se inicia en un estado Q y termina en un estado R, el programa habrá funcionado correctamente. Aunque no mitiga los errores directamente, esta técnica es capaz de evitar muchos problemas al hacer más segura la programación gracias a una correcta especificación.

Un ejemplo de la está lógica la podemos observar en este ejemplo:

$$\{Q \equiv 1 \leq n \wedge n \leq 1000\}$$
$$fun\ \ maximo(a : vect; n : entero) dev(x : entero)$$
$$\{R \equiv (\forall \alpha \in \{1..n\}.x \geq a[\alpha])\}$$

Podemos ver como ha especificado una función para calcular el máximo de un vector de enteros. En esta forma observamos que la variable «Q» define como debe ser «n» (ni menor que uno ni mayor que 1000). En la parte central vemos la especificación de la función con sus variables de entrada y salida y los tipos de dichas variables. En la parte final podemos analizar la postcondición «R», en la cual observamos que tiene que devolver el mayor valor de todo el *array* que está siendo analizado.

Edsger W. Dijkstra [Dij75] ideó una extensión de la lógica de Hoare, con la semántica de transformación de predicados, donde estableció el concepto de la precondición más débil:

$$wp(\mathcal{S}, \mathcal{R})$$

En donde S es una secuencia de comandos a cumplir y R una postcondición que debe cumplirse. El resultado de esta fórmula es

el valor mínimo o «más débil» que puede tener una variable para que programa termine de forma correcta.

2.4.2 Shape Analysis

En 1967 Reynolds [Rey67] crea *shape analysis*, técnica que surge para analizar el *heap* y saber como los punteros acceden a la memoria. Este análisis se realiza usando unas gráficas llamadas *shape graphs* o *alias graphs*, en las cuales se muestra la relación entre los punteros y las posiciones de memoria que están apuntando.

La ventaja de esta técnica es la claridad en la representación de resultados, ya que en un vistazo rápido a la gráfica puedes saber como son las relaciones entre los punteros y las posiciones de memoria a las que apuntan.

Un problema de este sistema es que no es sensible al flujo del programa, con lo que es posible que no detecte muchas vulnerabilidades que pueden ir asociadas a este hecho. Esto hace que el *shape analysis* se vuelva impreciso, y en muchas ocasiones puede generar errores que nos hagan creer cosas diferentes a las que en realidad haría el programa.

Debido a sus ventajas *shape analysis* ha sido ampliamente utilizado y ha dado lugar a varias variantes como *shape boundary* [Gos85], utilizando la transformada de Fourier [PF77], *Medial axis transformations* [MS70] o *shape decomposition* [KPK87].

2.4.3 Abstract Interpretation

Abtract interpretation es una aproximación semántica a la estructura del programa. Fue creado por Cousot [CC77] como un método para realizar análisis estáticos del programa. Sus principales aplicaciones son para decidir donde realizar optimizaciones y transformaciones, y la detección de errores.

Mediante *abstract interpretation* se puede puede analizar una aplicación en base a un dominio de valores abstractos, es decir, las variables del programa analizado tendrán una serie de valores para

las variables internas. Así se consigue una versión abstracta de la aplicación con datos abstractos introducidos en las variables. Gracias a esta visión abstracta no es necesario ejecutar el programa en ningún momento, consiguiendo un visión más amplia del funcionamiento del programa.

Abstract interpretion ha demostrado ampliamente que tiene bastantes aplicaciones en el terreno de la seguridad con múltiples estudios desde su aparición en 1977. Algunos de los más representantes en este terreno pueden ser la aportación para la optimización, análisis y verificación realizada por Hermenigildo [HPBLG05] en 2005, la búsqueda de erres en aplicaciones web realizada por Tripp et al. [TPC+13], la ofuscación de código que publico Dalla [DPG05], la búsqueda de *malware* que llevo a cabo Preda [PCJD07] en 2007, o más recientemente detectando código metamórfico [DPGD15].

2.4.4 **Value-Set analysis**

Balakrishnan introdujo la primera forma de reconocer los valores en la memoria mediante *value-set analysis* [BR04]. Esto es una forma abstracta de analizar un ejecutable para saber que valores va a tomar cada variable en un punto del programa.

Un ejemplo claro de éxito del *abstract interpretation* es la aplicación realizada por Balakrishnan llamada CodeSurfer[9], que surgió del estudio realizado por él sobre la memoria del sistema [BR04]. Codesurfer es una herramienta de verificación de errores en los programas realizados en los lenguajes de programación C y C++ para plataformas x86. Es capaz de mostrar un grafo con todas las variables y las llamadas a funciones, pudiendo analizarlo gracias a un sistema de navegación por el código rápido e intuitivo.

VSA lo podemos describir como un analizador linea a linea de lo que existe en los registros y en las variables en memoria (en la pila, que llama *a-locks*). Un ejemplo de VSA en un punto determinado de una función podría ser el que vemos en la figura 2.2.

En dicha figura, VSA comprueba el valor de los principales regis-

[9] http://www.grammatech.com/research/technologies/codesurfer

$$esp \rightarrow (\perp, -44), var_1 \rightarrow (0, \perp), var_2 \rightarrow (0, \perp), eax \rightarrow (\perp, 4[1, \infty])$$

Figura 2.2: Ejemplo de *Value-Set analysis*

tros y de las variables declaradas. En este ejemplo, el registro ESP está apuntando a un valor que es -44, menos el valor de la base de la pila. La variable «var_1» es la primera variable global, lo mismo que sucede con la variable «var_2», que en este caso es la segunda variable global. El registro EAX contiene el valor de la primera variable global, la cual va de 1 hasta infinito, ya que no se le ha declarado ningún limite en el código fuente.

Para hacer el trabajo más fácil a VSA, este usa IDA Pro para saber las direcciones conocidas y los *offsets* de la pila, información sobre los límites de los procedimientos y sobre las llamadas a las librerías del sistema (usando la tecnología FLIRT de IDA Pro). Con todo esto, hace una pequeña base de datos en los cuales se basa luego para hacer los análisis posteriores.

Balakrishnan también publica otro trabajo [BR10] en el cual desconfía de que los compiladores hagan correctamente su trabajo. Realiza pruebas con las cuales llega a la conclusión de que, aunque un programador realice el código perfectamente, el compilador puede cometer errores en el tiempo de optimización y hacer que se produzcan fallos en el programa. Por esta razón realizó VSA, y se basa en el código ensamblador devuelto por el compilador para realizar su análisis y asegurarse de que el compilador ha realizado correctamente el trabajo.

Como vemos, en los últimos cinco años la popularidad de *value-set analysis* ha aumentado mucho y ha sido usado como base para grandes productos como BAP han visto que este tipo de análisis tiene grandes ventajas. Por todo esto parece que está técnica tiene un gran futuro por delante.

2.5 Lógicas temporales

La lógica tal y como la conocemos hoy en día es una parte de la filosofía, y sus comienzos datan de la época de Aristóteles. Esto hace que se considere una de las disciplinas más antiguas que se ha mantenido hasta nuestros días. Sin embargo, la parte de la lógica que se introduce en el mundo de la informática es relativamente joven.

La lógica matemática fue introducida mucho más cerca de nuestro tiempo que de los primeros pasos de la lógica en la Grecia antigua. Para ser exactos fue en el año 1887 cuando Peano [Pea87] empieza a hablar de lógica matemática y introdujo los símbolos de unión $\cup$ e intersección $\cap$. Los siguientes pasos en este campo fueron realizados por Frege [Fre79] en 1879 que fue el primero en pensar que las matemáticas son reductibles a la lógica y mas tarde, en 1910, Russel y Whitehead [RW68] donde ya sentaron las bases de la lógica en tres libros llamados «Principia Matemática», los cuales convertían en axiomas gran parte de los conocimientos matemáticos de la lógica.

Las lógicas temporales son una parte de la lógica que incluyen una serie de operadores especiales para las operaciones relacionadas con el tiempo. En la lógica temporal la linea temporal puede ser mostrada de dos formas diferentes: como una linea o como un árbol con sus diferentes ramas. Los orígenes de la lógica temporal datan del año 1977, año en el que Pnueli [Pnu77] crea la lógica LTL (*Linear Temporal Logic*), marcando el comienzo de una nueva era para la lógica matemática. Años más tarde, en 1982, Clarke et al. [CE82] crea otra lógica que va a revolucionar la forma en la que se veía la linea temporal, CTL. En esta lógica el tiempo se ve como un árbol en el que una linea temporal puede tener una bifurcación de la cual se dividen dos caminos diferentes. Ese mismo año, aparece una técnica que parece juntar las dos lógicas anteriores. Esta nueva lógica creada por Emerson y Halpen [EH85] recibe el nombre de CTL* (pronunciada CTL *star*). Con esta nueva lógica se puede llegar a realizar operaciones que no pueden ser realizadas con las otras dos lógicas, añade complejidad en el proceso de verificación. Estás lógicas son las más utilizadas por los *model checkers* actuales,

normalmente implementando LTL o CTL, y en algún caso especial ambas lógicas.

Aunque las lógicas que hemos visto hasta ahora son las utilizadas en la actualidad existen otras lógicas con un reconocimiento muy similar a las anteriores, como es el caso del calculo μ creado por Kozen [Koz83] en 1983. De la lógica CTL han surgido varias lógicas derivadas: Timed CTL [Bou09], Fair CTL [EL85] o Action CTL [DNV90]. Además de las lógicas derivadas de CTL existen otras que también están siendo muy utilizadas en el mundo académico. Entre ellas podemos destacar la lógica de Hennessy-Milner [HM85], que tiene la particularidad de tener tener propiedades para los estados que componen la estructura a examinar.

Ahora vamos a ver con más detenimiento las principales lógicas temporales sobre las cuales están utilizando los *model checkers* más importantes.

2.5.1 **LTL**

La creación de LTL por parte de Pnuelli [Pnu77] en el año 1977 no hubiera sido posible sin apoyarse en el trabajo de otros grandes científicos. Lewis [Lew12], Kripke [Kri63], Kamp [Kam68], y Prior [Pri03] han sido los trabajos más importantes sobre los cuales la lógica temporal se ha formado, ya sea por usar sus estructuras de datos o por sus razonamientos.

Los primeros pasos de la lógica temporal no estaban orientados hacía la búsqueda de errores, sino que servía para hacer razonamientos sobre hecho que podían pasar, o no, sobre una linea temporal. Los primeros en pensar en está lógica para este propósito fueron Manna y Pnuelli [MP92] en el año 1992. En su trabajo eran capaces de deducir si el sistema que estaban analizando podía tener peligros que afectaran a su funcionamiento buscando estados inconsistentes.

Los componentes de LTL no difieren mucho de los de la lógica tradicional, conteniendo proposiciones atómicas y los operadores booleanos. El verdadero cambio se produce en los nuevos opera-

dores de tiempo que introduce. Vamos a ver cada uno de ellos con detalle.

- El operador «$\bigcirc$» significa «siguiente estado». Esto significa que la operación «$\bigcirc\varphi$» sera verdadero si en el segundo estado φ es verdadero.
- «U» es el operador «hasta», con lo cual «$\varphi U\psi$» indica que φ satisface el estado actual hasta que en un momento dado se cumple ψ.
- El operador «$\Box$» significa «siempre», con lo cual «$\Box\varphi$» es que φ se cumple durante todos los estados del camino.
- Y por último el operador «$\Diamond$», que significa «eventualmente», lo que quiere decir tenemos «$\Diamond\varphi$» entonces tenemos que «φ» se cumple en algún momento del tiempo futuro del camino.

La sintaxis de LTL también es bastante similar a la de la lógica proposicional. La sintaxis es la siguiente:

$\phi ::= true|false|a \in AP|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|\phi_1 \rightarrow \phi_2|\phi_1 \equiv \phi_2|\phi_1 U \phi_2|\bigcirc\phi|\Box\phi|\Diamond\phi$

La utilización de esta lógica (donde AP significa el conjunto de las proposiciones atómicas) está limitada a sistemas en los cuales un evento en el sistema (representado por un estado) solo tiene otro posible evento seguido en el tiempo. Es decir, es como trazar una linea temporal con sucesos que van a ir sucediendo dentro del sistema que se está analizando.

En el año 2002 se produjo un gran cambio dentro de la lógica LTL con el trabajo de Laroussinie et al. [LMS02]. En este trabajo se presento una nueva lógica llamada NLTL, que tenía una característica que no había sido vista hasta el momento de la publicación, la valoración de momentos de tiempo ya sucedidos, es decir, valorar el pasado. La inclusión de la valoración de estados del sistema que ya han sucedido durante un tiempo ha conseguido que se puedan verificar modelos que antes no se podían verificar, pero lamentablemente también ha conseguido que la complejidad sea mayor comparado con LTL. Incluso con este problema, la expresividad no se ha

visto afectada por la incorporación de los nuevos operadores que analizan el pasado. En los últimos años se han llevado a cabo varios estudios utilizando esta técnica en múltiples ocasiones, incluso analizando código binario [RBH+11], evaluando compiladores [XF12] o incluso Song [ST13a, ST14] (autor de SCTPL) lo utilizó para la detección de *malware*.

Existen varios programas capaces de realizar análisis utilizando LTL para comprobar diversas características del programa. Uno de los más reconocidos es NuSMV[10]. Este programa fue desarrollado por Cimatti et al.[CCGR99, CCG+02] a finales de la década de los noventa. Esta herramienta esta basada en SMV la primera herramienta capaz de realizar *model checking* sobre árboles binarios. La mejora que realizó Cimatti sobre SMV fue la inclusión de un solucionador del problema de satisfacibilidad booleana (en inglés «*Sat Solver*») y la librería creada por Somenzi para resolver los árboles de decisión binarios [Som12]. NuSMV usa tanto LTL como CTL para realizar sus cálculos de *model checking*.

Otro programa importante dentro de la comunidad de *model checking* y que utiliza LTL para realizar sus cálculos es SPIN[11] desarrollada por Holzman [Hol04] en el año 1980 y que todavía se encuentra en desarrollo en la actualidad después de haber liberado su código en la década de los noventa. Un importante avance para este *model checker* fue la inclusión del algoritmo desarrollado por Gerth et al.[GPVW95] y que ayuda enormemente en la tarea de la comprobación de las fórmulas mediante *model checking*. La rapidez de este algoritmo radica en el hecho de que su comprobación se basa en tablas que resuelven los procesos del *model checking* sobre la marcha.

2.5.2 CTL

En el año 1982 Emerson y Clarke [CE82] publicaron un trabajo en el cual introducía una lógica que no había sido vista hasta ese momen-

[10] http://nusmv.fbk.eu/

[11] http://spinroot.com/spin/whatispin.html

to, CTL. Emerson siguió trabajando para conseguir una completa axiomatización de CTL y ese mismo año publicó junto con Halpen [EH85] unos nuevos axiomas. Pero no fue hasta el siguiente año cuando Ben-Ari et al. [BAPM83] publicaron las reglas completas de axiomatización de esta lógica.

Como podemos ver en la figura 2.3 mediante CTL es posible conocer si una condición se va a satisfacer a lo largo del tiempo, aunque existan varios posibles futuros. Existen varios tipos de operaciones que no existían en la lógica proposicional normal. CTL introduce los operadores existenciales $\exists$ y Universal $\forall$, para determinar si una condición se cumple en uno o varios caminos del grafo.

Los operadores que se introducen en la lógica CTL son los mismos que en LTL ($\Diamond\psi$, $\bigcirc\psi$, $\Box\psi$, $\psi U\phi$) para realizar operaciones con los caminos. El operador $\Diamond\psi$ sirve para indicar si un camino va a cumplir una determinada propiedad en un momento dado. El operador $\bigcirc\psi$ comprueba si una propiedad se va a cumplir en la segunda posición del árbol. El operador $\Box\psi$ comprueba si un camino cumple una determinada propiedad en todos los estados del grafo. Y por último la operación $\psi\mathcal{U}\phi$, el cual comprueba se cumpla ψ y que posteriormente se cumpla ψ.

En CTL estos operadores no pueden ir separados de los cuantificadores existencial y universal, con lo cual las operaciones siempre tiene que ser como las que hemos podido observar en la Figura 2.3.

2.5.3 CTL*

CTL* (pronunciado CTL estrella) fue creado por Emerson y Clarke [EH86]. Esta lógica es más expresiva que las lógicas temporales vistas hasta ahora, y por esta razón se pueden crear fórmulas que pueden ser realizadas por CTL* y no son realizables por ninguna otra lógica. Por ello puede decirse que CTL* fue creada directamente de LTL y CTL, porque contienen los operadores lineales de LT y los operadores de camino de CTL.

Aunque CTL* es más completa que otras lógicas temporales, no ha sido prácticamente usada en para el *model checking*. Lo que llama

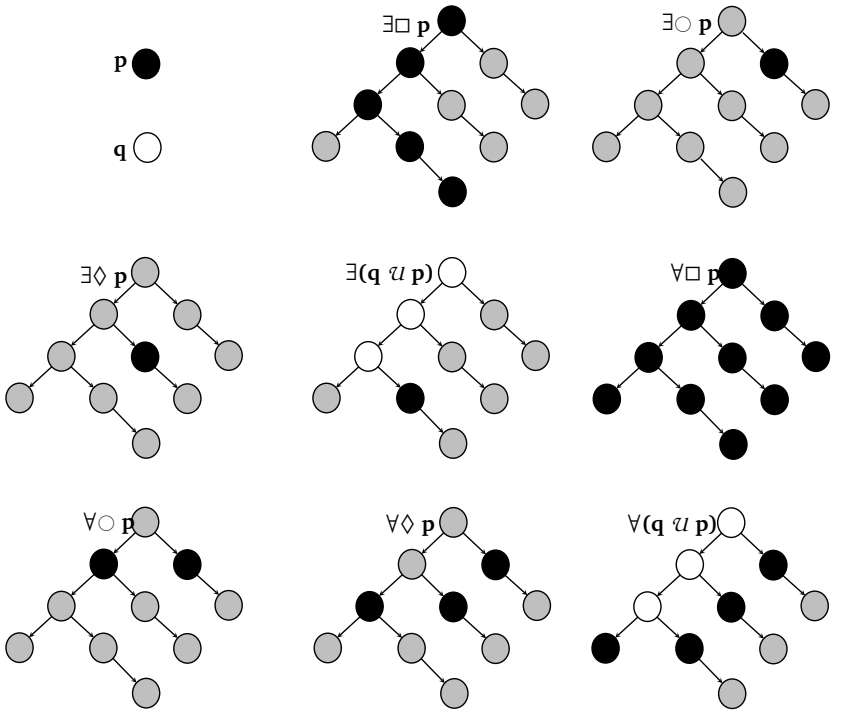

Figura 2.3: Representación gráfica de las fórmulas CTL, donde el estado negro satisface a p, el blanco satisface a q, y el gris muestra cualquier estado que no satisface ni p ni q.

la atención ya que la complejidad computacional de CTL* es exactamente la misma que LTL, PSPACE, hecho que ya demostró Lohmann [Loh07] en 2007.

2.5.4 CTPL

CTPL es una lógica temporal creada por Kinder et al. [KKSV05] para la búsqueda de *malware*. La sintaxis utilizada por Kinder para encontrar software malicioso es similar a la utilizada por CTL, pero los estados de Kinder son instrucciones ensamblador.

Además de eso Kinder utiliza `loc(L)` para controlar el orden el cual los argumentos son pasados a la pila. Esta instrucción viene adquirida del uso de IDA Pro. Este desensamblador divide las zonas

de memoria con saltos y a cada una de estas regiones les pone el prefijo «loc». Por esta razón Kinder hizo esta distinción también en su código, y de esta forma controla mejor todas las regiones de la memoria del proceso.

Un ejemplo de la sintaxis de CTPL puede ser el siguiente:

$$\text{EF (mov eax, 937} \wedge \text{AF(push eax))}$$

Figura 2.4: Ejemplo de fórmula CTPL.

En este ejemplo se busca que se encuentre una instrucción `mov eax 937` y más adelante en el código se encuentre una instrucción `push eax`, sea cual sea el camino de ejecución del binario.

Kinder et al. también introducen los cuantificadores existencial y universal para referirse a los posibles estados que puede tener un determinado registro, pudiendo ejecutar fórmulas como las que vemos en la Figura 2.5. En ella podemos ver la comprobación para saber si existe alguna orden que introduzca un cero en un registro y que más tarde en el programa dicho registro sea pasado a la pila.

$$\exists r \text{ EF (mov (r, 0))} \wedge \text{EF (push (r))}$$

Figura 2.5: Ejemplo de fórmula CTPL con cuantificador existencial.

Mediante esta lógica Kinder fue capaz de encontrar diversas variantes de tres tipos distintos de *malware* (NetSky, Mydoom y Klez) con una sola fórmula, demostrando de esta forma el potencial que tiene CTPL para la búsqueda de este tipo de software.

Song [ST12] [ST13b] creo en el año 2012 SCTPL una extensión de CTPL, la cual permitía hacer predicados con la pila. Con esto era posible aumentar la tasa de detección del *malware*, llegando a superar a muchos antivirus comerciales.

2.5.5 μ-calculus

El cálculo μ puede entenderse como una extensión de CTL. Fue creado por Kozen [Koz83] en 1983. El cálculo μ usa las mismas operaciones que las que se usan en CTL, pero con alguna diferencia. Una

de las principales diferencias es que en vez de usar los operadores existenciales y universales, el calculo μ utiliza $\langle\ \rangle$ que cumple la función de operador existencial y $[\,]$, que realiza la misma función que el operador universal. Con lo que podemos hacer una rápida equivalencia: $\exists \equiv \langle\rangle$ y $\forall \equiv [\,]$.

Aunque la principal diferencia entre CTL y el cálculo μ es el uso de puntos fijos. El cálculo μ tiene un punto fijo mínimo (μ) y uno máximo (ν), gracias a lo cual es posible dar una propiedades de corrección.

Las lógicas basadas en puntos fijos como el cálculo mu han tenido fama de ser incomprensible, y existen expertos que han pasado horas intentando comprender el cálculo μ, como indicaron Bradfield y Stirling [BS01]. Aun así, también indican que gran parte de las matemáticas necesarias para entender correctamente este campo no está cubierto por las universidades, haciendo que este aprendizaje más difícil.

El cálculo μ al igual que el resto de lógicas temporales también ha sido aplicado en *model checkers* para la búsqueda de errores, como por ejemplo: XMC [DSSRS02], mCRL2 [CGK+13] o Edinburgh CWB[12], los cuales solamente utilizan esta lógica para realizar sus comprobaciones.

2.6 Model checking y los contraejemplos

La técnica en la cual nos vamos a centrar más en este estado del arte es la técnica de *model checking* y más concretamente en los contraejemplos. Los contraejemplos son el ejemplo más parecido al estudio que vamos a llevar a cabo, y no solo devuelven valores booleanos, con lo cual la información ofrecida es mayor. Uno de los detalles que estamos buscando en esta investigación.

Las bases del proceso de *model checking* fueron establecidas por el trabajo de dos grupos de científicos: Emerson y Clarke [CE82, CES86, EC80] y Queille y Sifakis [QS82]. Aunque estos fueron los

[12]http://homepages.inf.ed.ac.uk/perdita/cwb/

predecesores del *model checking* al comienzo de este no se utilizaba este proceso para la detección de errores en el software. Esto no llego hasta el año 1990 en el cual Holzman [Hol90] advirtiendo de los problemas que traía consigo realizar pruebas de forma manual, hizo un sistema para automatizar dicho proceso mediante *model checking*.

El *model checking* es un sistema al que se le pasa una propiedad formal y se comprueba que esta satisface un sistema de transiciones. Éste representa al que se desea analizar, ya sea este hardware o software. Una propiedad formal es una fórmula que describe el funcionamiento del sistema, como las que hemos visto en las descripciones de las fórmulas en las secciones anteriores de este mismo capitulo. Por otro lado, el sistema formal es una descripción de los estados que tiene el sistema a analizar junto con las relaciones que existen entre dichos estados. Esta descripción normalmente se representa mediante un diagrama de estados que muestra claramente y de forma gráfica estas relaciones, aunque también se dan casos en los cuales estos sistemas formales pueden representarse mediante tablas.

Desde que Holzmann [Hol90] comenzó la pruebas sobre el software para la búsqueda de posibles errores en el proceso de programación, otros trabajos siguieron sus esfuerzos. Estaba claro desde el primer momento que las pruebas manuales no eran del todo precisas por dos razones particulares: la primera, porque el volumen de software a analizar es muy grande lo cual requería muchos recursos, tanto económicos como de personal cualificado. Y la segunda, porque al ser pruebas manuales realizadas por personas induce a la introducción de errores haciendo que se pasen por alto muchos errores.

Por todas estas razones, la automatización de los procesos de pruebas se convierte en uno de los principales puntos a tener en cuenta cuando se habla de la búsqueda de errores en el software. Los primeros en proponer el uso de automatizar estas tareas usando *model checking* fueron Callahan et al. [CSE+96] y Engels et al. [EFM97].

Para realizar dichas pruebas de software se precisan una serie de

pasos previos al comienzo de la automatización. Uno de los procesos más complicados es el de realizar un modelo abstracto del sistema sobre el cual se van a realizar las pruebas. Cada uno de los estados que contiene el sistema debe de estar representado en el diagrama o tabla que va a representarlo. En este área muchos investigadores ha realizado varios acercamientos para poder resolver gran parte de estos problemas, sobre todo en la forma de representar estos modelos.

Como hemos visto en las secciones anteriores, *model checker* es un herramienta para realizar verificaciones formales. En la lógica tradicional podemos introducir una propiedad y esta herramienta nos devolverá si ésta satisface el sistema que se desea examinar o no. Con lo cual las salidas típicas del *model checker* son verdadero o falso. En el caso de los contraejemplos se intenta hacer exactamente lo contrario, es decir, saber si una propiedad no cumple el modelo. En el caso de que esto ocurra, el *model checker* devolverá los caminos que han incumplido esta propiedad.

Esta técnica aplicada a la verificación puede no llegar a ser práctico en entornos reales. Las aplicaciones informáticas no solo dependen del flujo de ejecución que el programa dicta en su programación, y puede ser que por diferentes cambios en el hardware o software ajeno a él. Casi todas las validaciones se efectúan sobre el código fuente que está desarrollando el programador con lo cual el compilador y el propio sistema operativo pueden hacer que la aplicación falle sin que el programador haya introducido ninguna línea errónea en su código. Además la verificación de código no siempre puede ser aplicada, si el tamaño es muy grande la validación del modelo puede ser muy costosa en términos tanto de recursos del sistema que hace el *model checking* como en el tiempo necesitado para realizar las comprobaciones.

La mejor forma de utilizar los contraejemplos en el *model checking* es de interpretarlos como casos de uso para las realizar más pruebas. Cuando se prueba un software lo ideal es conseguir varios grupos de pruebas para hacer que sean lo más completas posibles. Por esta razón, uno de los mayores retos cuando se realiza *model checking* sobre un programa es buscar un conjunto muy grande de

pruebas para que el programa esté correctamente validado.

Aunque después de casi veinte años de investigaciones se han conseguido hacer muchos avances dentro de este campo, aun queda mucho trabajo por hacer. Los grandes adelantos conseguidos sobre todo en el campo de las optimizaciones dentro de los casos de uso han hecho que cada día el *model checking* se haga más aplicable a entornos productivos donde el tiempo de realización de las pruebas es vital.

Los sistemas temas embebidos, los cuales son totalmente dependientes de sucesos externos, han adoptado con más facilidad esta técnica [Bro05]. Estos sistemas por su forma de actuar hacen que las pruebas manuales sean más difíciles de ejecutar, mientras que para realizar *model checking* se pueden llegar a establecer unos límites que el sistema siempre reconozca. Esto unido al hecho de que en los últimos años se ha conseguido abstraer bastante muchos procesos ha producido que se estén llegando a realizar verificaciones en tiempo real. Aun con todos estos adelantos, existen muchos problemas que no han sido superados y que generan que todavía no se haya adaptado del todo esta técnica dentro del sector empresarial. Sin embargo, sigue habiendo problemas con este tipo de sistemas. En las próximas secciones hablaremos de cada uno de los problemas que tiene el *model checking* y los contraejemplos en la actualidad.

2.7 Sistemas de transiciones

Los sistemas de transiciones son modelos que describen los comportamientos de un sistema. Como hemos explicado con anterioridad, normalmente suelen ser grafos dirigidos, donde cada nodo representa un estado en el que puede llegar a estar el sistema, y las líneas que conectan los nodos son las posibles transiciones que pueden realizar los estados entre si.

El primer sistema de transición de estados fue creado en 1976 por Keller [Kel76], y fue rápidamente adaptado por la comunidad para la representación de sistemas.

En estos sistemas las transiciones hacen que el sistema pase de

un estado a otro de forma secuencial, es decir, un sistema nunca podrá estar en dos estados a la vez. Las transiciones en éstos son siempre dirigidas haciendo que se vea claramente cuales son los siguientes estados que un estado puede llegar a alcanzar.

Los estados en los sistemas de transición pueden representar cualquier cosa, lo cual los hace muy versátiles. Ejemplos de estados los podemos encontrar en la lógica CTPL donde cada estado es una instrucción ensamblador.

Los sistemas de transición normales son una tupla (S, Acc, →, I, AP, L), que significan lo siguiente:

S: Conjunto de estados

Acc: Conjunto de acciones

→: Subconjunto de S ×Acc ×S y es un conjunto de relaciones de transición.

I: Subconjunto de S y son los estados iniciales.

AP: El conjunto de las proposiciones atómicas

L: S → 2^{AP} son las etiquetas de cada estado.

Como podemos observar de esta manera es muy sencillo reproducir cualquier tipo de sistema. Una vez representado también resulta muy fácil ver como este método va recorriendo los estados, conociendo como son las distintas ejecuciones que se pueden llevar a cabo en la estructura que esta representada por este sistema de transiciones.

Dentro de la terminología del *model checking* se llama **ejecución** a una secuencia de estados que comienzan en un estado S_0 y terminan en otro estado S_n realizando acciones diferentes durante la ejecución. Por ello, si llamamos ST = (S, Acc, →, I, AP, L) un sistema de transición, podemos decir que una ejecución en este sistema es:

$$\varrho = s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 ... \alpha_n s_n$$

tal que:

$$s_i \xrightarrow{\alpha_{i+1}} s_{i+1}.\forall(0 \leq i \leq n).$$

Un ejemplo claro de el triunfo de los sistemas de transiciones para formalizar sistemas se encuentra en los métodos concurrentes basados en la comunicación entre procesos. Ejemplo de ello son las álgebras que han salido para el proceso de Handshaking, como ACP [BK85], CSS [Mil89, Mil99] (o su variante SCCS [Mil83], CAP [H$^+$85, Hoa78] o LOTOS [BB87].

Dentro de estos sistemas de transición el más famoso dentro del *model checking* es la estructura Kripke [Kri63]. El sistema interno de ésta estructura es muy similar al de otros sistemas de transiciones, como podemos observar a continuación:

- $S = \{s_i\}_{i=1}^{n} = \{s_1, s_2, \ldots, s_n\}$ es un conjunto de estados.
- $I \subseteq S$ es el conjunto de estados iniciales.
- $R \subseteq S \times S$ son las relaciones de transición entre estados, es decir, que para cada $s \in R$ existe un $s' \in S$ tal que $(s, s') \in R$.
- $K : \mathscr{S} \rightarrow 2^{AP}$, es donde se etiqueta la función, haciendo un mapa con cada estado y el contenido de este.

Un **camino** $p := (s_0, s_1, ...)$ de una estructura Kripke K es una secuencia infinita tal que $\forall i \leq 0 : (s_i, s_{i+1}) \in T$ para K. Una secuencia de ejecución de este modelo es un *camino*. Como los caminos son infinitos los bloqueos mutuos no pueden ser modelados directamente dentro de las estructuras Kripke. Esto es, sin embargo, posible si lo hacemos mediante un bucle de un estado. Una estructura Kripke define todos los posibles caminos de ejecución de un sistema. Definimos $Caminos(K, s)$ sea el conjunto de caminos de la estructura Kripke K que empiece en el estado s. Usamos $Caminos(K)$ como abreviación para $Caminos(K, s)|s \in S_0$.

2.8 Verificación

Uno de los grandes objetivos del *model checking* es el la verificación de un sistema de transición, es decir, comprobar si un modelo cumple una determinada propiedad. Se podría decir que mediante *model checking* se pueden aplicar las matemáticas a las tareas de verificar software y hardware. Por esta razón se han creado múltiples estudios en los cual se han utilizado desde diferentes lógicas temporales hasta estructuras de datos optimizadas, ya que cualquier método de verificación basado en *model checking* es tan bueno como lo ha sido el modelado del sistema que se desea analizar. Cuando hablamos de contraejemplos sabemos que no devuelve si una propiedad es satisfecha o no, sino que devuelve el ejemplo de como la propiedad no ha sido satisfecha. En el caso de realizar contraejemplos con LTL es fácilmente verificable, ya que la linea temporal de esta es fácilmente reconocible, debido a que solo existe una. En cambio, en CTL existe otro problema que es el conocer cual es el estado que realmente satisface una propiedad en la estructura Kripke. Esto llevó a la derivación de las trazas mediante los contraejemplos [CGP99, CGMZ95].

Para LTL [LP85, VW86] la negación de la propiedad es representada por la violación de un autómata que acepta infinitas palabras (autómata de Buchi). Para LTL la forma de encontrar el contraejemplo no es más que conocer cual es el camino que vuelve hasta el primer estado del sistema, con lo cual esta operación es más o menos sencilla.

En CTL [CES83, QS82] este enfoque es más complicado ya que una vez que se encuentra un estado que viola una propiedad, es necesario calcular recursivamente las subformulas para cada estado. En este caso existen varios algoritmos para recorrer el árbol: en profundidad o en anchura. Ninguna de las dos soluciones es válida al cien por cien de los casos, ya que el algoritmo de búsqueda en anchura siempre devuelve el camino más corto pero la cantidad de memoria que necesita para hacer esto es muy superior a la que necesita la búsqueda en profundidad. Mientras que los costes de almacenamiento que necesita la búsqueda en profundidad son muy

inferiores a los que necesita la búsqueda en anchura.

Tanto en LTL como en CTL si se visitan todos los estados de un sistema de transiciones y ninguno de ellos viola la propiedad se dice que dicha propiedad es consistente para el sistema.

Las primeras aproximaciones que se hicieron hacía estos modelos de verificación se basaron en *explicit model checking*. La forma más correcta de realizar este modelado fue la de representar el espacio como explicito, una vez realizado esto, solo queda buscar un estado que viole la propiedad. Aunque existe otra técnica que mejora la velocidad de búsqueda de contraejemplos que tiene *expecific model checking*, *directed model checking* [ELL01]. Esta técnica es más valorada entre los que buscan la generación de casos de prueba, más que para los que buscan la corrección del modelo.

Directed model checking [ELL01] mejora *expecific model checking* con un nuevo modelo de búsqueda, en el cual se mejora la velocidad en la cual los contraejemplos son generados. Lo que hace que esta técnica sea más rápida es que cambia el orden en como los estados son explorados durante el recorrido de las ramas. Esta técnica es válida si lo que se busca no es la corrección del modelo, sino la generación de contraejemplos, lo cual puede ser una buena idea para la generación de casos de prueba.

Otra técnica interesante para la búsqueda de contraejemplos es *simbolic model checking*, creada por McMillan [McM93] en 1993. Esta técnica ya se encuentra dentro de la segunda generación de *model checking* debido a la utilización de diagramas binarios de decisión [Bry86] para representar los estados y las relaciones entre estos de manera más efectiva de lo que se hacia hasta ese momento. Este hecho hace que la representación de estados puede albergar sistemas más largos, aunque al aumentar el numero de los estados surgen otros problemas, el rendimiento es muy bajo y la ordenación de estados demasiado compleja. Aunque existen acercamientos a como se debería de ordenar los estados de forma más eficaz, Bryan [Bry92] llego a la conclusión de que el problema del ordenamiento óptimo de un sistema es NP-Completo.

Bounded model checking creado por Biere et al. [BCCZ99] es la

tercera generación dentro del *model checking*. Este nuevo sistema se define como un método de satisfacción de restricciones. Con los cual se pueden utilizar los solucionadores del problema de satisfacibilidad booleana (*SAT Solvers*) para calcular los contraejemplos dentro de unos límites. Este modelo se muestra bastante eficaz siempre y cuando los límites puestos para resolver el sistema no sean muy grandes. Gran parte de la aceptación de esta técnica se debe a que ha triunfado en sistemas donde el resto de la lógica tradicional ha fallado, normalmente por falta de recursos. Aun así, *bounded model checking* ha dado resultados pésimos en sistemas en los cuales los otros métodos de *model checking* tienen resultados aceptables. A pesar de todo algunos grupos de trabajo siguen investigando en este área, como por ejemplo, Ramalho et al. [RFS+13] que han utilizado ésta técnica para realizar *model checking* sobre programas escritos en C++, Günther y Weissenbacher [GW14], quienes varían los límites de ésta técnica en su nuevo estudio, o Sousa y Sen [SS13] que utilizan la representación intermedia de LLVM para realizar un estudio independiente del lenguaje.

En los últimos años *bounded model checking* ha adquirido bastante fama debido al trabajo realizado por Merz et al. [MFS12] con la herramienta LLBMC. Ésta utiliza LLVM, para transformar C y C++ a LLVM-IR, una vez conseguido esto simplifican el resultado devuelto por esta herramienta y devuelven una representación preparada para trabajar con lógica matemática.

Desde el punto de vista de las lógicas que se aplican en los *model checkers* también surgen problemas. En CTL, los algoritmos son algoritmos se aplican sobre trazas lineales, haciéndolo más parecido a LTL que al propio CTL. De esta manera es más sencillo comprender, desde un estado inicial, cual es el estado que viola la propiedad [CGMZ95]. Sin embargo, son muy pocos los algoritmos de lógica temporal que muestran los resultados en forma lineal, como ACTL^{det} o LIN. En la mayoría de los casos los contraejemplos lineales no son suficientes para probar una violación o satisfacción de la propiedad. Por esta razón se lo más común es utilizar las trazas no lineales para mostrar los resultados del contraejemplo.

De ahí surge la necesidad de crear un algoritmo de sacará con-

traejemplos como si fueran árboles, cosa que no sucedió hasta el año 2002 de la mano Clarke et al. [CJLV02]. En un trabajo relacionado Wijesekera et al. [WASF07] crea lo que el llama *evidence graphs* para CTL, que es una relación formal entre los casos de prueba y los contraejemplos. Al no ser lineales los grafos es necesario crear varios casos de prueba para que el árbol pueda ser recorrido entero.

Meolic et al. [MFG04] también crearon un acercamiento diferente al resto. En este caso, un autómata creaba un superconjunto de todos los contraejemplos que se podían crear en un espacio finito y lineal, aunque eso solo se llevaba a cabo para un conjunto limitado de CTL.

2.9 Problemas con los model checkers

Como hemos visto los *model checkers* son muy utilizados en la actualidad y es un área de investigación muy activo, debido a que el rendimiento de estos no es el adecuado para los ordenadores actuales y existen grandes problemas de rendimiento. Uno de los obstáculos que hay en el *model checking* actual, es la explosión de estados. Incluso, en el caso que el rendimiento llegase a ser aceptable los resultados podrían no llegar a ser adecuados. Existen casos como las pruebas de regresión en los cuales son necesarios tratamientos especiales, lo cual hace todavía más difícil la correcta ejecución de estos algoritmos dentro de un *model checker*.

Vamos a enumerar los principales problemas con los que se encuentran los *model checkers* en la actualidad a la hora de implementar los algoritmos conocidos.

2.9.1 Problema de la explosión de estados

Uno de lo problemas más grandes en el *model checking* es la larga cantidad de datos que tienen que ser manejados. Por esta razón pueden surgir muchos errores, pero el problema más grande es la necesidad de procesar todos los datos. Este dilema se llama el problema de la explosión de estados. Se lleva estudiando y buscando

soluciones a este hecho desde hace muchos años, como por ejemplo, en donde cabe destacar el trabajo de Clarke ([CGJ+01], [CG87]) en la materia.

Las posibles soluciones a este problema se pueden agrupar en tres categorías principales: *model checking* simbólico con diagramas de decisión binarios, reducción de orden parcial y la abstracción.

Uno de los campos en los cuales se ha buscado una solución con más ahínco de este problema ha sido en el *model checking* simbólico, como demostró Clarke et al [CKNZ12] en 2012. Burch et al. [BCM+90] empezaron a proponer las primeras formas de representar el el diagrama de estados de forma simbólica, en vez de explicita, y a partir de ese momento surgieron varios tipos de optimizaciones a ese proceso.

La reducción de orden parcial fue creado por Peled [Pel93] para intentar reducir el gráfico de estados para hacer *model checking*. Esta técnica permite las reducciones al aprovecharse de que, normalmente, en los sistemas existen transiciones que sin importar el orden en el que estén ejecutados siempre dan el mismo resultado. Desde su creación está técnica ha sido refinada y se ha implementado en múltiples trabajos, incluso aplicándose a la validación de software [YCGK07], o mejorando el propio algoritmo [ZRZM12, Sha14].

Alguno de estos acercamientos intentan abstraer los datos para hacerlo más prácticos a la hora de trabajar con ellos. Cousot [CC99] es uno de los que utiliza este enfoque de una forma acertada. Alur et al. [ADAHM99] primero analiza los datos, luego intenta abstraer los datos obtenidos en el paso anterior, y finalmente verifica dichas las abstracciones.

Cuando hablamos de *model checking* de software han surgido acercamientos que lo que hacen es particionar el código en varios trozos, haciendo que un sistema grande se transforme en muchos pequeños. Un ejemplo de esta técnica la podemos ver en el trabajo realizado por Valdiviezo et al. [VCK14], en que primero detectaban las posibles zonas en la que podía existir una vulnerabilidad y más tarde, la separaban de tal manera que cada zona solo podía tener un potencial defecto. De esta forma reducían considerablemente el

número de elementos a examinar.

Como hemos observado, este campo ha tenido múltiples adelantos para combatir este problema [KGWT11, RBB13, GKO15], pero a su vez crea otros problemas de los cuales hablamos en las próximas secciones.

2.9.2 **Abstracción**

Como hemos visto en el apartado anterior, gran parte de los problemas que tenemos dentro del *model checking* se reducen al problema de la explosión de estados. Para remediar este inconveniente se desarrolló la abstracción. Dentro del área de la abstracción se han producido grandes adelantos en los últimos años, debido en gran parte a que se cree que es una de las grandes soluciones para el problema de la explosión de estados. Estos adelantos han traído consigo que las formas de realizar correcciones hayan mejorado sustancialmente haciendo posible la verificación de sistemas mucho más grandes que los que se analizaban hace unos años. La abstracción está íntimamente relacionada con la verificación y por esta razón se intenta mejorar tanto en este campo, para que las pruebas se hagan más complejas sin afectar al rendimiento.

Uno de los adelantos más importantes que se realizaron en este campo ocurrió en el año 1999, cuando Amman y Black [AB99] definieron una nueva forma de crear casos de prueba basados en la solidez. Fueron ellos los primeros en decir que para que un *model checking* sea solido, cualquier contraejemplo que se produzca en el modelo abstracto debería de producirse también en el prototipo original. Por esta razón propusieron *finite focus*, el cual considera solo un conjunto limitado de estados dentro de un sistema, de esta manera es capaz de abstraer, por ejemplo, variables dentro de un sistema muy grande. Cuando se esta analizando, el estado de la máquina puede pasar a solido (que es analizado por *finite Focus*) o a no solido (el cual no es analizado). Debido a este cambio, una vez que la máquina entra en un estado no solido, ya no es posible pasar a un estado robusto, ya que no se puede asegurar nada dentro del sistema después del paso a no solido. Para ejecutar las pruebas

en los estados en los cuales el sistema no es solido Amman y Black reescriben muchos operadores de las lógicas CTL y LTL para poder evaluar el tiempo en dicho estado.

Uno de las plataformas para *model checking* que más han avanzado en el campo de la abstracción puede ser *CounterExample Guided Abstraction Refinemento* (CEGAR) [CGJ+00]. Esta técnica trabaja rebajando los modelos abstractos hasta que nos se encuentran más falsos contraejemplos que verifiquen una propiedad. De esta manera, Clarke y sus colaboradores se aseguraban que cuando una propiedad se cumple en el modelo concreto, también se va a seguir comprobando en el modelo abstracto. Aunque en realidad el objetivo de un *model checker* es bastante diferente a este sistema ya que lo que debería de comprobar es exactamente lo contrario, que las propiedades que son infringidas por un modelo concreto deberían de serlo por uno abstracto.

2.9.3 **Generación de pruebas**

Como es normal cuando se realiza la verificación de software la creación de casos de prueba es una tarea muy importante. Cuando hablamos de pruebas manuales es muy importante tener unos buenos casos de prueba para que los analistas hagan correctamente su trabajo. La generación de éstos puede ser un proceso laborioso pero al tener que ser introducido en estas herramientas el proceso es aún más delicado, ya que un error puede hacer que en las pruebas salgan resultados erróneos.

La generación de casos de pruebas es un proceso tan importante como la abstracción de la cual hemos hablado antes, y las investigaciones sobre este tema también son muy numerosas. La generación más eficaz de casos de prueba prácticamente se inició con el comienzo del siglo, cuando *directed model checking* [ELL01] realizando una generación de contraejemplos muy efectiva, aunque no realizaba prácticamente ningún tipo de verificación. Más tarde, Heimdahl et al. [HRV+04] crearon *bounded model checking*, que realizaba una generación de casos de prueba muy eficaz, pero marcando unos límites.

Los acercamientos basados en límites o mutaciones de los casos de prueba normalmente son bastante eficaces pero sobrecargan el *model checker*. De esto ya se se dieron cuenta varios autores, Hong y Ural [HU05], Fraser y Wotawa [FW07d], y Zeng et al. [ZML07], mostrando que este hecho no era lo suficientemente bueno, ya que la sobrecarga en muchos casos era excesiva.

Black y Ranville [BR01] crean otra forma de generar casos de prueba. En este caso, examinan los casos de prueba generados para buscar casos redundantes, algo que era muy común hasta ese momento. Aunque mejoró bastante el proceso, Fraser y Wotawa [FW07b] demostraron que a pesar de que los casos de prueba no hayan sido duplicados por otros casos de prueba la cantidad de código redundante seguía siendo muy elevada, y normalmente contenía prefijos comunes.

Por esta razón, Fraser y Wotawa [FW07d] proponen un monitor de las propiedades de los casos de prueba para la generación de los mismos. Este monitor funciona de la siguiente forma: cuando se genera un nuevo contraejemplo, las propiedades de los contraejemplos generados hasta este momento se analizan contra este contraejemplo. Si se encuentra una propiedad que ya ha sido probada el *model checker* no la probará, reduciendo así el tiempo de las pruebas. La técnica concreta para llevar a cabo la monitorización propuesta por Fraser y Wotawa, es realizada por Havelund y Rosu [HR01] en 2001. Esta técnica también reescribe parte de la formulación y utilizan una nueva técnica para verificar si el sistema contiene una determinada traza de ejecución, y no solo eso, lo realiza en tiempo real. El sistema de analizar las trazas de Havelund y Rosu estaba muy por encima en términos de velocidad al que tenían sus antecesores, siendo capaz de realizar 3 millones de reescrituras por segundo.

En el año 2007, también Fraser y Wotawa [FW07a] representan modelos capaces de mutarse con propiedades de la lógica temporal, lo que permite hacer acercamientos basados en mutaciones. Cada modelo mutado sólo posee una característica pero es comprobada para saber cuales son los efectos que pueden tener sobre la mutación que ha realizado el sistema. Si una propiedad ya es cubierta por un

contraejemplo, no será necesario incluir esta característica dentro de la generación de casos de prueba. Además propusieron y usaron una generación incremental de casos de prueba unidos al proceso de monitorización que ellos mismos habían creado. Aun así, esta técnica tiene un gran problema, la decisión del orden en el cual se tienen que analizar las propiedades tiene una gran influencia sobre el tamaño de los casos de prueba. Para intenta arreglar esto, Fraser y Wotawa utilizan el mismo sistema que ya utilizó Hamon [HDMR04], utilizar las propiedades de forma aleatoria, lo cual hacia los casos de prueba bastante más pequeños.

Como hemos comentado anteriormente, cuando se convierte un contraejemplo en un caso de prueba, lo más común es que este caso nuevo contenga redundancia. La monitorización intenta evitar esto, pero puede llegar a causar que se creen prefijos idénticos. Un intento de luchar contra esto es mapear cada contraejemplo. Esto lo que propuso Hamon et al. [HDMR04], y con ello, después de crear cada contraejemplo, el estado inicial de cada proceso de verificación va pasando al siguiente, de esta forma se intenta que no suceda este error.

Otra forma de intentar reducir la creación de los casos de prueba fue realizada por Hong y Ural [HU05]. Su solución se basaba en asumir que los estados tienen relaciones entre sí y, de este modo, una identidad asume a otra, con lo que asegurando la primera también se asegura la segunda. Con este método se reduce el tiempo necesario para generar casos de prueba ya que primero se calcula un conjunto mínimo de identidades generadoras, para después se expanden estas identidades para conseguir más casos de prueba. Después, mediante las formulas LTL se hace que unas identidades asuman otras.

En 2007, Zeng et al. [ZML07] junta todos los casos que ha creado un *model checker* en una estructura a que llaman *test tree*. Cada contraejemplo que ha creado con esta herramienta se introduce en esta estructura y se van eliminando los casos idénticos para evitar la duplicación. Una vez que el árbol esté completo, llenando todos los requerimientos que han dictado los analistas para las prueba, se crea un conjunto de pruebas con todos los los caminos posibles

dentro del árbol. Fraser y Wotawa [FW07d] también realizaron este proceso sobre un método concreto, lo que lo hacía más realizable en términos prácticos.

2.9.4 **Optimizaciones**

Como se ha comentado anteriormente, el rendimiento es uno de los principales problemas en el *model checking* y afecta a cada uno de los procesos que se llevan a cabo dentro de él. La generación de casos de prueba no iba a ser una excepción y en muchos casos la generación no es lo suficientemente rápida como para ser aplicada a modelos reales. Si existen demasiados casos de pruebas, o estos son extremadamente grandes la ejecución de los mismos puede no llegar a ser aplicable en la práctica.

Lo mejor para generar casos de prueba sería realizar las primeras pruebas con un conjunto de casos de prueba mínimos. Con lo que hay que tener en cuenta diversos factores para llevar esto a cabo, como el numero de estados del sistema o en el de transiciones entre los estados. Como sabemos el problema entre las relaciones de los estados y las correlaciones entre ellos tienen una complejidad NP-hard como demostró Hong et al. [HCL+03].

En la mayoría de los casos las suites de pruebas no son mínimas, y normalmente los casos dentro de ellas no contribuyen a la detección de errores en las mismas. Ejemplos de esta carencia de contribución a la detección de errores los podemos encontrar en que en muchos casos los prefijos de dos casos de prueba son iguales y esto hace que se repitan las pruebas son los mismos prefijos varias veces. Blank y Ranville [BR01] muestran en su trabajo varios métodos para la eliminación de casos de prueba que son innecesarios.

Otra técnica fue la propuesta por Black [BR01], llamada minimización. Esta técnica selecciona un conjunto de casos de prueba que entran dentro de una cobertura que puede ser limitada, con lo cual el número de prueba va a ser considerablemente más bajo. Esta técnica esta basada en otra de reducción de los casos de prueba que ya definió Harrold et al. [HGS93] en 1993.

Como hemos comentado antes la dificultad de encontrar el camino mínimo u óptimo es un problema NP-hard, y la para la solución de esto se han propuesto muchas heurísticas [HGS93, RHVRH02, ZZM06]. Uno de los principales puntos que se intenta aliviar es la reducción de casos de prueba y conseguir que una suite de pruebas solo tenga un caso de prueba por cada subconjunto del sistema. El problema que puede tener una reducción de este tipo es que es posible que se vea muy dañada la detección de errores, como ya han demostrado en diversas ocasiones [HG04, JH03, RHOH98, WHLM95]. Aunque también hay que añadir, que aun teniendo la suite completa de pruebas y no realizando ninguna reducción es posible que no se ejecuten todos los subconjuntos del sistema. Heimdahl y Devaraj [HG04] hicieron experimentos en el contexto de los *model checkers* basados en la generación de casos de pruebas. En estos experimentos también llegaron a la conclusión de que la reducción de la suite de pruebas reduce considerablemente el número de casos de pruebas, pero la detección de errores también sufre las consecuencias de esta reducción.

Los *model checkers* que existen en la actualidad también tiene un alto número de redundancia cuando intentan crear un suite de pruebas, como ya demostraron Fraser y Wotawa [FW07b]. Este hecho repercute en el tamaño, haciendo que sea muy grande, y encontraría lo mismo que una suite de pruebas más optimizada. Una solución que proponen los autores es separar los prefijos comunes y recombinarlos, con lo cual los repetidos acaban desapareciendo y el tamaño de la suite de pruebas ha sido reducido.

Otra técnica que mejora considerablemente la velocidad para encontrar los errores es la priorización de casos de pruebas, que fue propuesta por Rothermel et al. [RUCH99]. Con este método se intenta priorizar a los casos de prueba, poniendo un determinado orden para que consigan resultados de la forma más rápida posible. Fraser y Wotawa [FW07c] también hicieron un rápido acercamiento a este sistema consiguiendo resultados bastante buenos.

2.9.5 Pruebas basadas en la cobertura

Las pruebas de calidad basadas en la cobertura son aquellas que marcan unos límites de los cuales el análisis no debe de pasar. Con solo pararse a pensar lo que representa poner límites a la búsqueda dentro de unos patrones podemos pensar que se van a perder muchos potenciales errores en la generación del modelo, pero este es un hecho que sucede al igual que con cual otro método que intente optimizar el modelo basado en el sistema.

Con solo este hecho se van a perder muchos eventos que podrían desencadenar en los errores que se están buscando, y para este propósito Heindalm et al. [HRV+04] realizaron pruebas de escalabilidad sobre los generadores de casos de prueba de los *model checkers*. Llegaron a la conclusión de que al hacer casos de prueba muy pequeños, la detección de errores es muy escasa. Para intentar resolver este problema realizaron otro estudio [HGW04] en el cual intentaban saber cuales los criterios de búsqueda óptimos. Para realizar este experimento utilizaron casos de prueba generados aleatoriamente y otros creados mediante la cobertura. Heindalm y colaboradores consiguieron que el modelo generado aleatoriamente siempre devolviera el contraejemplo más corto, devolviendo mejores resultados que los modelos generados mediante límites.

En el año 2005 Devaraj et al. [DHL05] realizaron otros experimentos que demostraban que con un criterio de cobertura apropiado se pueden realizar análisis de forma bastante efectiva, mientras que para la generación de casos de prueba mostraban bastantes carencias. El problema es que la parte que quede dentro de los límites de la cobertura no es seguro que consigan ejecutar todas las posibles propiedades del modelo. Para intentar que este problema no impida generar casos de prueba correctos, proponen la creación de variables auxiliares que indiquen las partes que han sido ejecutadas por este criterio. El problema puede llegar a darse si la cantidad de variables auxiliares es muy grande el rendimiento del sistema se puede ver afectado en gran medida.

Abdurazik et al. [AADO00] realizaron otro estudio para intentar encontrar los mejores criterios para la especificación de la cobertura.

Los casos de pruebas en este caso fueron creados automáticamente usando tres criterios: cobertura total de los predicados, de transición de pares y de mutación de la especificación. La conclusión mostró que los resultados de los predicados y la cobertura de transición de pares son muy similares entre ellos y los resultados son mucho mejores que los del tercer modelo.

2.9.6 **Pruebas de regresión**

Las pruebas de regresión se utilizan para realizar comprobaciones sobre software que ha sido modificado y se quiere asegurar que no se ha introducido ningún error nuevo. En la mayoría de los casos cuando esto sucede se tendrían que volver a realizar todas las pruebas, pero esto llevaría una gran cantidad de tiempo y de dinero para las empresas que lo realizan. Por esta razón, se intenta seccionar solo los conjuntos de pruebas que sean necesarios para llevar a cabo las labores de búsqueda de errores en el código que ha cambiado o que se ha introducido nuevo.

Existen algunos métodos para realizar las técnicas de regresión más efectivas, en las cuales se intenta no repetir la comprobación de código ya revisado. Una de ellas fue propuesta por Xu et al. [XDR04], que creó un comparador especial sobre las propiedades de los dos modelos que se están probando, el original y el nuevo con el código cambiado. Las propiedades muestran los caminos que han sido modificados, y se le introducen variables especiales para los identificar estos caminos en los análisis. Estas propiedades se comprueban en el modelo anterior y solo las que son parte del contraejemplo son consideradas para realizar las pruebas de regresión. Las pruebas que coinciden con el modelo original no han sido modificadas, con lo cual no se deben realizar nuevas pruebas con ellos.

Un buen estudio sobre las diferentes técnicas que se han creado para realizar pruebas de regresión fue realizado por Fraser et al. [FAW07]. En este caso las pruebas fueron realizadas con la ayuda de un *model checker*, ejecutando los casos de prueba en el nuevo modelo y comprobando las salidas, confirmando que casos de prueba son todavía válidos y cuales ya no los son. Una vez ahí se extraen

las propiedades de los dos modelos y se comprueba cuales han sido cambiadas. Si se han producido cambios en el sistema de transición se verán reflejadas en alguna de las propiedades.

Cuando se ha determinado que casos de prueba son obsoletos, existen diferentes acercamientos para generar nuevos casos de prueba. Normalmente lo primero que se hace es comprobar cual ha sido el comportamiento que ha cambiado dentro del sistema de acuerdo con un estado, y así determinar que casos de prueba están obsoletos. Si esto no se puede realizar, se generarán propiedades en ambos modelos, y se comprobará la diferencia que existen entre los dos, generando nuevos casos de prueba. Fraser et al. [FAW07] propone reescribir las propiedades y el propio modelo, con esto se conseguirá realizar solo los casos de prueba dentro de los nuevos cambios en el modelo. Los casos de prueba que se crean con este método y que son validos en el modelo antiguo, servirán para crear una nueva suite de pruebas. Estos métodos consumen mucho tiempo, ya que la creación y generación de nuevos casos de prueba son tareas muy costosas.

2.9.7 **Visibilidad**

En los modelos que estamos examinado suelen estar basados en programas por ello son completamente dependiente de valores del sistema, y variables. Estas variables pueden llegar a ser internas con lo cual no son completamente observables directamente, o mediante entradas al modelo. En el *model checking* es importante poder comprobar que las variables cambian o que alguna salida se ha modificado.

Existen principalmente dos acercamientos para poder observar cambios aun cuando las variables sean internas. Uno de los primeros métodos fue propuesto por Hong et al. [HLSU02] el cual crea un nuevo predicado llamado *Exist* que es verdadero cuando se produce un cambio en cualquier variable interna. También utiliza un estado de entrada como de salida para intentar comprobar que los casos pueden ser ejecutados sin error.

La segunda solución fue propuesta por Okun et al. [OBY03] en 2003. En su estudio proponen dos acercamientos que terminan con cambios observables en al salida. Para ello crea un sistema de expansión que cambia las variables internas por todos los estados posibles que pueden llegar a tener.

2.9.8 **Determinismo**

Un gran problema de los contraejemplos es que solo pueden funcionar contra modelos deterministas. Por esta razón muchos *model checkers* tienen las capacidades para determinar estos modelos. Por ello cuando estos detectan un modelo no determinista suelen analizarlo decir que la el modelo es inconcluso.

Una solución a este problema fue presentada por Fraser et al. [FW07c] en 2007. Su técnica es la de introducir un indicador que muestra si una acción no determinista se ha ejecutado o no. Si durante la ejecución del modelo el *flag* se mantiene en un valor verdadero significa que algo ha salido mal durante la ejecución, y en ese momento mostrará el resultado como inconcluso, dando por hecho que se trata de una ejecución de un modelo no determinista o que se ha detectado un error. Este modelo también se puede extender a las estructuras en forma de árbol en los cuales las bifurcaciones de las ramas pueden seguir un modelo no determinista.

En el estudio realizado por Fraser et al. [FW07c] si se detecta un resultado inconcluso durante la ejecución de un caso de prueba, se utiliza el último caso determinista como estado inicial del sistema y se crea un nuevo contraejemplo. Este contraejemplo se mostrará como una nueva rama del caso de prueba anterior, con lo cual al final el resultado será de una estructura en forma de árbol.

Otra forma de resolver el determinismo fue propuesta por Boroday et al. [BPG07], que distingue entre casos fuertes y débiles. Un caso de prueba t, en un modelo S, una mutación M es débil si M puede producir una secuencia de salida que S no puede producir. Y un caso de prueba t es fuerte si para cualquier salida de M en respuesta a t, la salida en S es diferente. Bajo esta lógica, un caso

de prueba débil puede ejecutar información de errores si es repetidamente ejecutada, ya que resetea las transiciones continuamente.

Este acercamiento es, en cierta forma muy similar al propuesto por Fraser et al. [FW07c], en el cual podríamos decir que un programa es débil si posee una salida inconclusa.

2.10 Programación verificada

Realizar un software correcto y que ha sido verificado ha sido el objetivo de muchos estudios científicos en la informática moderna. Actualmente ya poseemos buenos conocimientos para poder describir que hacen los programas y como lo hacen. Gracias a esto ya es posible aplicar técnicas de verificación de programas en la fase de diseño de la aplicación.

Aunque actualmente esta técnica todavía se encuentra en las primeras fases de su vida, comparada con otros lenguajes de programación, ya es posible verificar programas de tamaño medio mediante las técnicas que nos dan los poderosos programas de verificación. Esto puede hacer que aplicaciones críticas puedan llegar a ser completamente seguros ya que estaría verificado todo su comportamiento en cualquier circunstancia. Con el avance de esta técnica es posible que en los próximos años podamos tener gran cantidad de software verificado y no solo pequeñas aplicaciones críticas.

La utilización de la programación verificada ya se está planteando en mucho aspectos críticos de lo cimientos de la programación, uno de los cuales es la creación de un compilador completamente verificado, el cual está siendo desarrollado actualmente por Leroy [Ler09a], [Ler09b]. La creación de un compilador verificado es uno de los grandes hitos por parte de la programación ya que no se podrán introducir datos erróneos en ningún momento de la compilación, con lo cual el binario siempre se generará sin ningún error. Solo habría fallos si el programación ha programado con técnicas incorrectas introduciendo errores en su código.

Para realizar este compilador Leroy utiliza Coq[13], un comprobador de teorías que es capaz de validar cualquier código mediante lógica matemática y la deducción lógica. Con ello se puede decir que la programación que surge de esta rigurosamente basada en fundamentos matemáticos. Otro aspecto que se puede destacar de coq, es que aunque su código fuente no produce un binario que pueda ser ejecutado en alguna plataforma, su código es fácilmente exportable a Ocaml[14] el cual sí que puede generar un código ejecutable.

Ocaml es un lenguaje de programación desarrollado por INRIA[15] (el instituto francés para la investigación en ciencias de la computación y automatización), que admite los paradigmas de programación imperativa, funcional y orientada a objetos. Realizando un código equiparable a C/C++ en eficiencia.

Además del compilador verificado también existe *model checker* verificado, que fue creado por Sprenger [Spr98] formalizando el cálculo μ para realizar pruebas. Esta herramienta no es tan completo como otros que existen en la actualidad, y continua manteniendo los problemas que tienen los modernos que son inherentes a cualquier *model checker*.

Con las ventajas que ofrecen Coq han sido muchos los investigadores que han utilizado este lenguaje para validar sus programas que utilizan lógica temporal. Un estudio completo y la implementación de CTL fue realizado por Paiva [PC98], que fue capaz de sobreponer algunos problemas que contenía la representación de la lógica bajo este lenguaje de programación. Unos años más tarde, Coupet-Grimal [CG03] mostraba como realizar la completa axiomatización de la lógica temporal utilizando los tipos y estructuras de Coq. El estudio de la complejidad y la completitud de CTL mediante Coq no fue realizado hasta el año 2014 por Doczkal et al. [DS14], demostrando que Coq sirve para desarrollar todo tipo de teorías además de para realizar código verificado.

Una de las lógicas temporales más difíciles de conseguir representar mediante Coq era CTL*, por la complejidad que tiene esta

[13] https://coq.inria.fr/
[14] https://ocaml.org/
[15] http://www.inria.fr/en/

al unir dos lógicas temporales en una sola. Pero Tsai et al. [TW07] consiguieron llevarla a cabo en el año 2007, aunque no fueron capaces de conseguir realizar la completa axiomatización de todos los componentes de esta lógica.

Bajo el enfoque de la verificación de software han sido varios los trabajos que se han realizado en este área. Uno de los más destacados ha sido el realizado por Leucker y Schallhart [LS09], realizando unas librerías para la verificación de software en tiempo de ejecución con técnicas conocidas para esta tarea, incluso llegando a crear un sistema de monitorización para programadores.

2.11 Conclusiones

Los avances en algoritmos en los últimos años han conseguido que los usuarios disfruten de un código mucho más seguro. Mediante esto, no solo se consigue que los programas sean más robustos sino que además el usuario esté menos expuesto a ataques y al malware, ya que estos dos problemas se aprovechan de los errores en el software para conseguir más privilegios.

Los algoritmos para el análisis estático y el model checking, los cuales utilizan la lógica temporal, han conseguido que se realicen programas para la comprobación de fallos de una manera muy eficaz. Éstos incluso han logrado que la labor de la búsqueda de errores se haya realizado de manera más sencilla de cara al usuario.

Un ejemplo del triunfo de estos algoritmos ha sido Value-Set Analysis, que está siendo utilizado en gran cantidad de aplicaciones para comprobar la correcta ejecución del programa analizado. Bitblaze o Bat son dos plataformas que han incluido dentro su estructura este análisis para llevar a cabo sus estudios sobre seguridad haciéndolos altamente efectivos.

La otra rama de los algoritmos de búsqueda de errores son las lógicas temporales y el model checking. Varias son las herramientas están utilizando esta aproximación para validar modelos, localizando de manera eficaz errores o trozos de código que desea el usuario.

Ambas aproximaciones tienen problemas por la propia natura-

leza de la técnica de análisis estático. El tiempo necesario para conocer todos los posibles estados que contiene un programa puede llegar a ser muy grande, haciendo que la técnica resulte poco efectiva en la práctica. Aunque varios equipos de científicos están trabajando en mejorar estos problemas todavía no existe una solución clara y válida para este problema.

Con este conocimiento hemos adoptado la técnica que trabajaba más cerca del código ensamblador para poder comprender los errores que puede ocasionar el compilador. A partir de ella, hemos adaptado un nuevo algoritmo para que devuelva las rutas que llegan a ejecutar ese código erróneo. Esto lo vamos a ver con más detalle en el siguiente capítulo.

«Si la gente no piensa que las matemáticas son simples, es solo porque no se dan cuenta de lo complicada que es la vida.»

John Von Neumann (1903–1957)

Capítulo

3

Algoritmo de búsqueda de errores

Como hemos observado en el capítulo anterior, la implementación de algoritmos que busquen errores en el código fuente ha avanzado mucho en los últimos años. Aunque esto no ha contribuido a que el problema de los errores en el software desaparezca, debido a que nuevos errores aparecen y los programadores siguen cometiendo errores. Nuevos problemas se siguen produciendo tanto por errores humanos como por imperfecciones en el software.

Debido a esta problemática surge la lógica computacional de predicados en árbol orientada a caminos, «*Path Oriented Computational Tree Predicate Logic*» (POCTPL), una nuevo sistema de verificación formal para la revelación de errores en el software compilado. Esta lógica nace para cumplir un propósito claro, conocer cuales son las rutas del programa que verifican la formula introducida por el usuario, en forma similar a como lo realizan los contraejemplos. Cuando Kinder et al. [KKSV05] crearon CTPL, llegar a saber si un determinado binario era *malware* o no era condición suficien-

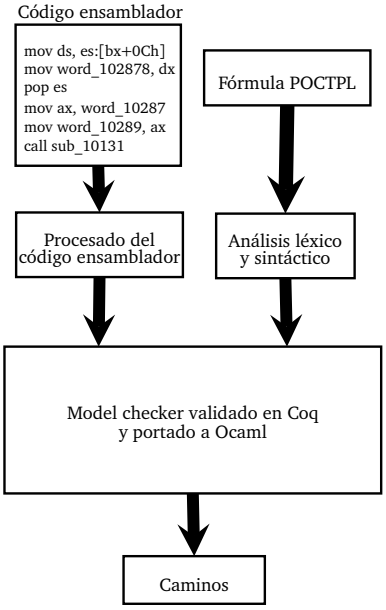

Figura 3.1: Estructura interna de la aplicación.

te para catalogarlo. Esto cambia en lo referente a la búsqueda de vulnerabilidades, donde conocer la localización de un error y donde se produce dentro del código es vital. Mediante esta técnica un programador puede reconocer fácilmente dónde se produce el error dentro de su código, haciendo que la tarea de corrección de errores se haga mucho más sencilla e intuitiva.

Realizar este tipo de control de errores no es posible mediante la lógica booleana tradicional, por esta razón hemos transformado CTPL para que pueda ser utilizado para la búsqueda de errores, de forma similar a como lo hacen los contraejemplos. Para ello, hemos transformado la lógica tradicional en teoría de conjuntos, con el fin de mostrar cada posible ruta en el binario como un conjunto. De esta forma, cada ruta que pueda tomar el binario será un subconjunto dentro del conjunto de todas las posibles rutas que puede tener un programa en tiempo de ejecución.

La Figura 3.1 muestra como se va a estructurar el proceso de la búsqueda de errores utilizando el *model checker* de POCTPL. Como

observamos en la figura, el primer paso es procesar el código ensamblador y adaptarlo para que sea posible manipular todos los datos que ofrece el ensamblador de las máquinas Intel x86. IDA Pro es uno de los mejores desensambladores que se encuentran actualmente en el mercado, debido a que realiza uno de los mejores análisis del código ensamblador. La forma que tiene esta aplicación de analizar el código la hace única en el mercado, consiguiendo descubrir toda la estructura interna del código, funciones, variables, llamadas a la API de windows, y demás información muy valiosa para el investigador. Una parte generalmente desconocida de IDA Pro es que puede ser manejado mediante línea de comandos, haciendo posible realizar procesos automatizados. Una de las opciones que posee este programa desde la línea de comandos es analizar el binario y crear un fichero con el código ensamblador ya procesado por él, representando toda la estructura del binario de forma que sea más sencilla la interpretación del programa. Este fichero ensamblador lo utilizará el siguiente módulo de esta plataforma.

En esta fase empieza el procesado de la información que ha devuelto IDA Pro. Para ello se realiza un parseado de toda la estructura del fichero ensamblador y esta información se pone en forma de una estructura Kripke. Una vez que suceda esto, se le introduce esta información al *model checker* para que una vez que reciba la fórmula que ha introducido el usuario empiece con el proceso de verificación del binario.

Cuando el usuario introduzca una fórmula esta es comprobada para que no se introduzcan fórmulas con una sintaxis incorrecta dentro del sistema. Cuando la fórmula ha sido introducida y comprobada, el sistema ejecutará el algoritmo para la búsqueda de caminos que cumplen la condición introducida por el usuario. Cuando el sistema termine de comprobar todos los posibles caminos dentro de la estructura Kripke devolverá los caminos que han satisfecho la fórmula, finalizando así el proceso de verificación.

3.1 POCTPL-IR

Aunque los lenguajes ensamblador, o también llamados nemotécnicos o nemónicos, no son complejos en lo relativo al propio lenguaje, la sintaxis necesaria para realizar programas es muy compleja. Existen dos tipos de lenguajes máquina dependiendo de la arquitectura del microprocesador CISC[1] y RISC[2] (este último se desarrolló posteriormente para facilitar la programación y depuración de programas en este lenguaje).

Aunque podría parecer que en principio la arquitectura que tendría que haber conseguido el mercado es la RISC, por la facilidad para programar cualquier programa con un número reducido de instrucciones, la tecnología que domina el mercado a día de hoy en todos los ordenadores personales y muchos servidores es la arquitectura CISC. El dominio de ésta es debido a que los microprocesadores que se han llevado el mercado actual en informática son los fabricados por las empresas Intel[3] y AMD[4], las cuales implementan microprocesadores basados en esta arquitectura.

Esta ha sido la razón principal por la cual hemos decidido realizar la investigación para esta tesis basándonos en el lenguaje ensamblador tipo CISC, más concretamente en el de Intel x86, ya que en la actualidad es uno de los más utilizados tanto para realizar aplicaciones como para la realización de los propios sistemas operativos con los que trabajamos todos los días.

A este lenguaje lo hemos llamada POCTPL-IR, o representación intermedia de POCTPL. Al utilizar este lenguaje nos hemos encontrado con varios problemas que hemos ido solventando a lo largo de la investigación de esta tesis. El primero es que el gran número de instrucciones que tiene este lenguaje iba a ralentizar mucho el proceso de la búsqueda del patrón que satisfaga la fórmula dada. Las comparaciones pueden llegar a ocupar mucho tiempo de CPU, y cuantos más códigos operacionales tenga que comprobar el progra-

[1] del inglés, *complex instruction set computer*
[2] del inglés, *reduced instruction set computer*
[3] http://www.intel.com
[4] http://www.amd.com

ma este se ejecutara más lento. En el caso de esta tesis estas comparaciones se harían por cada estado del sistema haciendo que fuera inviable el análisis de aplicaciones, incluso de las más pequeñas. Por esta razón hemos reducido el conjunto de instrucciones a examinar basándonos en la investigación que realizó Bilar [Bil07]. En este estudio demostró que el *malware* utiliza un conjunto reducido de códigos operaciones (también llamados *opcodes*), es suficiente para realizar casi todas las operaciones que realizan este tipo de binarios maliciosos. En nuestro caso es diferente, pero hemos visto que este conjunto de códigos operacionales es muy similar a los que usan los compiladores para compilar el código, con lo cual, es suficiente para reproducir patrones de comportamiento que puedan llevar a que se produzcan errores en el código.

Además de los códigos operacionales recomendados por Bilar se ha introducido otra instrucción, **Loc**. Esto no es realmente un código operacional sino que es una instrucción que viene adaptada desde IDA Pro, y se utiliza para hacer separaciones con los bloques en la memoria. Mediante esta instrucción se pueden conocer las regiones de la memoria del proceso en la cual se producen los saltos (sobre todo por los códigos operacionales de saltos condicionales) durante la ejecución del binario. Estas regiones serían la representación en texto de los bloques de ejecución que se pueden observar en la interfaz de IDA Pro en modo gráfico. En la Figura 3.2, podemos ver la representación gráfica de un bloque de memoria, esto en modo texto se encuentra dentro de una región que va representada por el código `loc` más un número hexadecimal.

Para la representación de nuestro lenguaje intermedio hemos decidido cambiar el número hexadecimal por uno decimal. Esta decisión se tomó debido a la facilidad que tiene Ocaml para manejar los números decimales con pocas instrucciones, el código sea más eficiente. En el caso de hacer la comprobación con números hexadecimales las operaciones para calcular se harían mucho más complejas haciendo que el tiempo para hacer cálculos se incrementase.

Otro problema que ha surgido es la complejidad de las operaciones que realizan los códigos operacionales en CISC. Para este tipo de arquitectura un mismo código operacional puede albergar

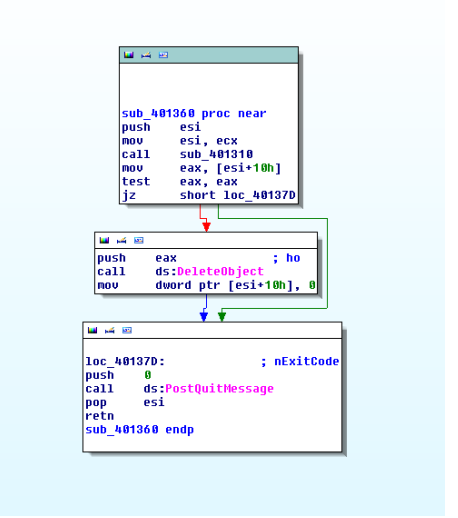

Figura 3.2: IDA Pro mostrando un salto que utiliza la instrucción **loc**.

operaciones con distinto tamaño de operandos. Por ejemplo, una operación de tipo `mov` puede hacer operaciones entre registros de 8, 16 o 32 bits. Aunque este tipo de operaciones están permitidas, este lenguaje no permite que, por ejemplo, una operación `mov` puede mover datos de un registros de 8 bits a uno de 32. El tamaño de los operandos siempre tiene que coincidir en todas las operaciones realizadas por los códigos operacionales de CISC.

Para solventar este inconveniente hemos creado un nuevos tipos de datos que tienen en cuenta el tipo de operación que va a realizar el código operacional. Los tipos de datos que hemos creado son los siguientes:

regAdd32 Instrucción que tiene en cuenta las operaciones que se producen entre un registro de 32 bits y una dirección de memoria. Un ejemplo de este tipo de instrucción puede ser la operación `mov edi, 0x0041E9CC`.

addAdd32 Esta instrucción contempla operaciones realizadas entre direcciones de memoria. La operación `mov 0x004C399C, 0x004C3988` puede ser un ejemplo bastante claro de este tipo de operaciones en el microprocesador.

addReg32 Instrucción que se utiliza cuando se realizan operaciones entre una dirección de memoria y un registro de 32 bits. Podemos observar un ejemplo de este tipo de instrucciones en `mov 0x00504C48, eax`.

regReg32 Esta instrucción se utiliza para las operaciones realizadas entre dos registros de 32 bits. Esta operación es común en instrucciones como `xor eax, eax`.

regAdd16 Instrucción para las operaciones que tengan un registro de 16 bits y una dirección de memoria del mismo tamaño. Un ejemplo de esta operación lo podemos observar en `cmp ax, 0x0042DAF6`.

addAdd16 Operación para realizar operaciones entre dos direcciones de memoria de 16 bits. La operación `test ax, ax`, es un claro ejemplo de este tipo de instrucciones.

addReg16 Esta instrucción se utiliza para las operaciones entre una dirección de memoria de 16 bits y un registro con idéntico tamaño. Un ejemplo de esta operación es la instrucción `xor 0x4522, ax`.

regReg16 Instrucción que se utiliza para las operaciones entre dos registros de 16 bits. Las instrucciones que utilizan este tipo de operaciones son similares a `add ax, bx`.

regAdd8 La instrucción que vemos aquí se utiliza para las operaciones entre un registro de 8 bits y una dirección de memoria del mismo tamaño. `and eax, 0x21AF` es un buen ejemplo de este tipo de operaciones.

addAdd8 Instrucción destinada a controlar las operaciones entre dos zonas de memoria de 16 bits. Esta operación es común encontrarla en operaciones como `test 0x00, 0x02`.

addReg8 Esta instrucción se utiliza para las operaciones entre una dirección de memoria de 8 bits y un registro del mismo tamaño. Este tipo de operaciones se usa en operaciones similares a `cmp 0xF2, al`.

regReg8 Instrucción que es utilizada para operaciones entre dos registros de 8 bits. La operación `xor ch, ch` es uno de los ejemplos de como se puede encontrar este tipo de operaciones.

regi32 Esta instrucción se utiliza para las operaciones que solo utilicen un registro de 32 bits. Esta operación es común en operaciones como esta `pop eax`.

addr32 La instrucción aquí mostrada sirve para las operaciones que solo utilizan una dirección de memoria de 32 bits. Instrucción que suele ser utilizada en instrucciones similares a `call 0x0023E300`.

regi16 Instrucción para las operaciones que solo utilizan un registro de 16 bits. Como ejemplo de este tipo de operaciones podemos ver `jmp ax`.

addr16 Las operaciones que solo utilizan una dirección de memoria de 16 bits deberán utilizar esta instrucción. Esta operación se utiliza en operaciones como `push 0x2BA0`.

regi8 Esta instrucción se utiliza para las operaciones que solo se ejecutan con un registro de 8 bits. Una de las operaciones más comunes que utilizan esto es `push ah`

addr8 Instrucción que se utiliza para las operaciones que solo utilizan una dirección de memoria. `pop 0x33` es un buen ejemplo de este tipo de instrucciones.

Como vemos tenemos todas las posibles operaciones que se pueden llevar a cabo entre direcciones de memoria y registros están reflejadas en nuestro sistema. por ello cada código operacional irá seguido de una de estas instrucciones para saber cual será el tamaño y el tipo de los argumentos que se le pasan al código operacional, y hacer que de esta forma no se produzcan operaciones ilegales en el sistema.

Otro cambio que era necesario para realizar una correcta verificación del lenguaje ensamblador era introducir procedimientos que

ayuden a representar las operaciones que se llevan a cabo con los registros de los microprocesadores. Aunque estos son de sobra conocidos hemos tenido que añadir unas instrucciones especiales para ciertas operaciones que normalmente ejecutan los programas.

Estas operaciones son utilizadas para controlar argumentos en la pila, ya sea de la propia función con la que se está trabajando o del puntero al cual está apuntando la pila en un momento dado. Las instrucciones que hemos introducido son las siguientes:

EBP_plus Para operaciones en las cuales se quieren haces operaciones con las partes más altas que el valor base de la pila. Normalmente esta operación se utiliza para conocer los argumentos que son pasados a una función.

EBP_minus Esta instrucción es para las operaciones que quieren conocer las partes más bajas que el valor base de la pila. Esto lo utilizan los compiladores para conocer cuales son las variables locales a una función.

ESP_plus Instrucción utilizada para las operaciones que son más bajas que el valor actual del puntero de pila. Se utiliza para conocer valores que acaban de ser introducidos en la pila para operaciones posteriores.

ESP_minus Esta instrucción se utiliza para conocer los valores más bajos que el valor actual del puntero de pila. Se utiliza para conocer valores que se han liberado de la pila.

Además de estos casos especiales existe un código operacional que llama directamente a funciones conocidas de la API de Windows o a las propias funciones definidas para el lenguaje C, pero aunque puede hacerlo a través de registros o de una dirección de memoria, estos dos casos no son relevantes para nuestro estudio.

Por esta razón tenemos una pequeña base de datos con llamadas relevantes y más peligrosas según Microsoft[5] dentro de su propia API de Windows y del propio lenguaje de programación C.

[5]https://msdn.microsoft.com/en-us/library/bb288454.aspx

La tabla 3.1 muestra algunos ejemplos de equivalencias entre código ensamblador normal y la representación intermedia de POCTPL. La mayor diferencia entre estas dos representaciones, es que en POCTPL tenemos que especificar el tamaño de las operaciones que va a realizar en código operacional, mientras que en el lenguaje ensamblador de Intel el tamaño está definido implícitamente.

Tabla 3.1: Equivalencia entre el código ensamblador de Intel y POCTPL-IR.

Código ensamblador Intel	POCTPL-IR
mov ah, 2	mov(regAdd8 ah 2)
xor eax, eax	xor(regreg32 EAX EAX)
push esi	push (reg32 ESI)
add ax, 0Ch	add (regAdd16 AX mem)
call strcmp	call strcmp

En el siguiente punto veremos como se estructura nuestra representación intermedia del lenguaje ensamblador dentro de una estructura Kripke. Pero antes de entrar en ese nivel de detalle necesitamos saber otros detalles que vamos a necesitar para conocer perfectamente como funciona nuestro algoritmo de búsqueda.

3.2 La estructura Kripke de POCTPL

Para la representación del programa, utilizamos una estructura Kripke, una idea más sencilla de un sistema de transiciones normal, y que ya hemos explicado en el capítulo anterior. Esta estructura nos permite representar el programa como si fuera un grafo, en el cual cada nodo se corresponde con una instrucción ensamblador y un conjunto de posibles caminos que el programa puede seguir, incluyendo los saltos condicionales en incondicionales.

Podríamos decir que, la estructura Kripke es una tupla donde:

- $S = \{s_i\}_{i=1}^{n} = \{s_1, s_2, \ldots, s_n\}$ es un conjunto de estados.

- $I \subseteq S$ es el conjunto de estados iniciales.
- $R \subseteq S \times S$ es un conjunto en el cual se muestran las transiciones entre estados, satisfaciendo $(s_i, s_j) \in R$ implica $i < j$, ya que queremos tener un estructura de grafo acíclica. R también satisface $\forall i = 1, \ldots, n-1$ existe $j = 1, \ldots, n$ tal que $(s_i, s_j) \in R$; por ejemplo, cuando solo existe un nodo final.
- $L : S \rightarrow 2^{AP}$, (donde AP son las proposiciones atómicas, del inglés *atomic propositions*) es una función que enlaza cada estado con un conjuntos de proposiciones, en nuestro caso, las instrucciones dentro del código ensamblador.

Nuestra estructura Kripke es finita, ya que S es finito. Por ello, asumimos que cada función del programa, tiene un principio (como por ejemplo, «`push bp`», «`mov bp, sp`», el cual suele ser el comienzo de todas las funciones) y un final (por ejemplo, «`retn`»).

En nuestra estructura Kripke, S sera una secuencia finita de estados s_1, s_2, s_3, denotados por una secuencia no repetida de enteros, donde $s_i \in S$, y $i \in \mathbb{N}$.

La I de la estructura Kripke contendrá todos los posibles puntos de entrada del programa (o código que IDA Pro no encuentra como acceder exactamente hasta ese punto). Esto puede hacer que en muchos casos se estudien puntos muertos del programa o partes que un investigador tendrá que comprobar para saber como un programa puede acceder hasta ese punto del programa. Esto es debido a que existen muchas operaciones dentro del programa que son calculadas en tiempo de ejecución y no es posible saber cual va a ser su valor mediante un análisis estático. Por tanto asumimos que el programa puede tener varios puntos iniciales y varios puntos finales (debido a los diferentes caminos que puede soportar el programa). Además también asumimos que todos los nodos, excepto el nodo final de cada camino, tienen al menos un nodo sucesor.

R es un conjunto de pares en S. Este conjunto de pares representa los arcos de conexión entre dos estados. De esta forma, si el nodo s_1 está conectado con los nodos s_2 y s_3, tendremos los arcos (s_1,s_2) y (s_1,s_3) en R.

Finalmente L es la función que representa las relaciones entre los estados y los elementos del código ensamblador que queremos analizar, por ejemplo: (s_1, {`push (regi16 bp)`}), (s_2, {`mov (regReg16 bp sp)`}).

Un ejemplo de una estructura Kripke en nuestro sistema se muestra en la Figura 3.3. La representación de esta estructura es:

- $\mathscr{S} = \{s_1,s_2,s_3,s_4\}$
- $\mathscr{R}: = \{(s_1,s_2),(s_1,s_3),(s_2,s_4),(s_3,s_2)\}$
- $\mathscr{L}: = \{(s_1, \{\texttt{jne 0}\}), (s_2, (\{\texttt{mov eax, 3}\})), (s_3, (\{\texttt{mov eax, 5}\})), (s_4, \{\texttt{retn}\})\}$

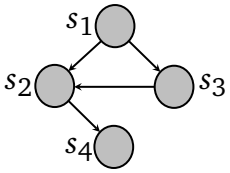

Figura 3.3: Sistema de transición para un programa simple.

La Figura 3.3 muestra la representación gráfica de una estructura Kripke mostrando dos caminos diferentes: (i) s_1, s_2, s_4 y (ii) s_1, s_3, s_2, s_4. Aunque esta estructura Kripke es muy sencilla sirve para ilustrar de forma muy simple como serán el resto de las estructuras que se crearán mediante nuestra plataforma.

3.3 Definición del lenguaje

La semántica de POCTPL es similar a CTL y, por consiguiente, a CTPL. La principal diferencia es que tanto CTL, CTPL como SCTPL son booleanas, mientras que nuestra lógica, POCTPL, se basa en álgebra de conjuntos. Por esta razón, nuestro lenguaje devuelve un conjunto de caminos que satisfacen la fórmula dada en vez de devolver un simple valor verdadero o falso.

3.3.1 Sintaxis

En CTL existen estados y caminos. En nuestro caso, los estados corresponden a códigos operacionales del lenguaje ensamblador con sus correspondientes parámetros. Un camino es un conjunto ordenado de estados, lo que significa que un camino debe de contener códigos operacionales consecutivos que son parte del programa. Un camino puede representar un flujo completo de ejecución desde el estado inicial del programa hasta un estado que no tenga ningún sucesor (ya sea porque ha llegado al final del programa o que IDA Pro no ha sabido resolver correctamente cual es la siguiente instrucción). Por lo tanto, todos los estados que conforman un camino deben ir de un menor a un mayor valor de estado. Finalmente, conviene recordar que todos los caminos en POCTPL están representados por la estructura Kripke.

Por lo tanto, las reglas de POCTPL son:

1. p es una fórmula de estado.
2. Si ϕ es una fórmula de estado o de camino, entonces $\neg\phi$ es una fórmula de estado o de camino.
3. Si ϕ y ψ son fórmulas de estado o de camino simultáneamente, entonces $\phi \cap \psi$ es una fórmula de estado o de camino.
4. Si φ es una fórmula de estado o de camino, entonces $\exists \bigcirc \varphi$ es una fórmula de camino.
5. Si φ y ω son fórmulas de estado o de camino simultáneamente, $\exists(\varphi\ U\ \omega)$ es una fórmula de camino.
6. Todo lo demás no es ni una fórmula de estado ni una fórmula de camino.

De la misma forma que sucede en CTL, las fórmulas de estado muestran las propiedades de los estados, mientras que las fórmulas de camino muestran las propiedades de un camino. POCTPL define las mismas operaciones que CTL: $\exists\bigcirc$, $\forall\bigcirc$, $\exists(\varphi\ U\ \omega)$, $\forall(\varphi\ U\ \omega)$,

$\exists\Box$, $\forall\Box$, $\exists\Diamond$, y $\forall\Diamond$. El significado de estas operaciones son equivalentes a sus análogos en CTL, pero basadas en teoría de conjuntos. Estas operaciones pueden utilizarse para conjuntos de estados o de caminos, pero en ambos casos devolverán un conjunto de caminos. Además incluimos una nueva operación ($\triangle$), y también definimos las operaciones $\exists(\varphi \triangle \omega)$ y $\forall(\varphi \triangle \omega)$. La operación $\triangle$ devuelve un conjunto de estados o caminos si el conjunto o estado φ es seguido por cualquiera de los conjuntos o estados en ω. Con CTPL y las operaciones tradicionales, resultaba imposible saber si dos estados o caminos eran adyacentes uno al otro.

El operando $\triangle$ es necesario debido a la naturaleza de las operaciones en POCTPL. Cada operación devuelve todos los caminos que satisfacen la fórmula dada, no porciones de estos caminos. De esta forma, no es posible definir fórmulas que comprueben la adyacencia de dos caminos o dos estados con las operaciones definidas en las lógicas anteriores. Como consideramos importantes la verificación de esta condición (por ejemplo, una instrucción `pop` después de una `call`) definimos la operación $\triangle$ representada en la Figura 3.4.

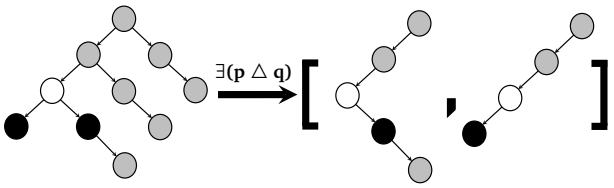

Figura 3.4: La fórmula $\triangle$ representa dos estados adyacentes, donde el estado blanco satisface **p**, el estado negro satisface **q** y el estado gris no satisface ni **p** ni **q**.

Para asegurarse de que las operaciones entre caminos (conjunto ordenado de estados) que devuelven un solo estado son posibles, el resultado de esta operación es siempre un camino (en otras palabras, un conjunto con un solo estado).

La sintaxis de POCTPL es definida desde las ecuaciones 3.1 hasta 3.9:

$$< register8 >::= \textbf{AH} \mid \textbf{AL} \mid \textbf{BH} \mid \textbf{BL} \mid \textbf{CH} \mid \textbf{CL} \mid \textbf{DH} \mid \textbf{DL} \mid \textbf{Register8}. \tag{3.1}$$

$$< register16 >::= \textbf{Ax} \mid \textbf{Bx} \mid \textbf{Cx} \mid \textbf{Dx} \mid \textbf{SI} \mid \textbf{DI} \mid \textbf{SP} \mid \textbf{BP} \mid \textbf{IP} \mid \textbf{SP\_plus} \mid \textbf{SP\_minus} \mid \textbf{BP\_plus} \mid \textbf{BP\_minus} \mid \textbf{Register16}. \tag{3.2}$$

$$< register32 >::= \textbf{EAX} \mid \textbf{EBX} \mid \textbf{ECX} \mid \textbf{EDX} \mid \textbf{ESI} \mid \textbf{EDI} \mid \textbf{EBP} \mid \textbf{ESP} \mid \textbf{EIP} \mid \textbf{CS} \mid \textbf{DS} \mid \textbf{SS} \mid \textbf{ES} \mid \textbf{FS} \mid \textbf{GS} \mid \textbf{EBP\_plus} \mid \textbf{EBP\_minus} \mid \textbf{ESP\_plus} \mid \textbf{ESP\_minus} \mid \textbf{Register32}. \tag{3.3}$$

$$< address >::= mem \tag{3.4}$$

$$< holdOne >::= \mid \textbf{regi32(}<register32>\textbf{)} \mid \textbf{addr32(}<address>\textbf{)} \mid \textbf{regi16(}<register16>\textbf{)} \mid \textbf{addr16(}<address>\textbf{)} \mid \textbf{regi8(}<register8>\textbf{)} \mid \textbf{addr8(}<address>\textbf{)} \mid \textbf{val(}<nat>\textbf{)} \tag{3.5}$$

$$\begin{aligned} < holdTwo >::= & \\ & \mid \textbf{regAdd32(}<register32>, <address>\textbf{)} \\ & \mid \textbf{addAdd32(}<address>, <address>\textbf{)} \\ & \mid \textbf{addReg32(}<address>, <register32>\textbf{)} \\ & \mid \textbf{regReg32(}<register32>, <register32>\textbf{)} \\ & \mid \textbf{regVal32(}<register32>, <nat>\textbf{)} \\ & \mid \textbf{addVal32(}<address>, <nat>\textbf{)} \\ & \mid \textbf{regAdd16(}<register16>, <address>\textbf{)} \\ & \mid \textbf{addAdd16(}<address>, <address>\textbf{)} \\ & \mid \textbf{addReg16(}<address>, <register16>\textbf{)} \\ & \mid \textbf{regReg16(}<register16>, <register16>\textbf{)} \\ & \mid \textbf{regVal16(}<register16>, <nat>\textbf{)} \\ & \mid \textbf{regAdd8(}<register8>, <address>\textbf{)} \\ & \mid \textbf{addAdd(}<address>, <address>\textbf{)} \\ & \mid \textbf{addReg8(}<address>, <register8>\textbf{)} \\ & \mid \textbf{regReg8(}<register8>, <register8>\textbf{)} \\ & \mid \textbf{regVal8(}<register8>, <register8>\textbf{)} \\ & \mid \textbf{addVal(}<add>, <nat>\textbf{)} \end{aligned} \tag{3.6}$$

$$
\begin{aligned}
&< opcode >::= \\
&\textbf{mov(}<holdTwo>\textbf{)} \mid \textbf{push(}<holdOne>\textbf{)} \\
&\mid \textbf{call(}<holdOne>\textbf{)} \mid \textbf{pop(}<holdOne>\textbf{)} \\
&\mid \textbf{cmp(}<holdTwo>\textbf{)} \mid \textbf{jz(}<holdOne>\textbf{)} \\
&\mid \textbf{lea(}<holdTwo>\textbf{)} \mid \textbf{test(}<holdTwo>\textbf{)} \\
&\mid \textbf{jmp(}<holdOne>\textbf{)} \mid \textbf{add(}<holdTwo>\textbf{)} \\
&\mid \textbf{jnz(}<holdOne>\textbf{)} \mid \textbf{xor(}<holdTwo>\textbf{)} \\
&\mid \textbf{and(}<holdTwo>\textbf{)} \mid \textbf{retn} \mid \textbf{loc}\ integer
\end{aligned}
\tag{3.7}
$$

$$
\begin{aligned}
&< state >::= \textbf{TRUE} \mid \textbf{op}\ < opcode > \\
&\mid < state > \cap < state > \mid \neg < state >
\end{aligned}
\tag{3.8}
$$

$$
\begin{aligned}
&< path >::= \textbf{id}\ < state > \mid < path > \cap < path > \\
&\mid \neg < path > \mid \exists\bigcirc < path > \mid \exists(< path >\ \mathcal{U}\ < path >) \\
&\mid \exists\Box < path > \mid \exists(< path > \triangle < path >) \\
&\mid \exists\bigcirc < state > \mid \exists(< state >\ \mathcal{U}\ < state >) \\
&\mid \exists\Box < state > \mid \exists(< state > \triangle < state >)
\end{aligned}
\tag{3.9}
$$

Como podemos ver en las ecuaciones 3.1, 3.2 y 3.3, POCTPL ha sido diseñado para analizar binarios que han sido desensamblados con la notación Intel x86. Esta decisión en el diseño de POCTPL es debido a la salida que nos devuelve IDA Pro por defecto después de analizar un fichero.

Las ecuaciones 3.1, 3.2 y 3.3, definen los diferentes registros del procesador de 8, 16 y 32 bits respectivamente. También hemos añadido las instrucciones que hemos comentado anteriormente (ESP_minus, ESP_plus, EBP_minus, EBP_plus) para poder analizar operaciones más complejas.

La ecuación 3.4 muestra un tipo que indica una posición en la memoria. Como las posiciones de memoria pueden llegar a cambiar por diversos factores (dependiente de la máquina en la que se ejecute, de las protecciones como ASLR, etc.), hemos decidido no representar la dirección de memoria concreta a la que iría a acceder la CPU. Mediante esta abstracción se facilita mucho la búsqueda de patrones en el momento de realizar la fórmula para encontrar un determinado patrón.

En las ecuaciones 3.5 y 3.6, podemos empezar a ver la primera sintaxis original de POCTPL. <*HoldOne*> y <*HoldTwo*> representan el numero de argumentos de cualquier código operacional. De esta forma, en la ecuación 3.5 definimos dos tipos:

- Operaciones que se realizan con un solo registro, definidas como `regi`.
- Operaciones que se realizan con una dirección de memoria. Definidas como `addr`.

De igual forma, en la ecuación 3.6, podemos ver que existen cuatro tipos de operaciones:

- Operaciones entre dos registros. Definidas como `regReg`.
- Operaciones entre dos direcciones de memoria, definidas como `addAdd`.
- Operaciones entre un registro y una dirección de memoria, definidas como `regAdd`.
- Operaciones entre una dirección de memoria y un registro, definidas como `addReg`.

Además hay otro detalle que hemos considerado a la hora de crear la representación intermedia: el tamaño de la información que pueden llegar a manejar los códigos operacionales. En la mayoría de las arquitecturas este tamaño varía de 8 a 32 bits. Por esta razón, hemos añadido los sufijos 32, 16 y 8 a <*HoldOne*> y <*HoldTwo*>. Como sucede en el lenguaje ensamblador de Intel, POCTPL-IR solo permite operaciones entre registros con el mismo tamaño.

En la ecuación 3.7, definimos los códigos operacionales del lenguaje ensamblador. En nuestro caso utilizamos los que Bilar define como los más usados en el *malware* [Bil06]. Aunque este tipo de análisis queda fuera de los límites de esta investigación, el cual está orientado a la búsqueda de vulnerabilidades, hemos considerado estos códigos operacionales como la base para analizar el código

ensamblador con POCTPL. Después de verificar diversos errores en aplicaciones hemos observado que estos 14 códigos operacionales son suficiente para definir formulas que sirvan para encontrar vulnerabilidades en el software compilado.

En la ecuación 3.8, podemos observar las diferentes operaciones con conjuntos de estados.

Finalmente, la ecuación 3.9 define las operaciones que pueden ser aplicadas a conjuntos de caminos (operaciones de álgebra de conjuntos, de lógica temporal CTL y la nueva operación $\triangle$). Además, éstas pueden tener conjuntos de estados de caminos como operandos, aunque el resultado siempre será un conjunto de estados.

Como podemos observar, en POCTPL también tenemos la posibilidad de introducir tanto caminos como estados. De esta forma, es más sencillo encontrar cadenas dentro del código fuente si conocemos una secuencia de códigos operacionales. A su vez, también hace más sencilla la composición de fórmulas POCTPL.

La Tabla 3.2 y la Figura 3.4 definen las operaciones básicas con estados y caminos en detalle. Además, las Tablas 3.3 y la Tabla 3.5 definen las operaciones derivadas por medio de los axiomas de álgebra de conjuntos.

Las fórmulas en las Tablas 3.4 y 3.5 operan con estados y siempre devuelven caminos, ya que es necesario para convertir un conjunto de estados a caminos. Los que son usados como entrada para las fórmulas de estado serán devueltos como un camino en el cual los estados están ordenados acorde al orden de los estados en la estructura Kripke.

Tabla 3.2: Operaciones básicas definidas en la ecuación 3.8 donde, φ es un código operacional y Φ y Ψ son estados o un conjunto de estados.

Operación		Definición
TRUE	$\Leftrightarrow$	Esta operación devuelve el conjunto entero de estados que se encuentran en la estructura Kripke.

Continua en la página siguiente

Operación		Definición
op φ	$\Leftrightarrow$	Esta operación devuelve el conjunto que comprende solo los estados que contienen el código operacional especificado, si está presente dentro de la estructura Kripke. Si no lo está devuelve conjunto vacío. Definimos esta operación para introducir estados en fórmula dado su correspondiente código operacional.
$\Phi \cap \Psi$	$\Leftrightarrow$	Esta fórmula realiza la operación de intersección entre dos conjuntos de estados. En otras palabras, devuelve un conjunto de estados que está presente en los dos conjuntos.
$\neg\Phi$	$\Leftrightarrow$	Esta operación devuelve los estados que no están presentes en el conjunto de estados especificados por Φ.

Tabla 3.3: Operaciones entre estados derivadas a través de los axiomas del álgebra de conjuntos, donde Φ y Ψ son conjuntos de estados.

Operación		Definición
$\Phi \cup \Psi$	$\Leftrightarrow$	Esta operación es la de unión entre conjuntos. Ésta se consigue mediante la equivalencia entre la unión y la intersección: $\neg((\neg\Phi) \cap (\neg\Psi))$. El resultado será un conjunto de estados conteniendo todos los estados presentes en cada uno de los conjuntos especificados por los operandos.
$\Phi \rightarrow \Psi$	$\Leftrightarrow$	Esta operación representa la implicación entre conjuntos. En este caso, la derivamos siguiendo la siguiente equivalencia: $((\neg\Phi) \cup \Psi)$

Continua en la página siguiente

Operación		Definición
$\Phi \longleftrightarrow \Psi$	$\Leftrightarrow$	Este procedimiento representa la operación de co-implicación entre conjuntos. Como en los anteriores casos esta operación se deriva siguiendo la siguiente equivalencia: $(\neg\Phi) \cup \Psi) \cap ((\neg\Psi) \cup \Phi)$.

Tabla 3.4: Operaciones básicas sobre caminos donde σ es una fórmula de estado, Φ y Ψ son fórmulas de camino, φ y ω denota un operando que puede ser ambos un camino (conjunto ordenado de estados) o un conjunto de caminos.

Operación		Definición
id σ	$\Leftrightarrow$	La operación **id** devuelve un conjunto de caminos que contendrán aquellos representados por el conjunto de estados en σ, si esta presente en la estructura Kripke. Si solo existiese un camino, la salida sería un conjunto de caminos que contendría un solo camino. Esta operación es útil para introducir un camino, en una fórmula de caminos que requiere un conjunto de estos como operando.
$\Phi \cap \Psi$	$\Leftrightarrow$	Esta operación devuelve un conjunto de caminos conteniendo los que están presentes en ambos conjuntos especificados como operandos. Los dos caminos deben de ser iguales para que sean considerados equivalentes, conteniendo la misma secuencia ordenada y consecutiva de estados.

Continua en la página siguiente

Operación		Definición
$\neg\Phi$	$\Leftrightarrow$	Esta operación devuelve un conjunto de caminos conteniendo todos los presentes en la estructura Kripke y no que están en el conjunto de caminos especificados.
$\exists \bigcirc \varphi$	$\Leftrightarrow$	Esta operación devuelve un conjunto de caminos en la estructura Kripke que contienen, partiendo de la segunda posición, un conjunto de estados equivalentes en el camino (conjunto ordenado de estados) especificado como argumento.
$\exists(\varphi\ \mathcal{U}\ \omega)$	$\Leftrightarrow$	Esta operación devuelve el conjunto de caminos que existe en la estructura Kripke que en algún instante del camino contiene φ y, en cualquier momento, contiene un camino que empieza en ω.
$\exists\Box\ \varphi$	$\Leftrightarrow$	Esta operación devuelve el conjunto de caminos en la estructura Kripke que, en cualquier camino p especificado en φ, contiene repetición de la secuencia de estado en p.
$\exists(\varphi\ \triangle\ \omega)$	$\Leftrightarrow$	Devuelve el conjunto de caminos en la estructura Kripke que, en cualquier momento, contiene alguno de los caminos especificados en φ inmediatamente seguida por cualquiera de los caminos especificados en ω.

Tabla 3.5: Operaciones con caminos derivadas de los axiomas de álgebra de conjuntos. φ y ω denotan operandos que pueden ser ambos un conjunto ordenado de estados o uno de caminos.

Operación		Definición
$\exists\Diamond\varphi$	$\Leftrightarrow$	Esta operación devuelve un conjunto de caminos en la estructura Kripke que, en cualquier punto, contiene un conjunto ordenado de estados de cualquier camino en φ. Esta operación es obtenida desde la siguiente derivación: $\exists((TRUE)\ \mathcal{U}\ \varphi)$
$\forall \bigcirc \varphi$	$\Leftrightarrow$	Esta operación es obtenida por medio de la derivación: $\neg\exists \bigcirc \neg\varphi$. Esto devuelve los caminos en la estructura Kripke que en la segunda posición contienen estados o caminos devueltos por la operación φ.
$\forall(\varphi\ \mathcal{U}\ \omega)$	$\Leftrightarrow$	Esta operación se consigue mediante a los axiomas de la lógica clásica a través de la siguiente derivación: $\neg\exists(\neg\omega\ \mathcal{U}(\neg\varphi \cap \neg\omega)) \cap \neg\exists\Box\neg\omega$. Devuelve los caminos que empiezan por empiezan por φ y en cualquier punto satisfacen ω.
$\forall\Box\varphi$	$\Leftrightarrow$	Esta operación es análoga a $\exists\Box\varphi$, y se obtiene mediante la siguiente derivación: $\neg\exists\Diamond\neg\varphi$. Devuelve los caminos en la estructura Kripke que tienen en cualquier posición los estados o los caminos devueltos por la operación φ.

Continua en la página siguiente

Operación		Definición
$\forall\Diamond\varphi$	$\Leftrightarrow$	Esta operación es obtenida mediante la siguiente derivación: $\neg\exists\Box\neg\varphi$. Esta operación devuelve el conjunto de caminos en la estructura Kripke que, en cualquier punto, contiene los estados o caminos devueltos por φ.
$\forall(\varphi \triangle \omega)$	$\Leftrightarrow$	Esta operación es análoga a $\exists(\varphi \triangle \omega)$, y se obtiene a través de la siguiente derivación: $\neg\exists(\neg\omega \triangle (\neg\varphi \cap \neg\omega)) \cap \neg\exists\Box\neg\omega$. Esta operación devuelve los caminos que en cualquier punto tienen los caminos o estados devueltos por φ y en el siguiente estado o camino coincide con ω.

3.3.2 Semántica operacional

Para cualquiera de las fórmulas de estado, $\vDash$ es la relación entre una estructura Kripke $\mathscr{M}$, un estado $\mathtt{s}$, y para la fórmula de estado Φ: $\mathscr{M}, \mathrm{s} \vDash \Phi$.

Para cualquier fórmula de camino $\vDash$ es una relación entre un camino σ y una fórmula de este tipo φ: $\mathscr{M}, \sigma \vDash \Phi$. En nuestro caso omitiremos $\mathscr{M}$ por claridad.

Sea $\mathrm{p} \in AP$ una proposición atómica, $\mathscr{M} = (S, I, R, L)$ es una estructura Kripke, $\mathtt{s} \in S$ es un estado, ρ un camino, $\mathtt{paths}(\mathscr{M})$ denota el conjunto de caminos en la estructura Kripke $\mathscr{M}$, Φ y Ψ son fórmulas de estado, φ y ω son fórmulas de camino y $\mathtt{size}(\psi)$ es el tamaño del camino ψ. La relación de satisfacción $\vDash$ es definida como vemos en la Figura 3.5.

Como sucede en la lógica de primer orden, si φ es una fórmula con una variable libre x, entonces $\forall x\varphi$ y $\exists x\varphi$ son fórmulas POCTPL. Esta característica ya la implementado Kinder cuando desarrollo CTPL, pero cuando transformamos estas operaciones a POCTPL, $\forall$ y $\exists$ son equivalentes. Por lo tanto, no es necesario de-

$s \vDash p$	**iff**	$p \in \mathscr{L}(s)$
$s \vDash \neg\Phi$	**iff**	$\neg(s \vDash \Phi)$
$s \vDash \Phi \cap \Psi$	**iff**	$(s \vDash \Phi) \cap (s \vDash \Psi)$
$\rho \vDash \exists\bigcirc \Psi$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\sigma[1] \vDash \Psi$
$\rho \vDash \exists(\Phi\ \mathscr{U}\ \Psi)$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\exists j{\geq}0.(\sigma[j] \vDash \Psi \cup$ $(\forall(0{\leq}k{\leq}j).\sigma[k] \vDash \Phi))$
$\rho \vDash \exists\Box \Phi$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\forall j{\geq}0.\sigma[j] \vDash \Psi$
$\rho \vDash \exists(\Phi \triangle \Psi)$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\exists j{\geq}0.(\sigma[j] \vDash \Phi \cup$ $(\sigma[j{+}1] \vDash \Psi))$
$\rho \vDash \exists\bigcirc \varphi$	**iff**	$\exists\sigma \in paths(\mathscr{M}).$ $\forall(0{\leq}j{\leq}size(\varphi)).\sigma[1{+}j] \vDash \varphi[j]$
$\rho \vDash \exists(\varphi\ \mathscr{U}\ \omega)$	**iff**	$\exists\sigma \in paths(\mathscr{M}).$ $\forall(0{\leq}j{\leq}size(\varphi)).(\sigma[j] \vDash \varphi[j]) \cup$ $(\exists(k{\geq}j).(\forall(0{\leq}m{\leq}size(\omega)).$ $\sigma[j{+}k{+}m] \vDash \omega[m]))$
$\rho \vDash \exists\Box \varphi$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\forall j{\geq}0.$ $\sigma[j] \vDash \varphi[j \bmod size(\varphi)]$
$\rho \vDash \exists(\varphi \triangle \omega)$	**iff**	$\exists\sigma \in paths(\mathscr{M}).\exists j{\geq}0.$ $\forall(0{\leq}k{\leq}size(\varphi).(\sigma[j{+}k] \vDash \varphi[k] \cup$ $\exists(0{<}m{\leq}size(\omega)).$ $(\sigma[1{+}j{+}k{+}m] \vDash \omega[m]))$

Figura 3.5: Semántica de POTCPL.

finir la relación de satisfacción para el operador $\forall$, porque ya está definido por el operador $\exists$.

Un ejemplo de la operación $\exists$ está detallado en la ecuación 3.10. Esta fórmula devolverá los caminos que empiecen con la operación `add` sobre un registro de 16 bits que está seguido en algún momento por la operación `push` sobre el mismo registro.

$$\exists x \in register16.\exists((add(regAdd16\ x\ 3)) \mathscr{U}\ (push(regi16\ x))) \tag{3.10}$$

Esta es la razón por la cual las ecuaciones 3.1, 3.2 y 3.3 contie-

nen unos tipos que no están en la notación original de Intel. Estos tipos tratarán siempre como verdadero cualquiera de los tipos de la misma estructura.

3.3.3 Axiomas

En las secciones previas hemos introducido algunas equivalencias para obtener las operaciones $\exists\Diamond$, $\forall\Diamond$, la operación de unión ($\cup$), la de implicación ($\rightarrow$) y la de co-implicación ($\leftrightarrow$). Estas equivalencias derivan de las fórmulas ya conocidas de la álgebra de conjuntos. Con lo cual, POCTPL debe satisfacer los siguientes axiomas para ser matemáticamente correcto.

En la Figura 3.6 se muestran los axiomas del álgebra de conjuntos que POCTPL debe satisfacer.

Tabla 3.6: Axiomas del álgebra de conjuntos del que derivan las operaciones. Donde A, B y C son conjuntos.

Axioma	Definición
$A \cup (B \cup C) \equiv (A \cup B) \cup C$ $A \cap (B \cap C) \equiv (A \cap B) \cap C$	Propiedad Asociativa
$A \cup (B \cap C) \equiv (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) \equiv (A \cap B) \cup (A \cap C)$	Propiedad distributida
$\neg(A \cup B) \equiv \neg A \cap \neg B$ $\neg(A \cap B) \equiv \neg A \cup \neg B$	Leyes de Morgan
$A \cup B \equiv B \cup A$ $A \cap B \equiv B \cap A$	Propiedad conmutativa
$A \cap (A \cup B) = A$ $A \cup (A \cap B) = A$	Absorción

Continua en la página siguiente

Axioma	Definición
$A \cup A = A$ $A \cap A = A$	Idempotencia
$A \cup \varnothing = A$ $A \cap \varnothing = \varnothing$	Conjunto vacio

3.3.4 **Representación del código ensamblador**

En la Figura 3.6 podemos observar la equivalencia entre el código ensamblador de Intel x86 y el POCTPL-IR. Como podemos ver, en la columna del ensamblador x86 aparecen códigos que no existen en la otra columna, esto es debido a que no son parte de los códigos operacionales más utilizados según Bilar [Bil07], como hemos comentado anteriormente. En la columna derecha, vemos el código transformado a la representación intermedia. Los saltos en el código son representados gráficamente.

La estructura Kripke $\mathscr{M}$ para este ejemplo es:

- $\mathscr{S} := (s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10})$.
- $\mathscr{R} := (s_1, s_2), (s_2, s_3), (s_2, s_4)\ (s_3, s_4), (s_3, s_{13}), (s_4, s_5), (s_5, s_6), (s_6, s_7), (s_6, s_{12}), (s_6, s_{13}), (s_7, s_8), (s_8, s_9), (s_8, s_{12}), (s_9, s_{10}), (s_{10}, s_{11}), (s_{11}, s_{12}), (s_{11}, s_{12}).\ (s_{12}, s_{13}), (s_{13}, s_{14}), (s_{14}, s_{15})$.
- $\mathscr{L} :=$ (s_1, {`loc 1`), (s_2, {`cmp` (regAdd8 AL 61)}), (s_3, {`cmp` (regAdd8 AL 32)}), (s_4, {`loc 2`), (s_5, {`dec` (regi16 CX)}), (s_6, {`cmp` (regAdd8 AL 49)}), (s_7, {`loc 3`), (s_8, {`cmp` (regAdd8 AL 50)}), (s_9, {`push` (regi16 Ax)}), (s_{10}, {`mov` (regAdd16 AL 97)}), (s_{11}, {`pop` (regi16 Ax)}),(s_{12}, {`loc 4`), (s_{13}, {`loc 5`), (s_{14}, {`xor` (regReg16 Ax Ax)}), (s_{15}, {`retn`})

En la columna de la izquierda de la Figura 3.6 podemos observar un salto a la instrucción previa para ejecutar un bucle decrementando `ecx` en cada salto a través del código operacional `loop`. Este

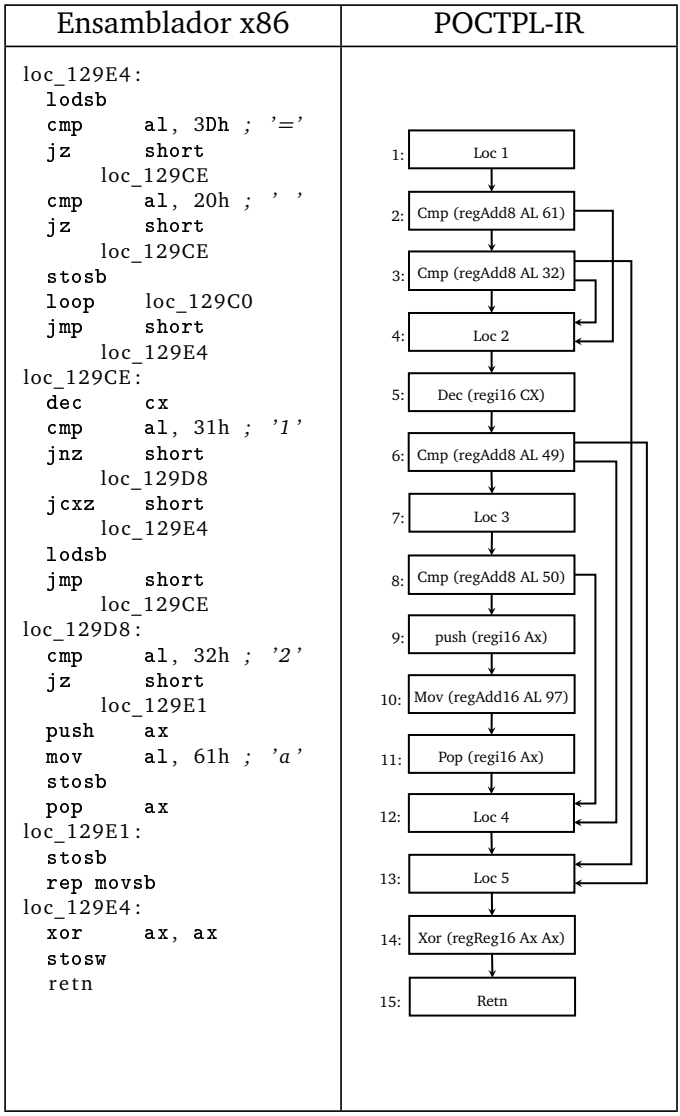

Figura 3.6: Equivalencia entre el lenguaje ensamblador de Intel y el código equivalente en POCTPL-IR.

tipo de saltos no son relevantes para POCTPL, ya que son parte del mismo camino, con lo que solo serán considerados los saltos hacia instrucciones posteriores.

En este ejemplo, podemos extraer seis posibles caminos que el

programa puede ejecutar. Después de la primera `cmp (regAdd8 AL 61)`, el programa puede ir a la instrucción `loc 2` o continuar hacía `cmp (regAdd8 AL 32)`, que a su vez puede continuar hacía `loc 5` o seguir hacía `dec (regi16 CX)`. Lo mismo sucede para la instrucción seis: `cmp (regAdd8 AL 49)`, y para la instrucción ocho: `cmp (regAdd8 AL 50)`. Ambas instrucciones pueden saltar hasta la instrucción 12: `loc 4`, o incluso la seis, puede llegar a saltar a la instrucción 13: `loc 5`. Considerando los números de cada instrucción los caminos posibles en este ejemplo son los siguientes:

- $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow s_{10} \rightarrow s_{11} \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow s_{10} \rightarrow s_{11} \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$
- $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_{12} \rightarrow s_{13} \rightarrow s_{14} \rightarrow s_{15}$

Aunque en este ejemplo no existen muchos caminos, en los programas que se analizarán en un entorno real tendrán muchos más caminos.

Con la estructura Kripke $\mathscr{M}$, es posible aplicar POCTPL para conocer si un camino satisface una fórmula matemática. Las siguientes fórmulas son ejemplos que satisfacen la estructura Kripke anterior:

- s $\vDash \exists\bigcirc$(op cmp (regAdd8 AL 32))

- s $\vDash$ $\exists$((op (mov (regAdd16 (AL 97)))) $\triangle$ (op (pop (regi16 Ax))))
- s $\vDash$ $\exists\Diamond$(op (mov (regAdd16 (AL 97)))) $\cup$ ($\exists\Diamond$ (op (xor (regReg16 Ax Ax))))
- s $\vDash$ $\exists\Diamond$(op (cmp (regAdd8 (AL 49)))) $\cup$ ($\exists$((op (cmp (regAdd8 (AL 49)))) $\mathcal{U}$ (op (pop (regi16 Ax))))

3.4 Conclusiones

El algoritmo creado en este capítulo es capaz de reconocer patrones dentro del código ensamblador, para ello utiliza fórmulas similares a las creadas para CTL. A través de estas fórmulas se puede detectar código de forma mucho más dinámica y flexible a como lo realizan las tradicionales firmas estáticas. Es por ello que los nodos utilizados en CTL en nuestro caso son instrucciones de código ensamblador.

Al utilizarse una notación similar a CTL es fácil que cualquier persona con conocimientos básicos de lógica puede llegar a entender rápidamente como realizar fórmulas, y en poco tiempo puede llegar a mayores complejidades en la creación de éstas.

La mayor complejidad de nuestro algoritmo para el usuario es la separación entre las fórmulas de caminos y las de estados. Aunque no es común que dos fórmulas de diferente tipo trabajen juntos, en nuestro caso éstas se complementan perfectamente, haciendo que la salida del sistema siempre muestre los caminos que cumplen la fórmula representada con estados.

Uno de los problemas que se pueden detectar durante la creación de los algoritmos es que a su vez están implementados en lenguajes no seguros, haciendo que los mismos algoritmos puedan cometer errores, con lo que mostrarán tanto falsos positivos como negativos. Por esta razón nosotros hemos implementado nuestro código en un lenguaje que comprueba matemáticamente las funciones para que éstas no contengan errores, lo que veremos en el capítulo siguiente.

«Nos cuesta admitir errores, porque eso significa renunciar a la seguridad que esos supuestos simplificadores nos proporcionan.»

Daniel Kahneman (1934–)

Capítulo

4

Verificación

Para realizar la implementación de POCTPL, hemos decido utilizar Coq, un verificador de teorías interactivo capaz de trabajar con múltiples datos. De este modo, usaremos Coq para asegurar que nuestro lenguaje es matemáticamente correcto probando los teoremas ya descritos.

La razón de utilizar un lenguaje verificado es muy simple. Parece poco razonable realizar un programa para encontrar errores en el código de terceros que a su vez pueda contener errores. Si el propio programa que realiza la verificación contiene errores significa que muchos problemas en el código analizado pueden ser pasados por alto, haciendo esta tarea totalmente inútil. Mediante Coq queremos conseguir que el *model checker* sea correcto (que cumple su propósito) y completo (que solo hace lo que está diseñado para hacer). Al hacer el programa mediante Coq también probamos que la aplicación tiene un significado matemático correctamente diseñado y se prueba que cumple las especificaciones del lenguaje.

Hasta el momento han sido pocos los *model checkers* que han utilizado un lenguaje verificado para realizar esta labor, haciendo que

nunca se pueda estar totalmente seguro de que un *model checker* ha encontrado todos los errores en el código, o bien que no interpreta como error un código que es correcto.

Como hemos visto en capítulos anteriores usamos la representación simbólica de fórmulas propuesta por Lewis y Langford [LLL59] para la representación de fórmulas. Pero en el código fuente de la aplicación verificada que hemos realizado hemos utilizado la implementación textual, ya que no es posible introducir símbolos en el código fuente. En la tabla 4.1 está representada la equivalencia entre estas dos representaciones.

Tabla 4.1: Equivalencia entre la representación simbólica y textual.

Simbólica	Textual
$\bigcirc\Phi$	XΦ
$\Diamond\Phi$	FΦ
$\Psi\ U\ \Phi$	Φ U Ψ
$\Box\Phi$	GΦ
$\Phi \bigtriangleup \Psi$	Φ FX Ψ
$\exists\Phi$	EΦ
$\forall\Phi$	AΦ

4.1 Estructura del lenguaje ensamblador

Antes de comenzar con la explicación de como hemos implementado las fórmulas POCTPL, es importante saber como será representado el lenguaje ensamblador en Coq. Esta programación es completamente funcional, acercándose más al concepto matemático de los lenguajes de programación.

Para este fin, hemos creado estructuras inductivas para construir la representación que tomará dicho lenguaje en nuestro sistema. La Figura 4.1 muestra tres estructuras inductivas creadas para representar los registros de la CPU.

```
Inductive register32:Set :=
   | EAX : register32 | EBX : register32
   | ECX : register32 | EDX : register32
   | ESI : register32 | EDI : register32
   | EBP : register32 | ESP : register32
   | EIP : register32 | CS : register32
   | DS : register32  | SS : register32
   | ES : register32  | FS : register32
   | GS : register32  | Register32: register32
   | EAX_plus : nat -> register32 | EAX_minus : nat -> register32
   | EBX_plus : nat -> register32 | EBX_minus : nat -> register32
   | ECX_plus : nat -> register32 | ECX_minus : nat -> register32
   | EDX_plus : nat -> register32 | EDX_minus : nat -> register32
   | EDI_plus : nat -> register32 | EDI_minus : nat -> register32
   | ESI_plus : nat -> register32 | ESI_minus : nat -> register32
   | EBP_plus : nat -> register32 | EBP_minus : nat -> register32
   | ESP_plus : nat -> register32 | ESP_minus : nat -> register32.

Inductive register16:Set :=
   | Ax : register16  | Bx : register16
   | Cx : register16  | Dx : register16
   | SI : register16  | DI : register16
   | SP : register16  | BP : register16
   | IP : register16  | Register16: register16
   | SP_plus : nat -> register16 | SP_minus : nat -> register16
   | BP_plus : nat -> register16 | BP_minus : nat -> register16.

Inductive register8: Set :=
   | AH : register8   | AL : register8
   | BH : register8   | BL : register8
   | CH : register8   | CL : register8
   | DH : register8   | Register8 : register8
```

Figura 4.1: Registros en Coq.

Cada estructura sirve para tratar los diferentes tamaños de los registros en la CPU: registros de 32 bits, de 16 y de 8 bits. Además, en los tipos `register32` y `register16` hay otros registros para simular las operaciones de suma y resta con los registros que son necesarios para hacer cálculos con la pila. Mediante estas tres estructuras de datos cubrimos todos los posibles valores que puede tener una variable en el compilador y que nos ayude a nuestra búsqueda de patrones para encontrar vulnerabilidades.

Para representar un código operacional con sus parámetros del registro hemos creado dos estructuras inductivas llamadas `holdOne`

y `holdTwo`.

```
Inductive holdOne:Set:=
    | regi32:register32->holdOne
    | addr32:mem->holdOne
    | regi16:register16->holdOne
    | addr16:mem->holdOne
    | regi8:register8->holdOne
    | addr8:mem->holdOne.
    | val:nat->holdOne.
```

Figura 4.2: Tipo HoldOne para códigos operacionales que solo tienen un argumento.

```
Inductive holdTwo:Set:=
    | regAdd32:register32->nat->holdTwo
    | addAdd32:mem->mem->holdTwo
    | addReg32:mem->register32->holdTwo
    | regReg32:register32->register32->holdTwo
    | regVal32:register32->nat->holdTwo
    | regAdd16:register16->mem->holdTwo
    | addAdd16:mem->mem->holdTwo
    | addReg16:mem->register16->holdTwo
    | regReg16:register16->register16->holdTwo
    | regVal16:register16->nat->holdTwo
    | regAdd8:register8->mem->holdTwo
    | addAdd8:nat->nat->holdTwo
    | addReg8:nat->register8->holdTwo
    | regReg8:register8->register8->holdTwo
    | regVal8:register8->nat->holdTwo
    | addVal:mem->nat->holdTwo.
```

Figura 4.3: Tipo HoldTwo en Coq, para códigos operacionales que tiene dos argumentos.

Las estructuras `holdOne` y `holdTwo` mostradas en las Figuras 4.2 y 4.3 son las que dirán el número de argumentos que puede contener un código operacional. Dentro de cada estructura podemos observar el tipo de operaciones que son aplicables en cada una de ellas, además del número de bits que lleva la operación. Hay un tipo que no lleva número de bits y sería el de la operación entre

una dirección de memoria y un valor, ya que a términos de nuestro programa no tiene una importancia vital conocer el tamaño de esta operación.

Para diferenciar las diferentes direcciones de memoria y valores en lenguaje ensamblador, usamos dos tipos diferentes: el tipo para referenciar las direcciones de memoria es llamado `mem` y es usado para representar cualquier operación que se haga con la memoria. Los valores inmediatos (como por ejemplo los literales que son especificados como parámetros de una instrucción en ensamblador) son representados con el tipo `Val` en Coq, que en la práctica es un entero.

Como hemos comentado anteriormente, considerando los códigos operacionales relevantes mostrados por Bilar [Bil07], definimos la estructura inductiva *opcode*. Además, hemos incluido la instrucción «`loc`» que pertenece a la representación de IDA Pro para los bloques de memoria, explicado en el capítulo anterior.

```
Inductive opcode:Set :=
 | mov:holdTwo->opcode   | push:holdOne->opcode
 | call:function->opcode | pop:holdOne->opcode
 | cmp:holdTwo->opcode   | jz:holdOne->opcode
 | lea:holdTwo->opcode   | test:holdTwo->opcode
 | jmp:holdOne->opcode   | add:holdTwo->opcode
 | jnz:holdOne->opcode   | retn:opcode
 | xor:holdTwo->opcode   | and:holdTwo->opcode
 | loc: nat->opcode.
```

Figura 4.4: Los códigos operacionales que se valoran dentro de POCTPL.

Como vemos en la Figura 4.4 los códigos operacionales tiene asignados un número de argumentos, con lo cual cada operación tiene un número fijo de argumentos. Un caso especial sería el código operacional «`call`» el cual llamará a las funciones que se consideren peligrosas y o que puedan llegar a incurrir en una vulnerabilidad.

4.2 La estructura Kripke en Coq

La estructura Kripke que hemos implementado en Coq la mostramos en la Figura 4.5:

```
Record kripke: Set :=
   KripkeS {
       S_List : list nat;
       R_List : list (nat nat);
       L_List : list (nat opcode)
   }.
```

Figura 4.5: Estructura Kripke en Coq.

Como hemos comentado anteriormente, la estructura Kripke será usada para almacenar el grafo que será analizado por POCTPL. La estructura interna es similar a la que hemos descrito en el capitulo anterior:

- *S* (S_List) guarda los valores numéricos representando el número de nodo. El primer nodo del grafo siempre será el número uno, y se irá incrementando.
- *R* (R_List) guarda la relación entre nodos. Como en POCTPL, si el flujo de control no puede saltar a los nodos previos el grafo será un grafo acíclico o, en nuestro caso, un árbol.
- *L* (L_List) guarda las relaciones entre nodos y su contenido. Un ejemplo de esta relación lo podemos observar en la Figura 4.6.

```
(1,mov (regAdd32 EAX 1)::(2,add (regReg32 EDI EAX))::(3,call gets)::nil
```

Figura 4.6: Ejemplo de *R* (R_List) .

En Coq, la estructura Kripke se representa de tal forma que puede ser fácilmente manejada por el programa. En este caso, hemos decidido transformar la estructura Kripke en una lista de listas y hemos creado una nueva notación para hacer las pruebas más fáciles. Por lo cual definimos un camino como el que vemos en la Figura 4.7.

```
Inductive Path (A:Type):Type :=
   | PNil : Path A
   | PCons : opcode -> Path A -> Path A.

Infix "-->" := (PCons opcode)
(at level 60, right associativity)
: list_scope.

Notation " [ ] " := (PNil opcode)
(at level 30).

Definition POCTPLPath: Type := Path opcode.
```

Figura 4.7: Definición de un camino POCTPL.

Como vemos en la Figura 4.7 cambiamos al notación de los caminos en Coq, para que se vean mejor. La notación de la separación por comas entre un estado y el siguiente no es muy práctica de cara a ver fácilmente donde termina un estado y empieza el siguiente. También cambiamos la forma en la que se muestra el final del camino haciendo que sea también más claro ver donde finaliza éste. Con todo esto, en Coq, se pueden expresar los caminos como vemos en la Figura 4.8:

```
(1,mov (regAdd32 EAX 1)-->(2,add (regReg32 EDI EAX))-->[]
```

Figura 4.8: Salida mostrada por Coq.

Otra notación que es importante para entender como funciona POCTPL en Coq, es el hecho de que para Coq, un árbol es una lista

de caminos y por esta razón definimos el tipo `POCTPLTree` como una lista de caminos. Tal y como puede verse a en la Figura 4.9:

```
Definition POCTPLTree:Type := list POCTPLPath.
```

Figura 4.9: Definición de POCTPLTree.

Estas definiciones simplifican el concepto de la estructura Kripke y facilitan el trabajo con ella en Coq, haciendo el código más intuitivo a primera vista.

Como hemos visto, la estructura Kripke que nosotros hemos creado no contiene la variable I, que tiene la estructura Kripke original. Esto es debido a que no la generamos directamente mediante Coq, sino que cuando se crean los caminos en la parte generada en Python[1] (explicado con detalle en el siguiente capítulo), se valoran los posibles puntos de entrada del programa y se le van pasando a la parte generada por Coq. Al hacer esto, es posible que Python realice una parte del trabajo que sería para el proceso que maneja el código desarrollado en Coq, liberando a este de gran parte de la carga. También sería posible hacer optimizaciones sobre la búsqueda de los caminos a examinar, restando tiempos en procesos que puedan ser mayores.

4.3 POCTPL

Como hemos mencionado, al contrario que en CTPL, en POCTPL debemos considerar los diferentes tipos de conjuntos, y también definiremos distintas operaciones para los grupos de estados y de caminos. Estos dos tipos de conjuntos son independientes y tienen que ser tratados separadamente en Coq por su incompatibilidad de tipos. En algunos casos, este hecho hace necesario implementar dos versiones de la misma operación, de esta manera podremos operar con los dos tipos de operandos.

[1] https://www.python.org/

```
Inductive opStateFormula :=
   | TRUE : opStateFormula
   | op : opcode->opStateFormula
   | noS : opStateFormula->opStateFormula
   | andS : opStateFormula->opStateFormula->opStateFormula
   | orS : opStateFormula->opStateFormula->opStateFormula
   | impS : opStateFormula->opStateFormula->opStateFormula
   | coimpS : opStateFormula->opStateFormula->opStateFormula.
```

Figura 4.10: Operaciones realizadas entre estados.

Las diferentes operaciones con estados en POCTPL se muestran en la definición del tipo `opStateFormula` que podemos ver en la Figura 4.10. Este tipo representa un conjunto de estados en la definición formal que describimos en la sección 3.3. Como podemos ver, todas las operaciones definidas aceptan como argumentos un tipo `opStateFormula`, esto quiere decir que las operaciones se pueden ir repitiendo unas dentro de otras, y recursivamente se irán resolviendo, hasta que llegue un momento en el que encontremos un `op` o un `true`. `True` devuelve todos los estados posibles en el sistema mientras que `op` manejará las operaciones que tengan códigos operacionales.

La Figura 4.11 muestra la implementación realizada para la función `SatState` que realiza todas las operaciones para estados declarados por el tipo `opStateFormula`. Como podemos ver, las funciones `TRUE`, `op`, `noS` and `andS` están creadas directamente en Coq, pero las operaciones de unión (`orS`) e implicación (`impS`) son derivadas a través de los axiomas de álgebra de conjuntos usando las operaciones de negación (`noS`) e intersección (`andS`). Las operaciones `orS` y `impS` son implementadas usando las funciones previamente programadas y por esta razón no es necesario realizarlas separadamente.

La Figura 4.12 muestra la declaración para el tipo `opType`. Este tipo representa un conjunto de caminos en la especificación de POCTPL descrita en la sección 3.3. Como podemos observar, todas las operaciones definidas para este tipo devuelven valores de tipo `opType`. Sin embargo, no todas las operaciones aceptan `opType` co-

```
Fixpoint SatState (o:opStateFormula)
(l:list opcode): list opcode :=
    match o with
       | TRUE => l
       | op a => opInList a l
       | noS a => negState l (SatState a l)
       | andS a b => interState
          (SatState a l)(SatState b l)
       | orS a b => negState l (interState
          (negState l (SatState a l))
          (negState l (SatState b l)))
       | impS a b => negState l
          (interState (SatState a l)
          (negState l (SatState b l)))
       | coimpS a b => interState (negState l
          (interState (SatState a l)
          (negState l (SatState b l))))
          (negState l (interState
          (SatState b l) (negState l
          (SatState a l))))
   end.
```

Figura 4.11: Función SatState.

mo operandos. Por un lado, están `id` que devuelven un tipo `opType` (conjunto de caminos) con el camino en la estructura Kripke (si está presente) que contiene todos los estados especificados en el tipo conocido `opStateFormula`. Por otro lado, las operaciones `EX`, `EU`, `EG`, `EF`, `AG`, `AX`, `AF` y `AU` con una segunda versión `EXS`, `EUS`, `EGS`, `EFS`, `AGS`, `AXS`, `AFS` que acepta operandos del tipo `opStateFormula` (conjunto de estados) en vez de `opType`. Estas operaciones son implementadas por separado porque ambos tipos son completamente independientes y no pueden estar juntos en una función realizada en Coq.

La figura 4.13 muestra la implementación de la función `Sat`, que es la base de todo el sistema para controlar como se introducen las fórmulas en Coq. Similarmente a la función `SatState` muchas operaciones son creadas considerando la derivación a través de los axiomas descritos en la sección 3.3. De este modo, `andC` y `impC` son obtenidos a través de la derivación con las operaciones `noC` y `orC`. Como podemos ver en la Figura 4.13 la realización de estas

```
Inductive opType :=
    | id : opStateFormula -> opType
    | noC : opType -> opType
    | orC: opType -> opType -> opType
    | andC : opType -> opType -> opType
    | impC : opType -> opType -> opType
    | coimpC : opType -> opType -> opType
    | EX: opType -> opType
    | EXS: opStateFormula -> opType
    | EU : opType -> opType -> opType
    | EUS: opStateFormula -> opStateFormula
          -> opType
    | EG: opType -> opType
    | EGS: opStateFormula -> opType
    | EF: opType -> opType
    | EFS: opStateFormula -> opType
    | AG: opType -> opType
    | AGS: opStateFormula -> opType
    | AX: opType -> opType
    | AXS: opStateFormula -> opType
    | AF: opType -> opType
    | AFS: opStateFormula -> opType
    | AU: opType -> opType -> opType
    | AUS: opStateFormula -> opStateFormula
          -> opType.
```

Figura 4.12: Operaciones que se pueden realizar con caminos.

funciones está basada en otras previamente definidas, y basándose en los axiomas de los cuales se derivan estas operaciones. Un ejemplo muy sencillo de ver esta derivación lo tenemos en la función `EF`, es derivada desde `EU`, donde la primera parte siempre está a `true`. Finalmente, todas las funciones basadas en el operador $\forall$ están definidas basándose en la anterior operación análoga con el operador $\exists$. Las dos operaciones llaman a la misma función pero realizando diferentes operaciones.

Los axiomas conocidos de CTL y que usamos en POCTPL, son:

- $\text{EF}\varphi = \text{TRUE}\ U\varphi$
- $\text{EG}\varphi = \neg\text{F}\neg\varphi$

Como podemos observar, esta estructura está preparada para ser

```
Fixpoint Sat (o:opType)(lo:list opcode)(l:CTPLTree){struct o}:
CTPLTree := match o with
   | id a => inTree (SatState a lo) l
   | noC a => neg l (Sat a lo l)
   | orC a b => neg l (inter (l \ (Sat a lo l))(neg l (Sat b lo l)))
   | andC a b => inter (Sat a lo l)(Sat b lo l)
   | impC a b => (neg l (inter (Sat a lo l)(neg l (Sat b lo l))))
   | coimpC a b => inter (neg l (inter (Sat a lo l)(neg l (Sat b lo l)))
      ) (neg l (inter (Sat b lo l)(neg l (Sat a lo l))))
   | EX a => operationXC (Sat a lo l) l
   | EXS a => operationX (SatState a lo) l
   | EU a b => operationUC (Sat a lo l)(Sat b lo l) l
   | EUS a b =>operationU(SatState a lo)(operationUS(SatState b lo) l)
   | EG a => operationGC (Sat a lo l) l
   | EGS a => operationG (SatState a lo) l
   | EF a => (operationUC (lo2CTPLTree lo)(Sat a lo l) l)
   | EFS a => operationU (SatState TRUE lo)(operationUS (SatState a lo)
      l)
   | EFX a b => operationFXC (Sat a lo l)(Sat b lo l) l
   | EFXS a b => operationFX (SatState a lo)(SatState b lo) l
   | AG a => neg l (operationUC (lo2CTPLTree lo)(neg l (Sat a lo l)) l)
   | AGS a => neg l operationU (SatState TRUE lo)(operationU2 (negState
      lo (SatState a lo)) l)
   | AX a => neg l (operationXC (neg l (Sat a lo l)) l)
   | AXS a => neg l (operationX (negState lo (SatState a lo)) l)
   | AF a => neg l (operationGC (neg l (Sat a lo l)) l)
   | AFS a => neg l (operationG (negState lo (SatState a lo)) l)
   | AU a b => inter (neg l (operationUC (neg l (Sat b lo l)) (inter ((
      neg l (Sat a lo l)) (neg l (Sat b lo l))) l))) (neg l (
      operationGC (l \ (Sat b lo l)) l))
   | AUS a b => inter (neg l (operationU  (negState (SatState b lo) lo)
      (operationU2 (interState (negState (SatState a lo) lo) (negState
           (SatState b lo) lo)) l))) (neg l (operationG (negState (
      SatState b lo) lo) l))
   | AFX a b => inter (neg l   (operationFXC (neg l (Sat b lo l)) (inter
       ((neg l (Sat a lo l)))  (neg l (Sat b lo l))) l)) (neg l (
      operationGC(neg l (Sat b lo l)) l))
   | AFXS a b => inter (neg l (operationFX (negState(SatState b lo) lo)
      (interState (negState    (SatState a lo) lo) (negState (SatState
      b lo)lo)) l)) (neg l (operationG (negState        (SatState b lo)
       lo) l))
end.
```

Figura 4.13: Función Sat.

recursiva, por lo cual es posible incluir operaciones dentro de las propias operaciones tal y como sucede en las matemáticas normales.

4.4 Notación

Las llamadas a las librerías de C deben de ser especialmente consideradas, ya que en numerosos casos las llamadas incorrectas a estas librerías son las que producen múltiples errores y vulnerabilidades. Aunque en ensamblador puro no aparecen las llamadas a dichas librerías, por medio de la salida que nos devuelve el análisis efectuado por IDA Pro es posible ver las llamadas a estas librerías, con lo que se facilita mucho su búsqueda. Para abstraer estas direcciones de memoria y facilitar la interoperatibilidad(cada sistema puede cargar las librerías de C en diferentes regiones de la memoria), las llamada a dichas funciones en las librerías C son abstraídas usando una notación especifica. En este caso solo usaremos las llamadas a las funciones más peligrosas según la página web de Microsoft «*Security Development Lifecycle (SDL) Banned Function Calls*»[2] para evitar incluir todas las posibles funciones que tiene C. La inclusión de muchas librerías en la base de datos hace que Coq tenga que comprobar cada una de ellas a la hora de constatar si una función corresponde con la que se busca, haciendo el análisis más lento. Por eso es vital controlar el número de funciones que van a ser utilizadas, y no rellenar el programa con llamadas que no serían utilizadas.

Con esta incorporación es más sencillo definir una fórmula usando el nombre de la función, que apunta a una determinada región en la memoria. La Figura 4.14 muestra un ejemplo de una llamada a la función `strcpy`.

Para facilitar el proceso de escribir fórmula POCTPL, definimos

[2]http://msdn.microsoft.com/en-us/library/bb288454.aspx

```
mov eax, [ebp+4]
mov [esp+4], eax
mov eax, [ebp+8]
mov [esp], eax
call strcpy
```

Figura 4.14: Ejemplo de llamada a `Strcpy`.

una gramática especifica dentro de Coq para simplificar ciertas expresiones, consiguiendo de esta manera, tener una sintaxis más clara. Esta gramática usa corchetes (`[...]`) para los operaciones con estados y paréntesis (`(...)`) para las de caminos. De esta manera, las operaciones con corchetes seíian equivalentes a las funciones que aceptan conjuntos de estados como operandos (`EXS, EUS, EGS, EFS, AGS, AXS, AFS`), mientras que las expresiones con paréntesis se referirán a las operaciones análogas con conjuntos de caminos (`EX, EU, EG, EF, AG, AX, AF`). Esta notación abstrae al usuario que maneje Coq de cambiar el nombre de la llamada dependiendo de la función que maneje (la Figura 4.15 ilustra esta notación).

```
Notation "EX[ A ]" := (EXS A) (at level 30).

Notation "EX( A )" := (EX A) (at level 30).
```

Figura 4.15: Notación de EX para estados y caminos.

La misma notación se usa para el resto de las operaciones con la excepción de la fórmula EU. En éstas la notación cambia ligeramente, por ello es necesario escribir una coma entre las dos fórmulas para hacer más fácil distinguir ambos operandos. En la Figura 4.16 podemos ver un ejemplo de esta notación.

```
Notation "EU( A , B )" := (EU A B) (at level 30).

Notation "EU[ A , B ]" := (EUS A B) (at level 30).
```

Figura 4.16: Notación de la fórmula EU para estados y caminos.

Para simplificar la introducción de códigos operacionales en las fórmulas, definimos una notación basada en el carácter "`|`" para denotar códigos operacionales (esta operación puede verse en la Figura 4.17).

La Figura 4.18 ilustra las tres notaciones definidas previamente.

```
Notation "| A |":= (cardinality A) (at level 30).
```

Figura 4.17: Notación utilizada para los códigos operacionales en Coq.

La fórmula devolverá el conjunto de caminos que tienen tanto una llamada a `strcpy` y a `gets` en cualquier punto del programa.

```
EF[|call strcpy|] and EF[|call gets|]
```

Figura 4.18: Sencillo ejemplo de una fórmula POCPL.

En la Figura 4.18 observamos que aunque la fórmula `and` no tiene corchetes, EF los necesita, ya que la fórmula EF trabaja con estados, como se puede interpretar por la notación `|call gets|`.

Todas estas notaciones ayudan al usuario a escribir fórmulas POCTPL de una manera más sencilla en Coq, abstrayéndole de las diferentes implementaciones para las funciones que aceptan estados o conjuntos de caminos. Como resumen podemos decir que las reglas importantes para escribir fórmulas en POCTPL son tres: (i) las de caminos se escriben con paréntesis, (ii) las de estado se escriben con corchetes y (iii) los códigos operacionales se escriben siempre entre tuberías.

4.5 Verificación de la implementación

Mediante Coq podemos verificar la implementación de POCTPL descrita anteriormente, comprobando que el código es correcto a través lemas y teoremas. Como POCTPL está basado en el álgebra de conjuntos, podemos verificar nuestras funciones usando las leyes y axiomas de dicho álgebra. Más concretamente, nosotros verificamos nuestras implementaciones de `or`, `and`, `imp` y `not` para que satisfagan las siguientes leyes:

Asociativa: Las fórmulas `or` y `and` satisfacen las leyes asociativas. En otras palabras, **A** `and` (**B** `and` **C**) es equivalente a (**A** `and` **B**) `and` **C**.

Distributiva: Las fórmulas `or` and `and` cumplen esta ley. Por tanto, **A** `and` (**B** `or` **C**) es equivalente a (**A** `and` **B**) `or` (**B** `and` **C**).

Conmutativa: Tanto la fórmula `or` y como la fórmula `and` satisfacen la ley conmutativa. De esta manera, **A** `and` **B** es equivalente a **B** `and` **A**.

Idempotencia: Ambas fórmulas, `or` y `and`, satisfacen la ley de idempotencia, con lo cual **A** `and` **A** es igual a **A**, y lo mismo sucede con la fórmula `or`.

Absorción: Esta ley señala que **A** `and` (**A** `or` **B**) es igual a **A**. Lo mismo sucede con **A** `or` (**A** `and` **B**) la cual es **A**.

Doble complemento: $\neg\neg$**A** es igual a **A**.

Leyes de Morgan: A través de las leyes de Morgan podemos probar que $\neg$**A** `and` **B** es equivalente a **A** `or` $\neg$**B**. También podemos probar que $\neg$**A** `or` **B** es igual a **A** `and` $\neg$**B**.

Conjunto vacio: $\mathbf{A} \cup \varnothing = \mathbf{A}$ y $\mathbf{A} \cap \varnothing = \varnothing$.

En base a estar reglas realizamos la verificación de nuestro lenguaje usando las herramientas que nos ofrece Coq. La metodología está basada en la verificación de cada función envuelta en `Sat` y `SatState`. De esta forma, el proceso de verificación es más fácil a través de las herramientas de Coq para la verificación. Al probar estas leyes también podemos verificar que las operaciones `imp` y `or` están implementadas correctamente.

4.6 Validación del código

La forma de validar en funciones se realiza mediante teoremas. De esta forma comprobamos que una función puede llegar a ejecutar

cualquier tipo de código sin errores.

Un ejemplo sencillo de ver es la comprobación de la comparación de los registros de 32 bits.Veamos como se ha realizado la validación de la función `regCompare32`. Esta función simplemente devuelve «`true`» si la comparación es verdadera y «`false`» si no coinciden los registros que se le han pasado.

```
Theorem regCompare32T1: forall (r:register32),
    regCompare32 r r = true.
Proof.
   intros.
   induction r; simpl; auto;
   rewrite natCompBT1; reflexivity.
Qed.
```

Figura 4.19: Validación de Teorema 1 para `regCompare32`.

El primer teorema que se comprobó es el que podemos observar en la Figura 4.19. Aquí se comprueba que para cualquier registro del tipo `register32` (que hemos visto anteriormente), si el registro es igual éste siempre devolverá «`true`». Como se puede observar después de varias operaciones esto queda probado mediante la orden Qed (que en Coq significa que la fórmula queda validada).

```
Theorem regCompare32T2: forall (r1 r2:register32),
    regCompare32 r1 r2 = regCompare32 r2 r1.
Proof.
   intros.
   induction r1; elim r2; intros; simpl; auto;
   rewrite natCompBT2; reflexivity.
Qed.
```

Figura 4.20: Validación de Teorema 2 para `regCompare32`.

Otra comprobación que se ha realizado a la función `regCompare32` ha sido la que podemos observamos en la Figura 4.20. En ella validamos que el programa tiene que ser igual si el orden de los factores

cambia, sea cual sea el registro que le pasamos. Como se puede observar, esta función también queda validada mediante Coq de manera formal.

```
Theorem interT7: forall (x:CTPLTree),
   inter x x = x.
Proof.
   intros.
   induction x.
   simpl; auto.
   simpl.
   rewrite pathCompareT3.
   rewrite interT3.
   simpl.
   rewrite IHx.
   admit.
Qed.
```

Figura 4.21: Comprobación de la función que implementa la operación de intersección.

El teorema de la Figura 4.21 es más complejo que los anteriores debido a lo que implica un dato de tipo `CTPLTree`, es decir, un árbol completo. Esto hace que las comprobaciones sean más complejas, ya que hay que comprobar que para cada camino del árbol y para cada estado del árbol la función funciona. Esto se puede hacer apoyándose en otros teoremas ya comprobados. En el teorema que estamos examinando podemos observar que se utilizan los teoremas «`pathCompareT3`» e «`interT3`» para realizar ciertas operaciones, facilitando la comprobación de teoremas.

Como se puede comprobar en estos ejemplos las pruebas de validación son complejas, pero permiten certificar que la función se desarrolla correctamente. Si se diera el caso de que ésta estuviera mal programada, nunca se podría finalizar el teorema con un «`Qed`». En muchas ocasiones durante el intento de comprobar un teorema se puede observar donde está el error, permitiendo que el programador cambie el punto de vista para afrontar el teorema.

4.7 Ejemplo de uso dentro de Coq

Aunque el propósito principal del `model checker` no es la ejecución directamente sobre Coq, es importante hacer estas pruebas para comprobar que funciona correctamente antes de exportarlo a Ocaml para tener un ejecutable. Para realizar estas pruebas solo necesitamos definir una función en Coq que acepte la estructura Kripke con el código ensamblador transformado y la fórmula para validar.

```
Eval compute in poctplMC
(id(op (mov (regAdd32 EAX 5))))
(KripkeS (1::2::3::nil)
((1,2)::(1,3)::(2,3)::nil)
((1,mov (regAdd32 EAX 5)::
(2,mov (regAdd EBX 1)::
(3,retn)::nil)))).
```

Figura 4.22: Ejemplo de ejecución de `poctplMC` con una estructura Kripke sencilla.

En la Figura 4.22 vemos un ejemplo de la ejecución de la función que sirve para ejecutar todo el programa, `poctplMC`. La salida de esta ejecución se muestra en la Figura 4.23. En este ejemplo, el programa devuelve dos caminos que satisfacen la fórmula (en este caso los que empiezan por "`mov (regAdd32 EAX 5))`".

```
(mov (regAdd32 EAX 5) —> mov (regAdd EBX 1) —> retn —> []) ::
(mov (regAdd32 EAX 5) —> retn —> []) :: nil
```

Figura 4.23: Salida de Coq devolviendo dos caminos.

La ejecución de código dentro de Coq tiene dos problemas que hay que destacar.

El primer problema es que Coq es un verificador de teorías y no genera ningún tipo de programa ejecutable haciendo imposible su utilización en conjunto con otro lenguaje de programación o con

otro programa. Otro punto es la rapidez de ejecución. Coq es un lenguaje bastante lento porque no está preparado para ello, sino para realizar pruebas de validación de teorías. Aun así, se sigue programando mucho con Coq, además de los ejemplos que hemos visto, INRIA trabaja en un analizador de C verificado, del cual han sacado sus primeros estudios en el 2015 [JLB+15].

Como para nosotros era importante que el código de Coq trabajara en conjunto con otras aplicaciones (como IDA Pro) y que realizara las operaciones de la forma más rápida posible, era imprescindible que el código se exportará a un lenguaje más potente, en este caso Ocaml. Desde Coq es posible extraer el código a Ocaml de manera muy simple, conservando las propiedades que tiene el código de Coq. De este modo, tenemos un ejecutable programado en Ocaml funcional y plenamente verificado con solo una orden desde Coq.

4.8 Conclusiones

La programación funcional es un paradigma que es difícil de adaptar para los programadores de lenguajes imperativos o incluso los orientados a objetos. Aún con esto, los lenguajes funcionales tienen la ventaja de ser más orientado a los lenguajes matemáticos, permitiendo la verificación del propio lenguaje.

Esto es lo que hemos llevado a cabo en este capítulo utilizando Coq. Se han verificado matemáticamente mediante el calculo de predicados cada una de las funciones que componen el algoritmo COCTPL. A su vez, se ha comprobado también la validez de su lenguaje intermedio POCTPL-IR. Con esta comprobación hemos validado que con cada posible entrada a nuestro sistema éste se va ejecutar correctamente sin cometer errores.

Aunque el trabajo para desarrollar este proceso es mayor que el necesario para realizar este mismo programa en un lenguaje imperativo, las ventajas de que éste va a funcionar correctamente en cualquier circunstancia son mayores en términos de seguridad.

«Sobrestimamos el evento y subestimamos el proceso, cada sueño realizado ocurrió gracias a la dedicación a un proceso.»

John C. Maxwell (1947–)

Capítulo

5

Resultados y complejidad

Con el fin de evaluar nuestra lógica en un entorno real, hemos implementado una herramienta que está compuesta por dos módulos diferentes, uno programado en Python (usado para transformar el código ensamblador a POCTPL-IR) y el otro escrito en Ocaml (el *model checker* de nuestra lógica).

La decisión de utilizar estos dos lenguajes viene determinada por tener un software completamente verificado y sin errores en su implementación, como hemos visto en el capitulo anterior. La utilización de Python viene determinada por conseguir una programación ágil y poco propensa a errores.

La utilización del Ocaml ha venido impuesta por el uso de Coq para realizar la verificación. La extracción del código desde el verificador solo puede ser realizada a tres lenguajes: Ocaml, Haskell y Scheme. Hemos escogido Ocaml porque es un lenguaje que está siendo utilizado ampliamente en el terreno científico y en la segu-

ridad de código, como lo demuestran proyectos como CompCert [Ler09a, Ler09b] o Bitblaze [SBY+08].

En las siguientes secciones vamos a ver como se ha realizado la plataforma, los resultados obtenidos y un análisis de la complejidad del algoritmo creado para POCTPL.

5.1 Plataforma

Vamos a analizar como funciona nuestra plataforma para el análisis de vulnerabilidades. Vamos a ir viendo paso a paso, como funciona cada parte de la plataforma y como ha sido tratado.

El primer paso para analizar un archivo binario es extraer el código ensamblador con el que se puede observar su comportamiento. Para este propósito usamos IDA Pro, pero no es necesario tener la versión de pago de esta herramienta, la gratuita nos ofrece el desensamblado de binarios x86, que es justamente lo que necesitamos. Para realizar esta tarea y que sea transparente al usuario se hace una llamada a IDA Pro mediante linea de comandos, lo que produce un binario con el mismo nombre que tiene el fichero que se está analizando y con extensión .asm.

Una vez que el código ha sido extraído y la estructura Kripke creada, parseamos la salida del ensamblador mediante Python. El parseo de este código es una tarea muy compleja debido a la cantidad de formas que pueden tomar los datos en este lenguaje. Además de realizar el parseo del código y transformarlo en POCTPL-IR hay que generar una estructura Kripke que muestre los diferentes caminos que puede tomar el binario durante su ejecución. Para hacer esto se realiza un análisis de los saltos y las funciones que contiene el binario para mostrarlo como diferentes vías de ejecución.

La propia herramienta posee una forma de mostrar la estructura Kripke mediante la herramienta Graphviz [EGK+02]. A través de esta salida gráfica, se puede ver como será la estructura Kripke resultante del análisis del binario que ha realizado IDA Pro. El resultado de uno de estos análisis de forma gráfica lo podemos observar en la Figura 5.1. Como comprobamos en la imagen, existen

tres tipos diferentes de nodos, los iniciales (en color verde), los finales (en color rojo) y los nodos (en color blanco). Podemos observar que existen gran número de nodos verdes, es decir, de puntos en los que IDA Pro no sabe exactamente como llegar hasta esa función o trozo del programa. Esto es debido a que se trata de funciones o trozos de código para los que el programa no puede calcular de manera estática como llegar hasta ese punto, ya sea porque calcula el salto en tiempo de ejecución, por trozos de código muerto por parte del compilador o por fallos en la heurística de IDA Pro. Los nodos finales, corresponden a códigos operacionales «`Retn`», lo que significa que el código ha llegado a un punto muerto en el que no puede continuar.

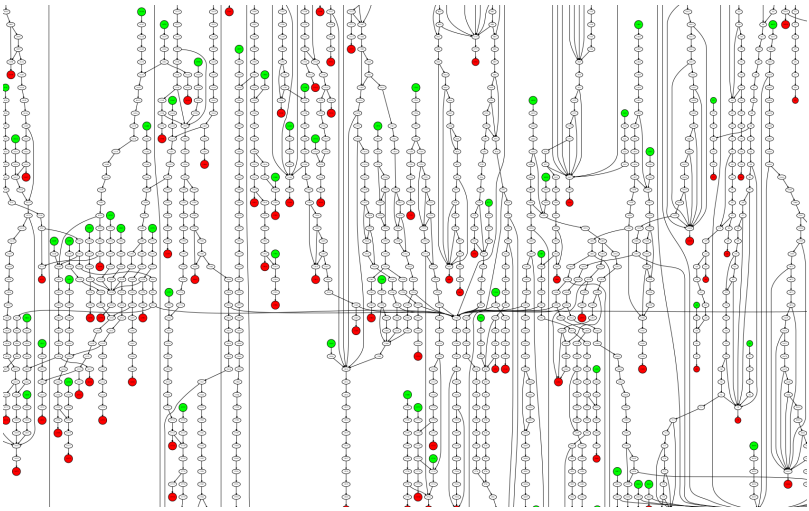

Figura 5.1: Captura de pantalla mostrando la salida gráfica de los caminos de un binario.

Por otro lado, el usuario introducirá una fórmula que representará el código que deseamos encontrar dentro del grafo que contiene el flujo de control que representa la estructura Kripke. Para asegurarnos que esta fórmula está realizada correctamente se le realiza un análisis léxico y sintáctico, de la misma forma que lo realiza un compilador al código fuente que está analizando. Mediante esta comprobación nos aseguramos de que la fórmula introducida esté

correctamente formada y no contenga errores. Un estudio más detallado de esta parte de la plataforma lo encontramos en el Apéndice A.

El segundo paso es transformar el código fuente desarrollado en Coq a Ocaml. Mediante esta transformación podemos compilarlo en un ejecutable normal e integrarlo con los módulos previos para ejecutar la lógica implementada en Coq.

Con todo esto funcionando ya tenemos la estructura que vemos en la figura 5.2 completa, y lista para probar cualquier programa.

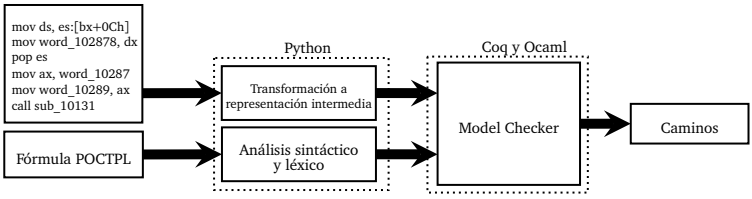

Figura 5.2: Diagrama que muestra el framework con los lenguajes de programación con los que ha sido realizada cada parte.

Una de las principales limitaciones que hemos encontrado durante la evaluación de esta plataforma es el proceso de desensamblado. A pesar de que IDA Pro es una de las mejores herramientas para realizar este proceso, todavía presenta algunas limitaciones relacionadas con la técnica de desensamblado. Algunos aspectos de código ensamblador son especialmente complejo para realizar un análisis estático. Este problema se puede ver claramente reflejado con los saltos indirectos que no pueden ser calculados estáticamente cuando el destino se calcula en tiempo de ejecución. Para intentar adivinar estos valores IDA Pro utiliza diferentes heurísticas considerando como funcionan los compiladores comúnmente. Desafortunadamente, éstas no pueden ser aplicadas para todos los casos y, en algunas ocasiones, el desensamblador identifica funciones o procedimientos pero no se puede determinar qué instrucciones saltan a esos bloques de código.

Como resultado, si consideramos el grafo del flujo de control (CFG) resultante del desensamblado mediante IDA Pro, no todo el

código puede llegar a ser alcanzado mediante la ejecución desde el punto de entrada del programa. Esto lo hemos solventado analizando cada uno de los estados que no tenían un estado anterior, marcándolos como puntos de inicio dentro del árbol. Con esto conseguimos que aunque IDA Pro no haya reconocido los puntos de acceso a un programa, un analista pueda ver la salida de un camino con una vulnerabilidad, y pueda analizar el código hasta llegar a ese punto, facilitando enormemente su labor.

Además, evitamos analizar el código correspondiente a las funciones importadas por las librerías (que se encuentra fuera del modulo que estamos analizando). Sin embargo como en el caso previo, sería posible generar una estructura Kripke para cada función exportada desde una librería para poder analizarla.

Los experimentos se han realizado en una máquina Intel Xeon 64bits 3.30GHz con 16 Gb de RAM y corriendo GNU/Linux.

5.2 Complejidad

Considerando la estructura generada por la salida de IDA Pro, podemos modelar la representación de un programa como un grafo K que satisface las siguientes propiedades:

1. Los arcos en K son dirigidos. La ejecución de instrucciones es secuencial, es decir, cada estado tiene un nodo predecesor y uno o dos estados subsecuentes.

2. K no tiene ciclos. La representación de el CFG como una estructura Kripke no considera bucles.

3. K solo tiene un estado inicial. Para cada ejecución, el código analizado solo tendrá un punto de entrada.

4. El máximo número de arcos de salida para cada nodo es 2. Los nodos de terminación (final del programa) tienen 0 arcos de salida. Los nodos intermedios correspondientes a códigos operacionales sin bifurcaciones que no alterarán el flujo del

programa, o los saltos incondicionales (el código operacional `jmp`) que solo tienen un arco de salida. Los nodos que evalúan una condición para alterar el flujo de ejecución del programa (saltos condicionales) tendrán dos arcos de salida diferentes. Los casos en los cuales los saltos indirectos pueden tener más de dos destinos diferentes no son considerados debido a las limitaciones de IDA Pro en el proceso de desensamblado.

De esta forma, la estructura Kripke puede ser considerada un árbol binario, en el cual el numero de caminos diferentes p_n que empiezan en el nodo n y finalizan en un nodo terminal t es igual al número de caminos para el subárbol formado por el primer nodo hijo y el número de caminos formados por el segundo nodo hijo. Dada esta propiedad, el numero de posibles caminos para n nodos es $P(n) = P(n-1) + P(n-2)$. Para cero nodos, el número posible de caminos es cero, para un nodo el numero posible de caminos es 1. Esta secuencia es equivalente a la sucesión de Fibonnaci, la cual es expresada por la función en la Ecuación 5.1.

$$p_n = \frac{\Phi^n - (1-\Phi)^n}{\sqrt{5}} \tag{5.1}$$

Una vez que tenemos el número de caminos, podemos calcular el máximo de estados para atravesar, S_n. La fórmula para calcular este número esta representada en la Ecuación 5.2. Como podemos observar, esta fórmula es una secuencia de Fibonacci convuelta, que probaron Hoggatt y Bicknell-Johnson [HJBJ77] en 1977.

$$S_n = \sum_{i=0}^{n+1} f(i) f(n-i) = f * f(n) \tag{5.2}$$

Dado el máximo numero de caminos desde un nodo simple (en este caso el punto de entrada de un programa), y conociendo el máximo número de estados que nuestro algoritmo va a atravesar, podemos estimar la máxima complejidad que tendrá POCTPL para una estructura Kripke con n estados. La ecuación que definiría este hecho la podemos observar en la Ecuación 5.3.

$$\mathcal{O}(n \cdot \Phi^n) \tag{5.3}$$

Sin embargo, cualquier software no va a presentar un número tan alto de caminos, considerando que la mayoría de los nodos solo tienen un arco de salida, es decir, la instrucción siguiente. En nuestros experimentos hemos medido el número de estados para atravesar por cada binario testeado. De este número depende completamente la complejidad del grafo.

Finalmente, también tenemos que tener en cuenta el problema de la explosión de estados cuando el número de estos crece enormemente. Hemos observado que el número máximo de caminos posibles se incrementa exponencialmente con el número de estados. El problema de la explosión de estados es uno de los clásicos del *model checking* y aunque existen acercamientos que intentan resolver este problema, queda por encima del enfoque de este trabajo.

5.3 Expresividad POCTPL

Podemos afirmar que POCTPL tiene una expresividad similar a la lógica de CTL. Cuando se han definido las fórmulas han sido pensadas para que fueran equivalentes a CTPL. Sin embargo, debido a la diferente naturaleza de las lógicas (CTPL está basada en lógica booleana, mientras que POCTPL lo está en el álgebra de conjuntos) hemos añadido otra operación ($\triangle$) para poder buscar una secuencia de *opcodes* inmediatamente precedida de otra sucesión de códigos operacionales. Aunque para este propósito CTPL usa el operador $\bigcirc$ en nuestro caso no se podía utilizar esta operación debido a que en las lógicas booleanas las operaciones se encadenan de diferente forma a como las encadenamos en POCTPL.

La principal contribución de POCTPL, es que los resultados devuelven conjuntos y gracias a esta propiedad es posible aplicarla a diferentes problemas. Por un lado, Kinder usó CTPL para detectar *malware* [KKSV05], para ello, definió las rutinas del código malicioso con la formulación de CTPL y con ello podía determinar si un binario contenía código malicioso. Por otro lado, existen áreas, co-

mo la búsqueda de vulnerabilidades en programas, para las cuales sería posible saber cuál es el proceso que lleva a desencadenar el error.

Por esta razón, creemos que está lógica mejorará en muchos aspectos el análisis de binarios, especialmente cuando se aplica a problemas que requieren identificar las partes del código que encajan con ciertas condiciones.

5.4 Vulnerabilidades encontradas

Una parte de las pruebas era demostrar la eficacia de la lógica desarrollada y el *model checker* mediante demostraciones en un entorno real. Por ello se llevaron a cabo pruebas con varios binarios buscando diferentes vulnerabilidades, que explicaremos en cada uno de ellos.

El criterio para la elección de estos binarios se ha basado en varios elementos. El primero, que el binario no fuese muy grande, para no tener que lidiar con el problema de explosión de estados. Un binario muy grande (con muchos caminos que analizar) puede hacer que la tarea del análisis dure semanas, haciendo que el estudio del binario sea muy engorroso en términos de tiempo.

Otro criterio para la elección del binario a examinar eran los problemas legales. Actualmente, las técnicas de ingeniería inversa pueden llegar a ser penadas por la ley[1] y son muchas las compañías que denuncian a analistas de seguridad por este hecho. En cualquier software moderno entre los términos y condiciones para usar el programa existe un punto en el que advierten de la prohibición de usar estas técnicas sobre el binario. Por esta razón, se ha escogido un software antiguo sin este tipo de restricciones, y que la realización de ingeniería inversa no traiga consigo problemas legales.

A continuación, se detallan aquellos binarios en los que se ha encontrado alguna vulnerabilidad que no habían sido encontradas

[1] http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31991L0250:EN:HTML

por ningún otro experto en seguridad, es decir, los comúnmente llamados *0-days*.

5.4.1 Simple Family Tree

Este programa es una aplicación para crear y gestionar árboles familiares. El usuario puede introducir todo tipo de datos sobre una familia, desde los típicos datos como fechas de cumpleaños y fotos, hasta eventos que han ocurrido durante el transcurso del árbol genealógico, como por ejemplo bodas, incluso puede introducir apodos con los que se llama a un miembro de la familia.

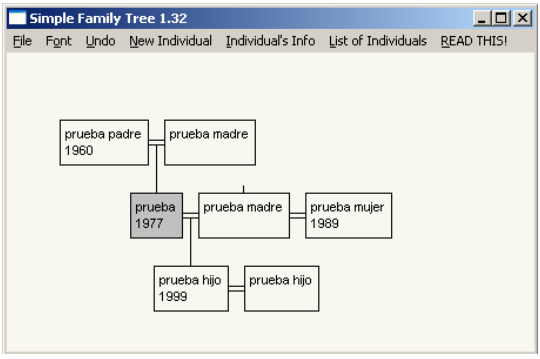

Figura 5.3: Captura de pantalla mostrando el programa SimpleFamilyTree.

La vulnerabilidad encontrada en este programa se produce por no controlar de forma correcta los datos que se introducen en la lectura de un fichero. En este caso la aplicación permite guardar los datos que ha ido generando durante el proceso de creación del árbol genealógico en un fichero con extensión `.ged`. Para leerlo el programa utiliza la llamada a la API «`ReadFile`», que tiene la forma que mostramos en la Figura 5.4.

Como observamos en la figura la llamada a esta orden necesita el número de bytes que van a ser leídos del fichero al cual le pasaremos un handle[2] correcto. En este caso esto lo soluciona haciendo antes

[2]Referencia a un bloque de memoria u objeto controlado por el sistema operativo

```
BOOL WINAPI ReadFile(
  _In_          HANDLE hFile,
  _Out_         LPVOID lpBuffer,
  _In_          DWORD nNumberOfBytesToRead,
  _Out_opt_     LPDWORD lpNumberOfBytesRead,
  _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

Figura 5.4: Estructura interna de la llamada a la API de Windows `ReadFile`.

una llamada a otra función de la API de Windows, `GetFileSize`, que tiene la forma que podemos observar en la Figura 5.5.

```
DWORD WINAPI GetFileSize(
  _In_       HANDLE hFile,
  _Out_opt_  LPDWORD lpFileSizeHigh
);
```

Figura 5.5: Estructura interna de la llamada a la API de Windows `GetFileSize`.

En este caso, con solo pasarle el *handle* del fichero, la función devolverá el tamaño que tiene el archivo. Esto podría estar bien si se realizara alguna comprobación de dicho tamaño en el trozo de código que va entre la llamada a `GetFileSize` y la llamada a `ReadFile`. Pero como podemos ver en la Figura 5.6 no sucede así.

En este caso después de la llamada a `GetFileSize`, la aplicación solo comprueba si lo que devuelve la función es cero, en cuyo caso toma otro camino, saltándose la llamada a la función `ReadFile`. Inmediatamente después de la instrucción `JE` hace una llamada a una función interna que no realiza nada con el tamaño del fichero ()que como podemos observar se guarda en `DS:[505148]`) y se le pasa posteriormente al registro `ECX`. Y después vemos como el valor de `ECX` se pasa mediante un `PUSH` a la función `ReadFile`, donde se puede llegar a realizar una escritura de la memoria completa al no existir ningún límite.

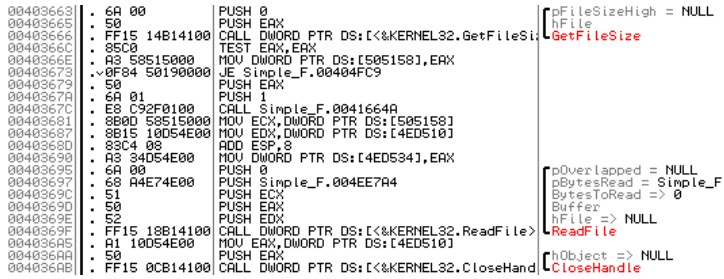

Figura 5.6: Captura de pantalla mostrando el trozo de código donde se produce el error en el programa Simple Family Tree.

En este caso la corrección de este error puede ser muy sencilla. Se podrían plantear varias posibles soluciones que ayudarían al programador a controlar este fallo en su código. La más simple sería poniendo un valor más pequeño o igual que el tamaño de la variable en la que se va a guardar el valor. De esta forma nos aseguraríamos que la llamada a la función `ReadFile` nunca pudiera desbordar el *buffer* porque no iba a leer más bytes que los que se le pasan en la llamada a función.

Aunque pueden existir otros métodos de control sobre los datos que devuelve la función `ReadFile`, lo más correcto es prevenir el error antes de que este se produzca y no controlar los datos que devuelve esta función. Aunque esta última solución puede resultar eficaz en términos de la ejecución del código, siempre es mejor prevenir los errores, que intentar repararlos por una mala llamada a una función.

5.4.2 FreeFloat FTP Server

El siguiente caso es de un servidor de FTP muy sencillo, FTP Server. Este programa es tan simple que no necesita apenas configuración, solo cuenta con un binario ejecutable «`exe`» que, una vez que se lanza, empieza a funcionar en el puerto 21 de nuestro ordenador permitiendo que cualquier usuario pueda acceder a la carpeta en la que se encuentra el programa.

Este programa tiene un problema al no controlar correctamente

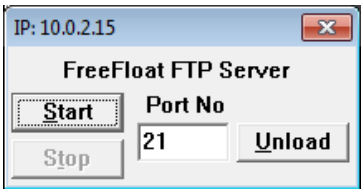

Figura 5.7: Captura de pantalla mostrando el programa FTPServer.

los parámetros que está recogiendo. En este caso el problema viene en la llamada a la función `recv`, que tiene la estructura que podemos observar en la Figura 5.8.

```
int recv(
  _In_   SOCKET s,
  _Out_  char *buf,
  _In_   int len,
  _In_   int flags
);
```

Figura 5.8: Estructura interna de la llamada a la API de Windows `recv`.

Como podemos observar, uno de los parámetros que recibe esta función es el tamaño del *buffer* que va a leer el *socket*[3]. Si este tamaño es introducido incorrectamente puede convertirse en un error de *buffer overflow*.

Ahora vamos a observar cual es el problema que tiene este programa. La función en la cual se llama a la parte vulnerable la podemos ver en la figura 5.9.

Comprobamos que al inicio de la función se introduce el valor 400 en el registro `ECX` que posteriormente es el que se introduce en la pila para llamar a la función `recv`. Ésta estaría bien programada si posteriormente el programa controlase verdaderamente el tamaño de esta variable que puede llegar hasta 400 caracteres y controlase la escritura de la pila, pero esto no sucede así.

[3]Sistema de comunicación entre procesos a través de la red

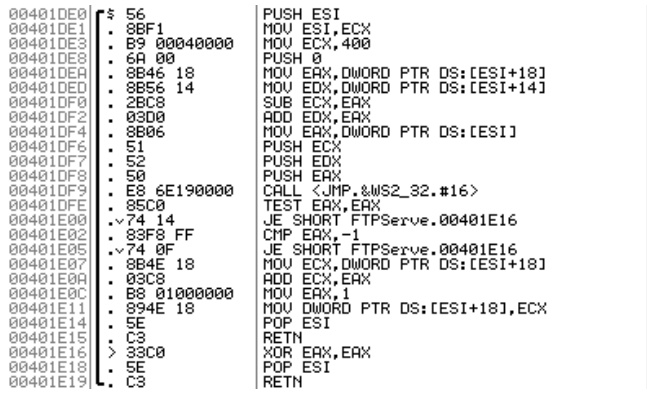

Figura 5.9: Captura de pantalla mostrando la función que contiene el error en FTPServer.

En este programa, posteriormente a la llamada a la función `recv`, se comprueba el tamaño que tiene el *buffer* introducido y se copia a la pila. El problema viene cuando se copia este valor ya que se hace sobre una variable local a la función para la que solo se han reservado 100 bytes. Esto trae consigo que si un usuario introduce un dato mayor de 100, se producirá un *buffer overflow* sobre el valor de retorno de la pila, con lo cual un atacante se puede hacer con el control de la aplicación.

Al igual que sucedía con el programa anterior, aquí el problema se encuentra en una mala llamada a la API, en este caso a la función `recv`. Al poner la capacidad de la variable 300 bytes más pequeño que el tamaño que le decimos a la función que `recv` que puede recibir, estamos propiciando que se puedan sobrescribir 300 bytes en la pila, lo cual, en la mayoría de los casos, producirá una sobrescritura del la dirección de retorno de la función y consecuentemente que el programa sea vulnerable a un ataque con cadenas ROP[4].

La solución a este problema es que el tamaño de la variable en la que se van a guardar los datos devueltos por la API sea igual o menor que el valor que se le pasa a esta. No tiene sentido hacer lo que ha implementado el programador en este trozo de código.

Este programa ya cuenta con una amplia historia de vulnerabi-

[4]*Return Oriented Programming*, un tipo de explotación del *buffer*

lidades como podemos observar en la web de exploit-db[5]. Muchas de ellas aunque aparecen referenciadas como diferentes ataques a servicios distintos dentro de la aplicación se producen por el problema que hemos encontrado. Exploits que han sido publicados como por ejemplo, `REST, PASV Buffer Overflow Exploit` o `ALLO Command Remote Buffer Overflow Vulnerability` se producen por la vulnerabilidad que ha sido descrita aquí, por lo que este programa no solo es vulnerable a las opciones del protocolo FTP que han sido descritos en este *exploit*, sino que es vulnerable a cualquier elemento que se le pase por internet y que sea procesado por la función `recv`. Esto hace que se puedan publicar muchos *exploits*, cada uno de ellos pudiendo explotar cada protocolo de FTP o incluso comandos que no existen dentro del protocolo FTP.

5.4.3 Ncplayer

En este caso la aplicación se trata de un programa para reproducir música en formato mp3 o de escuchar la radio directamente de internet. Trae la posibilidad de crear listas de reproducción de una manera muy simple y facilitando enormemente esta labor al usuario. Además esta herramienta cuenta con la posibilidad de hacer las operaciones mediante atajos de teclado.

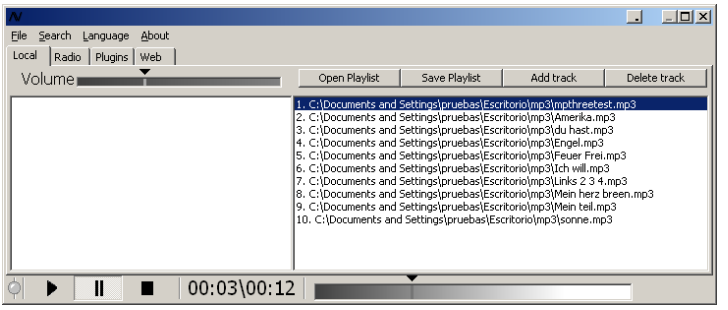

Figura 5.10: Captura de pantalla mostrando el programa ncPlayer.

El problema de esta aplicación viene por la lista de reproducción que no controla correctamente los limites de las rutas que se

[5]http://www.exploit-db.com/

le pasan al fichero. Si se le pasa una ruta suficientemente larga ésta puede provocar que se sobrescriba la dirección de retorno de la pila provocando un *buffer overflow*.

El error que tiene este programa es similar al que hemos visto tanto en el programa Simple Family Tree como en Isaac Software Contact, un mal control de la llamada al procedimiento `ReadFile`.

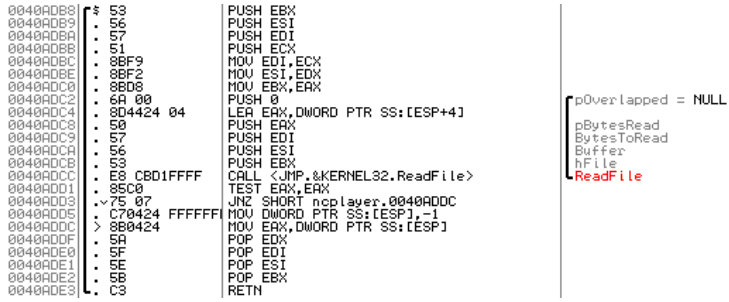

Figura 5.11: Captura de pantalla mostrando el trozo de código con error en ncPlayer.

En la Figura 5.11 podemos ver la función en la que se produce dicho error. En este caso podemos ver la función entera en la cual se realiza la llamada y podemos incluso adivinar cual es el código que escribió el desarrollador para esta función. La llamada a `ReadFile` se realiza con normalidad, pero en este caso se le pasa el tamaño del fichero que va a leer desde la llamada anterior a esta función. Aunque cuando se carga el fichero parece que funciona correctamente, en el momento que se intenta leer la lista de reproducción, esta produce un error que hace que se produzca un desbordamiento de pila.

En este caso existen varios problemas que han llevado a que se produzca este fallo. Por un lado no se controla correctamente el tamaño de la cadena cuando se va a leer el fichero, con lo cual es posible introducir una cadena muy larga en el fichero de la lista de reproducción y de esta forma ya conseguir que se pueda desbordar la memoria al poder sobrescribir páginas que se encuentren en zonas adyacentes a la memoria en la cual se va a guardar la lista de reproducción. Aunque es posible que este problema no se llegue a

producir porque no se introduzca una cadena tan larga como para llegar a sobrescribir una página diferente en la memoria, el no controlar este valor da paso al segundo error que contiene este programa.

Pero el error más importante ocurre cuando el usuario intenta reproducir el fichero que contiene una cadena extremadamente larga. En este caso tampoco se controla el tamaño que nos llega de la lectura del fichero (no se sabe si porque se da por hecho que viene controlado por otro sitio o simplemente por un descuido de los programadores), y ello propicia que aquí se produzca el desbordamiento del *buffer* si no se había producido anteriormente.

```
0041F0CC  . 8B40 04       MOV EAX,DWORD PTR DS:[EAX+4]
0041F0CF  . E8 E4BCFEFF   CALL ncplayer.0040ADB8
0041F0D4  . 83F8 FF       CMP EAX,-1
0041F0D7  .v75 02         JNZ SHORT ncplayer.0041F0DB
0041F0D9  . 33C0          XOR EAX,EAX
0041F0DB  > C3            RETN
```

Figura 5.12: Captura de pantalla mostrando la llamada a la función vulnerable.

Como hemos podido observar, todos estos errores parece que se están produciendo más por una mala programación por parte del programador, que por un fallo ocasionado por el compilador. Aunque este dato es difícil de comprobar sin la existencia del código fuente, el programador que haya realizado esta programación puede llegar fácilmente a la conclusión de como se produce este error mediante la comprobación del el camino que ha llevado a ejecutar que la fórmula devuelva un resultado.

5.5 Vulnerabilidades demostradas

Además de los *0-days* encontrados también se ha querido probar la eficacia de esta lógica con errores ya conocidos. Para ello se buscaron vulnerabilidades conocidas en páginas donde los investigadores suelen colgar sus *exploits*, como por ejemplo exploit-db[6] (de Offesive Security) o inj3ct0r[7].

[6] https://www.exploit-db.com/
[7] http://es.1337day.com/

En ciertas aplicaciones se buscaron los trozos de código que habían provocado errores, creando una serie de casos de prueba para intentar encontrar dicha vulnerabilidad, y se comprobaba que dichas fórmulas eran capaces de encontrar la vulnerabilidad por la cual se había llegado a crear el *exploit* que había sido publicado.

Incluso se ha dado un caso de un *exploit* reportado como que solo era una denegación de servicio, que nuestro sistema ha demostrado ser un *buffer overflow* con el cual se podría tomar el control de la máquina que esté ejecutando ese programa.

Siguiendo las mismas normas seguidas para buscar 0-days hemos demostrado los errores que se presentan en las siguientes subsecciones.

5.5.1 **Jaangle**

Esta aplicación es un reproductor y organizador de música. Jaangle es capaz de reproducir muchos tipos de formatos, los cuales pueden ser añadidos uno a uno o directamente añadiendo una carpeta con todos los ficheros de música que un usuario tiene en el ordenador. Además de las típicas características que tienen todos los reproductores de música, contiene otras particularidades como la visualización de las portadas de los discos, *skins*, radio, renombrar ficheros, y muchas más.

Para este reproductor fue encontrada una vulnerabilidad de denegación de servicio (conocido ampliamente como DoS, por su nombre en inglés *Denial of Service*) por Hadji Samir[8] en Diciembre de 2014. Una vulnerabilidad de denegación de servicio consigue que el programa se cierre sin que el usuario pueda hacer nada al respecto.

En este caso la vulnerabilidad se producía cuando un usuario introducía un nombre de fichero muy largo en una lista de reproducción (similar a como ocurría en ncplayer). En este caso Samir comprobó que al introducir la cadena larga en el reproductor y depurar el proceso el programa salía de una forma más o menos con-

[8] http://www.exploit-db.com/exploits/35532/

trolada, sin que pareciese que la pila era sobrescrita por la cadena que se introducía. Por esta razón se llegó a la conclusión de que se trataba de un error de denegación de servicio y que no se podía llegar a explotar como si se hubiese sobrescrito el retorno de la función creando un *buffer overflow*.

Después de nuestros análisis hemos descubierto que la vulnerabilidad creada por Samir sí se trata en realidad de una vulnerabilidad de *buffer overflow* encubierta. Después de ver como se ejecutaba el *exploit* pudimos observar que gran parte del error se encontraba en una función de denominadas peligrosas `wsprintf`, con lo cual decidimos crear una fórmula en la cual se mostrase el camino que llevara a la función `ReadFile` y además a `wsprintf`.

Con ello pudimos ver que uno de los caminos que pasaban por la función `wsprintf` tiene también una llamada a la API de Windows `isDebbugerPresent`. Con lo cual llegamos a la conclusión que si el investigador estaba depurando el proceso esta función sería llamada y terminaría el programa de manera controlada. Esto aunque sí que es una vulnerabilidad de denegación de servicio es algo que el programador de la aplicación tiene controlado y no se puede llegar a explotar.

Pero, ¿qué pasaría si se ejecutase la aplicación sin un depurador, es decir, que la función `isDebuggerPresent` no llegase a saltar? Este caso puede llegar a suceder, porque un camino de la salida de nuestro algoritmo devolvió esa opción, es decir, un camino que no llama a esa función y el programa sigue funcionando. En este caso, la información que se le ha pasado con la cadena muy larga llega a desbordar la dirección de retorno produciendo una vulnerabilidad de *buffer overflow*.

Aunque la trampa que preparó el programador ha servido para confundir al investigador de vulnerabilidades no ha conseguido engañar a nuestro programa en el que se ha visto correctamente reflejado el camino a seguir para llegar a conseguir esta vulnerabilidad no documentada hasta el momento.

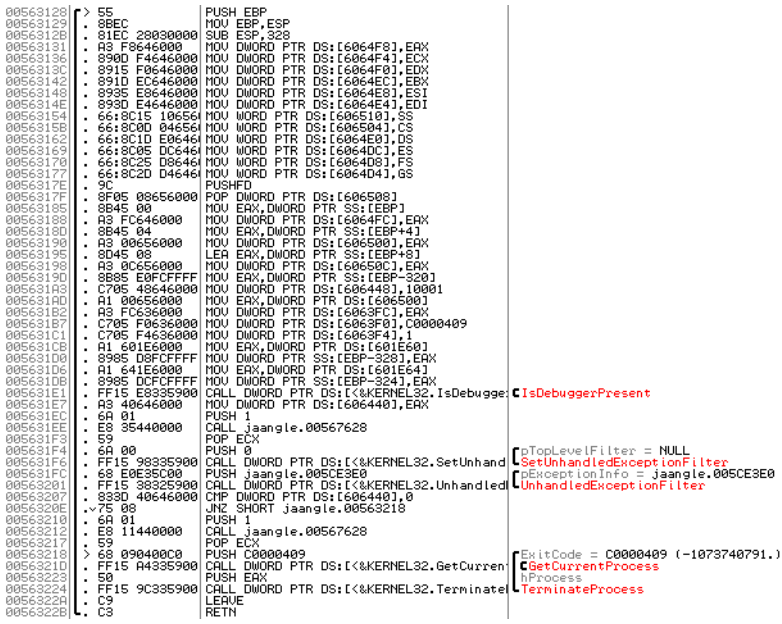

Figura 5.13: Funcion que comprueba si está se está depurando el proceso.

5.5.2 **Free Mp3 CD Ripper**

Esta aplicación se utiliza principalmente para pasar CDs de audio a formatos digitales, wav, mp3, ogg, entre otros muchos. Además de esto también posee la capacidad de crear imágenes de CDs para poder ser grabadas más adelante en otro CD, es decir, realizar copias de seguridad de un CD.

La vulnerabilidad encontrada en este caso se trata de un *buffer overflow* que permite asumir el control del ordenador atacado. Este exploit[9] fue creado por el grupo *Tunisian Cyber* en marzo del año 2015, y en la demostración del *exploit* podemos ver como el atacante puede llegar a ejecutar la calculadora de Windows, prueba de concepto de que cualquier cosa puede ser ejecutada dentro del ordenador atacado.

La vulnerabilidad se producía en una al crear un fichero .wav

[9]https://www.exploit-db.com/exploits/36465/

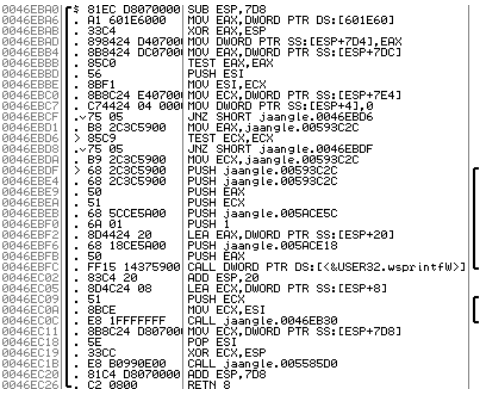

Figura 5.14: Función donde se realiza la llamada a `wsprintf` y la función donde se comprueba el depurador.

mal formado. El problema viene dado por una mala validación del fichero wav antes de leerlo, ya que el programador ni siquiera comprueba que la cabecera del fichero corresponda a una cabecera wav válida. Por esta razón el programa podía leer una estructura inválida que acabase sobrescribiendo el puntero de retorno en la pila.

El programa comprueba cada estructura del fichero wav mediante un `switch` que va llamando a las distintas secciones del fichero (riff, wave, fmt, fact, data,) tal y como puede observarse en la Figura 5.15.

Figura 5.15: Imagen que muestra como controla con un switch las distintas partes del fichero wav.

Dentro de las diferentes secciones va realizando llamadas a una función que contiene una `ReadFile` insegura, y no controla realmente los límites de cada sección. Este detalle hace posible que una sección pueda llegar a albergar una cadena sumamente larga que pise el valor de retorno de la pila. Como hemos visto anteriormente, si esto sucede un atacante puede tomar el controlo del programa y ejecutar cualquier otro proceso dentro de la máquina en la cual se está ejecutando esta aplicación.

Figura 5.16: Función donde se realiza la llamada a `wsprintf` y la función donde se comprueba el depurador.

Este caso ha sido bastante difícil de encontrar por el problema de las llamadas dinámicas. Como hemos visto en la Figura 5.15, el `switch` hace que se vayan leyendo las distintas partes del binario, pero en este caso no llama directamente a ninguna parte del binario. La llamada la hace con un «`CALL DWORD PTR SS:[EBP+C]`», y este detalle no puede ser seguido por IDA Pro mediante un análisis estático. Por esta razón en nuestros análisis no se mostraba una ruta completa hasta el binario sino un pequeño trozo que se veía una llamada a `ReadFile` potencialmente peligrosa.

Por esta razón hubo que hacer un posterior estudio del binario, mediante otras herramientas que analizan el programa en tiempo de ejecución, para encontrar cómo se llegaba hasta el código peligroso, y descubrir si era verdaderamente posible explotar esa falta de comprobaciones al realizar la llamada a `ReadFile`.

5.5.3 **Palringo**

Este cliente de mensajería soporta la comunicación con varios protocolos, logrando de esta forma el uso de los chats como Hangouts, Facebook, Jabber, Yahoo messenger, entre otros.

La vulnerabilidad que se ha encontrado en este programa es más difícil de explotar que en los casos anteriores, pero permitía un `buffer overflow` que podía hacerse con el control del ordenador atacado[10]. En este caso para conseguir encontrar explotar la vulnerabilidad se precisa que el programa pase a través de un *proxy*[11]. Éste está especialmente diseñado por un atacante para enviar una trama `TCP` maliciosa generando el *buffer overflow* en el programa.

```
005287E0 r$  B8 04100000  MOV EAX,1004
005287E5 .   E8 06F30800  CALL palringo.005B7AF0
005287EA .   A1 50526200  MOV EAX,DWORD PTR DS:[625250]
005287EF .   33C4         XOR EAX,ESP
005287F1 .   898424 00100 MOV DWORD PTR SS:[ESP+1000],EAX
005287F8 .   8B4E 04      MOV ECX,DWORD PTR DS:[ESI+4]
005287FB .   6A 00        PUSH 0                                   ┌Flags = 0
005287FD .   68 00100000  PUSH 1000                                │BufSize = 1000 (4096.)
00528802 .   8D4424 08    LEA EAX,DWORD PTR SS:[ESP+8]             │
00528806 .   50           PUSH EAX                                 │Buffer
00528807 .   51           PUSH ECX                                 │Socket
00528808 .   FF15 48865D0 CALL DWORD PTR DS:[<&WS2_32.#16>]        └recv
0052880E .   85C0         TEST EAX,EAX
00528810 .   7F 35        JG SHORT palringo.00528847
00528812 .   8B56 04      MOV EDX,DWORD PTR DS:[ESI+4]
00528815 .   52           PUSH EDX                                 ┌Socket
00528816 .   C646 08 00   MOV BYTE PTR DS:[ESI+8],0                └closesocket
0052881A .   FF15 3C865D0 CALL DWORD PTR DS:[<&WS2_32.#3>]
00528820 .   8B06         MOV EAX,DWORD PTR DS:[ESI]
00528822 .   8B50 08      MOV EDX,DWORD PTR DS:[EAX+8]
00528825 .   8BCE         MOV ECX,ESI
00528827 .   C746 04 0000 MOV DWORD PTR DS:[ESI+4],0
0052882E .   FFD2         CALL EDX
00528830 .   32C0         XOR AL,AL
00528832 .   8B8C24 00100 MOV ECX,DWORD PTR SS:[ESP+1000]
00528839 .   33CC         XOR ECX,ESP
0052883B .   E8 E32E0700  CALL palringo.0059B723
00528840 .   81C4 0410000 ADD ESP,1004
00528846 .   C3           RETN
00528847 >   50           PUSH EAX
00528848 .   8D4424 04    LEA EAX,DWORD PTR SS:[ESP+4]
0052884C .   50           PUSH EAX
0052884D .   8D4E 0C      LEA ECX,DWORD PTR DS:[ESI+C]
00528850 .   E8 BBE7EEFF  CALL palringo.00417010
00528855 .   8B16         MOV EDX,DWORD PTR DS:[ESI]
00528857 .   8B42 0C      MOV EAX,DWORD PTR DS:[EDX+C]
0052885A .   8BCE         MOV ECX,ESI
0052885C .   FFD0         CALL EAX
0052885E .   8B8C24 00100 MOV ECX,DWORD PTR SS:[ESP+1000]
00528865 .   33CC         XOR ECX,ESP
00528867 .   B0 01        MOV AL,1
00528869 .   E8 B52E0700  CALL palringo.0059B723
0052886E .   81C4 0410000 ADD ESP,1004
00528874 L.  C3           RETN
```

Figura 5.17: Función donde se realiza la llamada a recv.

En la Figura 5.17 vemos la función en la cual se produce la vulnerabilidad. Justo después de la llamada a la función `recv` se comprueba que se haya recibido algo (es decir, que el tamaño no sea cero), pero no se llega a controlar nunca el tamaño máximo de la cadena recibida. Al no realizar esta comprobación es posible que el

[10] https://www.exploit-db.com/exploits/35741/

[11] Servidor intermedio para conectarse a internet

atacante introduzca una cadena muy larga pudiendo sobrescribir la dirección de retorno de la función y provocar un *buffer overflow*.

5.6 Conclusiones

En este capítulo hemos realizado un estudio de la expresividad y complejidad de POCTPL, el cual ha resultado ser muy similar a la lógica CTL, encontrándonos también en el problema de la explosión de estados.

También hemos observado que nuestro algoritmo ha sido capaz de detectar errores no conocidos en el momento de su revelación. A su vez hemos detectado errores que ya habían sido reportados para demostrar que nuestra herramienta encuentra correctamente dichos errores.

Además, hemos detectado nuevas funciones peligrosas que no estaban etiquetadas dentro de este grupo por Microsoft. Las funciones recv y ReadFile son parte de la API de Windows y si se le llama con parámetros incorrectos sin comprobar el valor que devuelven pueden terminar produciendo vulnerabilidades de desbordamiento del buffer.

«¿Y a donde irá la recién nacida desde aquí? La red es vasta e infinita.»

Mayor Motoko Kusanagi

Capítulo

6

Conclusiones

Son muchos los problemas que aun quedan por resolver dentro del campo del *model checking*, ero aun son más los retos que hay que superar para la detección de errores dentro del código fuente. En este contexto, esta tesis se ha centrado en parte de esta problemática, la búsqueda de vulnerabilidades en el código ensamblador generado por los compiladores. Aunque, como hemos podido ver en capítulos anteriores, se han realizado trabajos previos en este área, el camino que queda por realizar todavía es largo y contiene múltiples obstáculos.

En este capítulo se detallan las principales contribuciones que han sido aportadas por la elaboración de la tesis, así como el trabajo futuro que es posible realizar mediante los resultados obtenidos. A su vez, se detallan las principales limitaciones que han surgido durante la elaboración de todo el proceso de investigación y de las cuales podrían surgir nuevas lineas de investigación. Concluiremos con unas consideraciones finales sobre todo el estudio y el código realizado.

6.1 Contribuciones

Las principales contribuciones que se han producido durante el desarrollo de esta investigación han sido las siguientes:

1. La creación de un nuevo algoritmo, llamado POCTPL, que cambia la salida booleana de la lógica CTPL a una que muestra el árbol de ejecución que conduce hasta el código indicado.

2. Se ha creado un lenguaje de representación intermedio del código ensamblador de Intel x86, llamado POCTPL-IR, el cual obtiene un mejor tratamiento para ser interpretado por cualquier lógica formal.

3. Hemos programado código en lenguaje Coq representando las propiedades de POCTPL-IR.

4. Se han creado y verificado teoremas que formalizan el código generado con Coq y demuestran que éste no contiene errores.

5. A su vez se ha desarrollado un parseador de código Intel x86 para la posterior transformación en POCTPL-IR y en una estructura Kripke.

6. Un analizador léxico y sintáctico ha sido creado para validar que las fórmulas que introducen los usuarios de POCTPL sean correctas.

7. Se han encontrado vulnerabilidades no descubiertas hasta el momento, y de otras ya comprobadas para validar la lógica creada.

8. Y finalmente se han detectado de nuevas funciones peligrosas dentro de la API de Windows que son capaces de cometer errores de *buffer overflow* si no están correctamente programadas.

6.2 Limitaciones

Como sucede con todas las lógicas y los *model checkers*, POCTPL contiene una serie de limitaciones que pasamos a detallar a continuación.

6.2.1 Problema de la explosión de estados

Entre las limitaciones con las que cuenta POCTPL se encuentran las mismas que trae consigo el *model checking* en general. Y por esta razón, el principal contratiempo de POCTL es el problema de la explosión de estados. Cuanto más grande es el tamaño del binario, existen más posibilidades de que la complejidad del análisis del flujo de ejecución también se haya incrementado. Esto, en muchos casos, puede hacer que el análisis del binario se extienda mucho en el tiempo, haciendo que sea inabordable en términos prácticos. Otro hecho que puede hacer que la complejidad del programa aumente, es que existan múltiples puntos de entrada dentro del mismo binario haciendo que surjan nuevos caminos para analizar.

6.2.2 Fórmulas complejas

Otro problema que hace que el tiempo de análisis aumente es la complejidad de las fórmulas. Esto se debe principalmente por la forma en la que es realizado el análisis de las fórmulas de forma recursiva por parte de Coq. Cuanto más larga sea la fórmula introducida más operaciones tendrá que hacer dentro de la estructura Kripke. Así como el problema de los árboles complejos es posible de resolver mediante alguna de las soluciones propuestas por otros autores, no es tan sencillo simplificar las fórmulas que se introducen en el sistema.

6.2.3 Lenguajes de programación

Aunque las principales aplicaciones del software están programadas por C/C++, existen otros muchos productos que son desarrolladas por otros lenguajes de programación. Nuestra lógica se defiende bastante bien ante programas realizados con lenguajes compilador, no está preparado para depurar lenguajes interpretados, como Java o Python.

En la sección de trabajo futuro veremos alguna posibilidad para subsanar esta limitación y, de esta forma, hacer que nuestra lógica sea capaz de analizar muchos más lenguajes de programación.

6.3 Trabajo futuro

Muchos de los trabajos que se pueden utilizar para continuar con esta investigación son para intentar superar las limitaciones que hemos visto que se han dado.

Vamos a ver cada posible linea de investigación futura con más detalle:

6.3.1 Lenguajes intermedios

Una de las limitaciones de esta tesis es que el lenguaje de programación está limitado a C/C++. Este hecho es posible resolverlo de varias formas, pero la más aconsejable es la utilización de un lenguaje intermedio.

Existen varios lenguajes intermedios que podrían aportar mucho a nuestra plataforma, el primero de ellos es LLVM-IR. Esta representación intermedia es la utilizada por el famoso compilador LLVM[1], que es capaz de crear binario partiendo de varios lenguajes de programación (como por ejemplo Haskell, Lisp, ActionScript, Go, Haskell, Java bytecode, Julia, Ruby, Swift, Python, entre otros). Para utilizar estos lenguajes LLVM ha creado varios front-ends que transforman el código del lenguaje elegido a LLVM-IR, mediante esto es

[1] http://llvm.org/

posible compilar cualquier lenguaje debido a que siempre las optimizaciones de código se realizan sobre la representación intermedia. Adoptando la de LLVM sería posible hacer que los front-ends de LLVM se pudieran utilizar para aplicar POCTPL y verificar muchos más binarios.

Otro lenguaje intermedio preparado para simular el código ensamblador es Vine-IL, realizado para el analizador estático de Bitblaze, Vine[2]. Esta representación intermedia realiza varias comprobaciones mejorando la del lenguaje ensamblador. A su vez utiliza técnicas para reducir los flujos de ejecución, como la eliminación de código muerto, propagación de variables, incluso crear trozos de gráfico que incluyen las instrucciones relevantes para el análisis.

Adaptar POCTPL-IR o Vine-IL supone la implementación de este lenguaje intermedio en Coq, incluyendo las verificaciones necesarias para la verificación de que este código se vaya a ejecutar sin errores. Esto supondría la reimplementación de la parte de Coq en la cual esta implementada POCTPL-IR y el estudio del nuevo lenguaje intermedio lo cual supone una nueva vía de investigación.

6.3.2 **Reducción de la gráfica**

Uno de los problemas más grandes a los que nos hemos enfrentado con la realización de esta tesis es el gran tamaño de la gráfica resultante del análisis de ficheros. Existen teorías que hacen que estas gráficas puedan ser más eficientes, como hemos visto en el estado del arte. Aun así es difícil hacer las gráficas más reducidas en nuestro caso, ya que la reducción de estados puede llevar consigo que se pierdan vulnerabilidades en el sistema (suceso que se ha demostrado en diversas ocasiones). En cualquier caso es necesario que se mejoren las gráficas para poder hacer más rápido el análisis de cualquier binario.

La mejor técnica que puede evitar la redundancia de código puede ser la de separar los prefijos y sufijos repetidos para combinarlos, con lo cual se reduce en gran cantidad en número de estados a ana-

[2]http://bitblaze.cs.berkeley.edu/vine.html

lizar.

La priorización de pruebas también es un campo que puede ser aplicado correctamente en nuestro caso. De esta forma, aunque el conjunto de pruebas no ha sido reducido de forma considerable, los resultados pueden ser obtenidos en menos tiempo.

6.3.3 **Plugins**

Otra idea con la que desarrollar la plataforma es la inclusión de *plugins* para diversos programas que facilitarían el uso para los usuarios finales.

La realización de *plugins* para compiladores es uno de los aspectos que pueden atraer a potenciales usuarios a utilizar nuestra plataforma. El análisis de los binarios después de ser compilados es uno de los objetivos importantes para la detección de vulnerabilidades por parte de POCTPL. Por esta razón sería posible crear un *plugin* que verifique la creación de binarios justo de entregarlos al usuario final, mostrando posibles errores para que el programa pueda solucionarlos lo más rápido posible.

Otro *plugin* en el cual la plataforma POCPL encajaría perfectamente sería uno realizado para IDA Pro. Con este *plugin* se aprovecharían varios aspectos. Al ser parte de esta herramienta se pueden aprovechar muchas más características de esta aplicación y no solo la parte de análisis. Sería posible visualizar los caminos de manera completamente gráfica mediante su interfaz. Otra ventaja de utilizar IDA Pro es que su lenguaje de *script* para realizar *plugins* es Python, que es con el cual hemos desarrollado gran parte de la plataforma. Por está razón es más fácil portarlo a un *plugin* de este programa que a cualquier otra plataforma.

6.3.4 **Lógica difusa**

La lógica que hemos implementado puede ser llevada a otros entornos para posibles aplicaciones más funcionales en la práctica.

En este caso, la lógica difusa puede ser aplicada a la búsqueda

de vulnerabilidades para saber el grado de peligrosidad que puede tener determinado código. Con lo cual ya no hace falta definir exactamente el código vulnerable, sino sería la misma aplicación la que diría lo peligroso que puede llegar a ser determinado código.

Este cambio supondría reimplementar toda la parte relacionada con la programación del *model checker*, es decir, la parte de Coq. Si se realiza este cambio, la parte de los teoremas que demostraban que el código era correcto para POCTPL debería de ser reexaminado, pero en muchos casos podría seguir validando casos, ya que muchas partes del código sería igual.

6.3.5 **Fórmulas que impliquen vulnerabilidades**

Aunque durante el desarrollo de esta tesis se han desarrollado fórmulas para la detección de vulnerabilidades mediante el estudio de estas, en este terreno todavía queda mucho trabajo por realizar. La creación de más fórmulas es imprescindible para la correcta detección de más tipos de errores, como los descritos en la introducción de esta tesis.

Dentro de este mismo contexto sería posible desarrollar una pequeña comunidad de personas que subirían fórmulas para encontrar diferentes vulnerabilidades dentro del código. De esta forma es posible, rápidamente, que se vayan detectando más vulnerabilidades, haciendo de esta herramienta una parte importante de la comunidad dedicada a la búsqueda de vulnerabilidades y produciendo así código más seguro.

6.4 **Preguntas de investigación**

Las preguntas y objetivos que se habían formulado al principio de esta tesis han sido respondidas durante el desarrollo de la misma. A continuación vamos a ver como se han cumplido cada uno de estos puntos.

6.4.1 **Hipótesis y objetivo principal**

La hipótesis con la que se inició la investigación para esta tesis fue la siguiente:

> *«Existe un método de representación de la memoria que es capaz de detectar errores conocidos, tanto en el heap como en la pila»*

Ésta ha quedado perfectamente validada a través del desarrollo del objetivo principal y los objetivos específicos que también se han ido cumpliendo. El método de representación que se ha creado para la creación se basó en la lógica booleana y ha sido capaz de encontrar los errores especificados mediante el uso de la API de Windows, a través del cual es posible controlar las llamadas para saber si se están realizando correctamente o si los valores que devuelven están validados de forma correcta para que no puedan incurrir en cualquier tipo de error que ponga en peligro el sistema.

Para llegar a validar la hipótesis de partida ha sido importante realizar el objetivo principal que fue establecido al principio de esta investigación:

> *«Diseñar e implementar un algoritmo verificado capaz de detectar los errores en los binarios antes de su ejecución.»*

Para cumplir este objetivo se ha creado la lógica POCTPL capaz de buscar vulnerabilidades analizando el código ensamblador sin necesidad de ejecutar el código. Para validar este código, y que no contuviera errores en sí mismo se ha realizado la implementación con Coq, que permite realizar la verificación mediante el lenguaje de cálculo de construcciones inductivas (suma del cálculo lambda y la lógica de orden superior).

6.4.2 **Objetivos específicos**

Para poder realizar satisfactoriamente el objetivo principal es importante desgranarlo en varios objetivos más pequeños que hagan

más fácil de cumplir la tarea de cumplir el principal.

Los objetivos que se marcaron al principio de la investigación fueron los siguientes:

> **Objetivo 1** *Diseñar e implementar un algoritmo de capaz de determinar cuál ha sido el camino que ha tomado una ejecución hasta encontrar un error, que actúe como base para el siguiente objetivo.*

Este objetivo se ha satisfecho con la lógica POCTPL la cual devuelve el camino que cumple la fórmula introducida por el usuario en el sistema, y que es validada por alguno de los caminos que se encuentran en la estructura Kripke que representa el binario.

> **Objetivo 2** *Diseñar e implementar un método verificado, que no genere errores en su ejecución capaz de detectar los principales errores producido en la programación de software.*

Mediante la implementación de nuestro algoritmo en Coq, y la validación mediante teoremas que verificaban el sistema programado ha sido posible llegar conseguir un programa que no puede producir errores, cumpliendo de esta manera el objetivo.

> **Objetivo 3** *Diseñar e implementar un método capaz de detectar y evitar los errores en las fórmulas de nuestro algoritmo que han sido introducidas por el usuario.*

La consecución de este objetivo se ha llevado a cabo mediante la programación de un analizador léxico y sintáctico que funciona de forma similar a como lo realizan los compiladores. Por esta razón se han utilizado las herramientas que utilizan éstos (lex y yacc), pero en su versión para Python.

> **Objetivo 4** *Diseñar e implementar un método capaz de parsear el código ensamblador y adaptarlo al lenguaje del algoritmo.*

Para la realización de este objetivo hemos utilizado varias librerías de Python que además de ser utilizadas para hacer la traducción del lenguaje ensamblador x86 a POCPTL-IR, transformará el código en una estructura Kripke entendible por el sistema.

Objetivo 5 *Demostrar la eficacia de los métodos anteriores con programas reales.*

En el momento de probar la eficacia de nuestro algoritmo lo hemos divididos en dos partes: i) búsqueda de nuevas vulnerabilidades en programas antiguos, y ii) búsqueda de vulnerabilidades conocidas y demostradas. En ambos casos hemos conseguido demostrar que la lógica es capaz de encontrar errores y devolver la ruta que lleva hasta ellos.

6.5 Análisis de los resultados obtenidos

Como hemos visto, se han encontrado varias vulnerabilidades nuevas. También se ha demostrado que POCTPL encuentra correctamente errores ya detectados e incluso explotados por otros investigadores. Esto demuestra que la herramienta produce buenos resultados en el campo de la investigación del fallos de seguridad.

Los errores que se han detectado mediante esta plataforma han sido varios pero con la mejora de las fórmulas pueden ser muchos más, y de diversos tipos. Aunque esta tarea lleva una gran cantidad de tiempo, es imprescindible para poder realizar unas fórmulas correctamente.

El mayor problema que se ha observado durante las pruebas sobre binarios ha sido la cantidad de tiempo necesario para realizar pruebas sobre determinados binarios. Los binarios muy grandes implican gran cantidad de estados y en la mayoría de los casos, gran cantidad de caminos, con lo que el tiempo necesario para poder realizar un análisis completo es muy largo. Aún en el caso que el tamaño del binario sea pequeño, es posible que la complejidad de caminos hiciera que el tiempo empleado para poder comprobarlos todos fuese también muy elevado.

Durante el transcurso de esta investigación se ha detectado que el origen de muchas vulnerabilidades es similar. Aunque Microsoft han marcado unas funciones como peligrosas hemos detectado dos llamadas a la API de Windows que sin estar en su lista tienen un riesgo muy alto.

Tanto la función `recv` como `readFile`, son funciones muy peligrosas, capaces de albergar fácilmente vulnerabilidades de desbordamiento de *buffer*, debido a la carencia de un método de control del tamaño de lo que reciben y leen, respectivamente.

Mediante el estudio de los resultados es posible detectar más funciones peligrosas dentro de la API de Windows que todavía no están catalogadas. PCTPL se presenta como una herramienta poderosa en este campo, que puede ayudar a mejorar el código fuente e incluso detectar fallos en las propias librerías de Windows.

6.6 Consideraciones finales

La lucha contra las vulnerabilidades está avanzando poco a poco con el paso de los años, haciendo programas más seguros y mayores protecciones que nos ayudan a estar más seguro contra *malware* o atacantes que intentan robar datos o utilizar nuestros ordenadores para otros fines.

Los precios por descubrir vulnerabilidades en software crítico cada día suben más, y las propias empresas abren programas para premiar a los investigadores que les indican partes vulnerables de su código. Incluso se están llegando a formar redes sociales de buscadores de errores, donde los investigadores presumen de las vulnerabilidades que has encontrado, haciendo esto más atractivo y consiguiendo que esta labor sea una especie de juego.

Cada día existen más mafias e incluso gobiernos que utilizan estas técnicas para sus fines, haciendo una *ciberguerra* de la cual gran parte de la población ni si quiera se da cuenta de su existencia, aunque gran parte de ellos participan involuntariamente. Las técnicas que utiliza todo tipo de *malware* se aprovechan de estas vulnerabilidades para atacar objetivos que cada día son más específicos,

como centrales nucleares en países determinados, como ya realizó el gusano Stuxnet[3].

La importancia de realizar código seguro ha sido la gran preocupación de las grandes empresas del software como lo han sido Google y Microsoft, realizando aportaciones periódicas en este campo. En esta tesis doctoral se ha desarrollado un sistema para que el código sea más seguro, haciendo que cada vez llegue menos software vulnerable a los usuario finales, que no tienen conocimientos de todo el peligro que conlleva tener una aplicación mal programada. Además se ha desarrollado un nuevo algoritmo para poder definir patrones dentro del lenguaje ensamblador, algo que puede ser llevado a otros campos más allá de la búsqueda de vulnerabilidades.

En este nuevo mundo que se está abriendo delante de nuestros ojos, las *ciber-armas* son la principal fuente de ataques. Estas armas en la mayoría de las ocasiones atacan los fallos en el software, y es en este punto donde nuestra tesis entra en acción. Porque como dijo Dan Geer(CTO de In-Q-Tel, firma creada por la CIA) en la conferencia Black Hat USA del 2015:

«No necesitamos conocer que armas tienen nuestros adversarios si tenemos algo como un inventario completo de todas las vulnerabilidades del mundo y las hemos compartido con todos los proveedores de software afectados».

[3]La que se ha considerado la primera *ciber-arma* de la historia.

Apéndices

«El éxito llega a quienes están dispuestos a trabajar un poco más duro que el resto.»

Og Mandino (1923–1996)

Apéndice

A

Verificación de las formulas

Un punto muy importante dentro de nuestro sistema es asegurarse que cuando el usuario introduce una fórmula no cometa errores en su redacción. Para realizar esta labor, es necesario hacer un análisis léxico y sintáctico de la fórmula antes de seguir con el proceso de verificación.

Aunque en un principio del desarrollo de la herramienta creímos que con un análisis realizado con expresiones regulares sería suficiente, según fuimos realizando varias pruebas llegamos a la conclusión que no es posible realizarlo de forma eficaz. Esto es debido a la complejidad que pueden llegar a tener las fórmulas, que es difícil controlar mediante expresiones regulares, puede llegar a permitir código erróneo.

Las mejores herramientas que existen para la realización de análisis léxicos y sintáctico son Lex y Yacc, y debido a que el programa estaba realizado utilizando Python, la mejor alternativa para hacer

esto era hacerlo con la herramienta `ply`[1]. Estas librerías para Python fueron creadas por David Beazley, y en ellas se implementan tanto la parte de Lex, como la parte de Yacc en un formato programable.

Lex es un creador de analizadores léxicos. La función principal de un analizador léxico es tomar un flujo de caracteres entrada y devolver un flujo de tokens como salida. Yacc, Yet Another Compilers Compiler, nos permitirá crear un programa que tome un flujo de tokens como entrada y reconozca a partir de ellos un lenguaje (en nuestro caso POCTPL).

La definición de los tokens es un aspecto muy importante de cualquier lenguaje, y de ella depende que el análisis sea correcto y corrija correctamente los errores que un usuario pueda incurrir al introducir la fórmula. En nuestro caso hemos definido la siguiente gramática:

ApiWindows	→	CLSIDFromProgID | ··· |VerQueryValueW
registers	→	eax | ··· |ah
hexa	→	0 | 1 | ··· | 9 | A | ··· |F
memory	→	0x hexa hexa hexa hexa hexa hexa hexa hexa
opcode	→	mov register register | mov register memory mov memory register | mov memory memory push memory | push register | call ApiWindows pop register |pop memory |cmp register register cmp register memory |cmp memory memory jz register |jz memory | lea register register lea register memory | lea memory memory test register register | test register memory test memory memory | jmp memory | jmp register add register register | add register memory add memory memory | jnz memory | jnz register retn | xor register register | xor register memory xor memory memory | and register register and register memory | and memory memory
LogicFormula	→	EX LogicFormula | AX LogicFormula EF LogicFormula | AF LogicFormula | EG LogicFormula AG LogicFormula | E LogicFormula U opcode A LogicFormula U opcode | opcode
Quantification	→	Exists register LogicFormula | Exists memory LogicFormula Exists opcode LogicFormula | ForAll register ForAll memory | ForAll opcode

En esta gramática podemos ver varias cosas que nos pueden lla-

[1] http://www.dabeaz.com/ply/

mar la atención. Una de ellas es la definición de ApiWindows, en la que se incluyen todas las llamadas a las funciones que se encuentran en la API de windows. Esto se ha especificado de esta forma para que un usuario pueda realizar fórmulas que contengan llamadas a estas funciones, y por ello, es preciso tenerlas todas definidas dentro de nuestra gramática.

Otro detalle que podemos observar en esta gramática, es que también incluimos información de los registros que se encuentran en el microprocesador. Al igual que en el caso anterior, el usuario puede introducir los registros del procesador al realizar fórmulas, y la gramática controlará que no introduzca ninguno que no exista dentro del procesadores modernos. En este caso se introducen todos los registros posibles que tiene, tanto en las versiones de 32, 16 y 8 bits.

En el caso de los códigos operacionales, hemos incluido los más importantes para la búsqueda de virus incluidos en el estudio del Christodorescu [CJ06]. Al hacer esto hemos realizado dos cosas: i) reducir la complejidad del programa que va a parsear el código ensamblador; ii) hacer la gramática más sencilla, lo cual hace que la ejecución sea más rápida.

El resultado final hace que el lenguaje sea rígido, pero proporciona una forma segura de realizar las fórmulas para cualquier usuario no experto, ya que el sistema es capaz de corregir cualquier error que pueda surgir ya sea en el léxico como en su sintaxis.

«La única forma de hacer un gran trabajo es amar lo que haces. Si aún no lo has encontrado, sigue buscando.»

Steve Jobs (1955–2011)

Apéndice

B

Instalación de la plataforma

La plataforma que ha surgido de la investigación para llevar a cabo esta tesis se encuentra albergada dentro de la página que contiene repositorios de software, Bitbucket[1]. Mediante la creación de este repositorio el usuario siempre puede actualizar su software de la misma forma que se va progresando en el desarrollo de la herramienta.

El hecho de alojar nuestro código en esta plataforma cumple dos propósitos principales. El primero es servir de copia de seguridad y de repositorio para el código según vaya siendo desarrollando y el segundo, es el de servir de escaparate para que otros programadores se interesen por el proyecto aportando código, ya sea en formato de plugins como programando la propia herramienta.

En este apéndice se va a explicar como es el proceso descarga, configuración y de instalación para que la plataforma que hemos creado pueda funcionar en cualquier ordenador.

[1] https://bitbucket.org/aszy/poctpl

2.1 Programas necesarios

Nuestra plataforma necesita de algunas aplicaciones externas para desempeñar labores que nuestra herramienta no realizaba por si sola.

La primera herramienta que están dentro de nuestro framework es IDA Pro en su versión gratuita[2]. Esta herramienta se utiliza para la extracción del código ensamblador del binario que se va a examinar.

La tremenda complejidad del análisis de ensamblador que contiene un binario hace que sea necesario utilizar una herramienta externa para poder realizar esta labor. Aunque en el mercado actual existen varias herramientas(como por ejemplo Win32Disam[3] para Windows o Hopper[4] para Linux y Mac OS X) que realizan esta tarea IDA Pro está considerada la que mejor realiza este proceso actualmente. Además con su herramienta *idag* es posible realizar desensamblado directamente desde linea de comandos, razón principal por la que nos decidimos por esta herramienta.

La parte menos delicada de nuestra plataforma ha sido desarrollada con Python 2.6[5], y la parte más delicada corre mediante OCaml[6].

Además de estos programas principales también es necesaria la instalación de la libreria de Python que nos permite el análisis léxico y sintáctico de las fórmulas, Ply[7].

2.2 Ejecución

Una vez instaladas todas la herramientas necesarias para su correcto funcionamiento, es posible ejecutar nuestra plataforma para el

[2] http://out7.hex-rays.com/files/idafree50.exe

[3] http://www.angelfire.com/dc/vashala/hacks%20pages/hack_programs/Win32dasm.html

[4] http://www.hopperapp.com/

[5] https://www.python.org/ftp/python/2.6/python-2.6.msi

[6] http://gallium.inria.fr/~protzenk/caml-installer/ocaml-4.01.0-i686-mingw64-installer3.exe

[7] http://www.dabeaz.com/ply/ply-3.4.tar.gz

análisis del binario.

El ejecutable de toda la aplicación es el fichero *poctpl*. Esta aplicación solo admite un parámetro, la ruta del binario que se va a analizar. Por ello la única forma de llamar al programa es la que vemos en la Figura B.1.

```
python poctpl <ruta del binario>
```

Figura B.1: Ejecución de la plataforma.

Una vez ejecutado este comando la aplicación mostrará una pequeña consola de comandos en la cual el usuario puede introducir la fórmula que desea comprobar dentro del binario. Esto lo podemos observar en la figura B.2.

```
poctpl > EF[mov regReg32 EAx EAx]
```

Figura B.2: Ejemplo de introducción de fórmula.

Una vez introducido este comando la aplicación comienza el análisis del binario, mostrando su salida en cuanto analice la fórmula en todos los posibles caminos que puede tomar el ejecutable.

2.3 Formulación

La formulación en POCTPL es muy sencilla y solo tiene una detalle, hay que tener en cuenta que fórmulas son de caminos y cuales son de estados. Las formulas de caminos tienen dos modalidades. La primera es que la siguiente fórmula siga siendo una de caminos y la segunda es que la próxima fórmula cambie a ser una de estados.

Para diferenciar esto se usan los paréntesis o los corchetes. Los primeros sirven para indicar que la siguiente fórmula de caminos, mientras que los segundos indican que la siguiente fórmula es de

estados. Por esta razón las dos fórmulas que vemos en al Figura B.3, son completamente diferentes.

```
EF(<formula de caminos>]
EF[<formula de estados>]
```

Figura B.3: Ejemplos de formulación en POCTPL.

Aunque las dos fórmulas tiene operaciones similares, la primera fórmula busca que en un momento determinado se encuentre un conjunto de estados, mientras que la segunda busca que un estado se encuentre en algún instante del camino.

Las operaciones básicas también son diferentes para diferenciar las que son de caminos de las que son de estados, como vemos en la Tabla B.1.

Cuadro B.1: Equivalencia entre las fórmulas de caminos y estados.

Operación	Caminos	Estados
and	&&	AND
or	||	OR
not	-	NOT
implicación	–>	IMP
coimplicación	<->	COIMP

Con estas operaciones y las nociones que hemos dado en el el capitulo dedicado a la verificación es posible realizar cualquier fórmula dentro del sistema.

Bibliografía

[AADO00] Aynur Abdurazik, Paul Ammann, Wei Ding, y Jeff Offutt. Evaluation of three specification-based testing criteria. En *Engineering of Complex Computer Systems, 2000. ICECCS 2000. Proceedings. Sixth IEEE International Conference on*, páginas 179–187. IEEE, 2000.

[AB99] Paul Ammann y Paul E Black. Abstracting formal specifications to generate software tests via model checking. En *Digital Avionics Systems Conference, 1999. Proceedings. 18th*, tomo 2, páginas 10–A. IEEE, 1999.

[ADAHM99] Rajeev Alur, Luca De Alfaro, Thomas A Henzinger, y Freddy YC Mang. *Automating modular verification*. Springer, 1999.

[BAPM83] Mordechai Ben-Ari, Amir Pnueli, y Zohar Manna. The temporal logic of branching time. *Acta informatica*, 20(3):207–226, 1983.

[BB87] Tommaso Bolognesi y Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1):25–59, 1987.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, y Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.

[BCM+90] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, y Lain-Jinn Hwang. Symbolic model checking: 10 20 states and beyond. En *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, páginas 428–439. IEEE, 1990.

[Bil06] Daniel Bilar. Fingerprinting malicious code through statistical opcode analysis. En *ICGeS'07: Proceedings of the 3rd International Conference on Global E-Security*. 2006.

[Bil07] Daniel Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[BK85] Jan A. Bergstra y Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121, 1985.

[BMKK06] Ulrich Bayer, Andreas Moser, Christopher Kruegel, y Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[BNS+06] David Brumley, James Newsome, Dawn Song, Hao Wang, y Somesh Jha. Towards automatic generation of vulnerability-based signatures. En *Security and Privacy, 2006 IEEE Symposium on*, páginas 15–pp. IEEE, 2006.

[Bou09] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231:323–341, 2009.

[BPG07] Sergiy Boroday, Alexandre Petrenko, y Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3–19, 2007.

[BR01] Paul E Black y Scott Ranville. Winnowing tests: Getting quality coverage from a model checker without

quantity. En *Digital Avionics Systems, 2001. DASC. 20th Conference*, tomo 2, páginas 9B6–1. IEEE, 2001.

[BR04] Gogul Balakrishnan y Thomas Reps. Analyzing memory accesses in x86 executables. En *Compiler Construction*, páginas 5–23. Springer, 2004.

[BR10] Gogul Balakrishnan y Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[Bro05] Manfred Broy. *Model-based testing of reactive systems: advanced lectures*, tomo 3472. Springer Science & Business Media, 2005.

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[Bry92] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[BS01] Julian Bradfield y Colin Stirling. Modal logics and mucalculi: an introduction. 2001.

[But69] Robert E Butts. *William Whewell's theory of scientific method*. University of Pittsburgh Pre, 1969.

[CC77] Patrick Cousot y Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. En *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 238–252. ACM, 1977.

[CC99] Patrick Cousot y Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, y Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. En *Computer Aided Verification*, páginas 359–364. Springer, 2002.

[CCGR99] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, y Marco Roveri. Nusmv: A new symbolic model verifier. En *Computer Aided Verification*, páginas 495–499. Springer, 1999.

[CE82] Edmund M Clarke y E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[CES83] Edmund Melson Clarke, E Allen Emerson, y A Prasad Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. En *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 117–126. ACM, 1983.

[CES86] Edmund M. Clarke, E Allen Emerson, y A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CG87] Edmund M Clarke y Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. En *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, páginas 294–303. ACM, 1987.

[CG03] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, y Helmut Veith. Counterexample-guided abstraction refinement. En *Computer aided verification*, páginas 154–169. Springer, 2000.

[CGJ+01] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, y Helmut Veith. Progress on the state explosion problem in model checking. En *Informatics*, páginas 176–194. Springer, 2001.

[CGK+13] Sjoerd Cranen, Jan Friso Groote, Jeroen JA Keiren, Frank PM Stappers, Erik P de Vink, Wieger Wesselink, y Tim AC Willemse. An overview of the mcrl2 toolset and its recent advances. En *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 199–213. Springer, 2013.

[CGMZ95] Edmund M Clarke, Orna Grumberg, Kenneth L McMillan, y Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. En *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, páginas 427–432. ACM, 1995.

[CGP99] Edmund M Clarke, Orna Grumberg, y Doron Peled. *Model checking*. MIT press, 1999.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, páginas 345–363, 1936.

[CJ06] Mihai Christodorescu y Somesh Jha. Static analysis of executables to detect malicious patterns. Informe técnico, DTIC Document, 2006.

[CJLV02] Edmund Clarke, Somesh Jha, Yuan Lu, y Helmut Veith. Tree-like counterexamples in model checking. En *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, páginas 19–29. IEEE, 2002.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, y Randal E Bryant. Semantics-aware malware detection. En *Security and Privacy, 2005 IEEE Symposium on*, páginas 32–46. IEEE, 2005.

[CKNZ12] Edmund M Clarke, William Klieber, Miloš Nováček, y Paolo Zuliani. Model checking and the state explosion problem. En *Tools for Practical Software Verification*, páginas 1–30. Springer, 2012.

[Coh97] Richard Cohen. The defensive java virtual machine specification. Informe técnico, Technical report, Computational Logic Inc, 1997.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, y Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. En *Usenix Security*, tomo 98, páginas 63–78. 1998.

[CSE+96] John Callahan, Francis Schneider, Steve Easterbrook, et al. Automated software testing using model-checking. En *Proceedings 1996 SPIN workshop*, tomo 353. Citeseer, 1996.

[CSX08] Christoph Csallner, Yannis Smaragdakis, y Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):8, 2008.

[DHL05] George Devaraj, Mats Per Erik Heimdahl, y Donglin Liang. Coverage-directed test generation with model checkers: Challenges and opportunities. En *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, tomo 1, páginas 455–462. IEEE, 2005.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DNV90] Rocco De Nicola y Frits Vaandrager. Action versus state based logics for transition systems. En *Semantics of Systems of Concurrent Processes*, páginas 407–419. Springer, 1990.

[DPG05] Mila Dalla Preda y Roberto Giacobazzi. Semantic-based code obfuscation by abstract interpretation. En *Automata, Languages and Programming*, páginas 1325–1336. Springer Berlin Heidelberg, 2005.

[DPGD15] Mila Dalla Preda, Roberto Giacobazzi, y Saumya Debray. Unveiling metamorphism by abstract interpretation of code properties. *Theoretical Computer Science*, 577:74–97, 2015.

[DS14] Christian Doczkal y Gert Smolka. Completeness and decidability results for ctl in coq. En *Interactive Theorem Proving*, páginas 226–241. Springer, 2014.

[DSSRS02] Yifei Dong, Beata Sarna-Starosta, CR Ramakrishnan, y Scott A Smolka. Vacuity checking in the modal mu-calculus*. En *Algebraic Methodology and Software Technology*, páginas 147–162. Springer, 2002.

[DZ03] Chen Ding y Yutao Zhong. Predicting whole-program locality through reuse distance analysis. En *ACM SIGPLAN Notices*, tomo 38, páginas 245–257. ACM, 2003.

[EC80] E Allen Emerson y Edmund M Clarke. *Characterizing correctness properties of parallel programs using fixpoints*. Springer, 1980.

[Edm94] Norman W Edmund. *The general pattern of the scientific method (SM-14)*. ERIC, 1994.

[EFM97] André Engels, Loe Feijs, y Sjouke Mauw. Test generation for intelligent networks using model checking. En *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 384–398. Springer, 1997.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, y Gordon Woodhull. Graphviz—open source graph drawing tools. En *Graph Drawing*, páginas 483–484. Springer, 2002.

[EH85] E Allen Emerson y Joseph Y Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of computer and system sciences*, 30(1):1–24, 1985.

[EH86] E Allen Emerson y Joseph Y Halpern. "sometimes.ªnd "not neverrevisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, 1986.

[EL85] E Allen Emerson y Chin-Laung Lei. Modalities for model checking (extended abstract): branching time strikes back. En *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 84–96. ACM, 1985.

[ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, y Stefan Leue. Directed explicit model checking with hsf-spin. En *Proceedings of the 8th international SPIN workshop on Model checking of software*, páginas 57–79. Springer-Verlag New York, Inc., 2001.

[FAW07] Gordon Fraser, Bernhard K Aichernig, y Franz Wotawa. Handling model changes: Regression testing and test-suite update with model-checkers. *Electronic Notes in Theoretical Computer Science*, 190(2):33–46, 2007.

[Flo67] Robert W Floyd. Assigning meanings to programs. En *Proc. Symp. in Applied Mathematics*, tomo 19, páginas 19–32. 1967.

[Fre79] Gottlob Frege. Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought. *From Frege to Gödel: A source book in mathematical logic*, 1931:1–82, 1879.

[FW07a] Gordon Fraser y Franz Wotawa. Mutant minimization for model-checker based test-case generation. En *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, páginas 161–168. IEEE, 2007.

[FW07b] Gordon Fraser y Franz Wotawa. Redundancy based test-suite reduction. En *Fundamental Approaches to Software Engineering*, páginas 291–305. Springer, 2007.

[FW07c] Gordon Fraser y Franz Wotawa. Test-case prioritization with model-checkers. En *25th conference on IASTED International*. 2007.

[FW07d] Gordon Fraser y Franz Wotawa. Using ltl rewriting to improve the performance of model-checker based test-case generation. En *Proceedings of the 3rd international workshop on Advances in model-based testing*, páginas 64–74. ACM, 2007.

[GCEC12] Clint Gibler, Jonathan Crussell, Jeremy Erickson, y Hao Chen. *AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale*. Springer, 2012.

[GKO15] Jan Friso Groote, Tim WDM Kouters, y Ammar Osaiweran. Specification guidelines to avoid the state space explosion problem. *Software Testing, Verification and Reliability*, 25(1):4–33, 2015.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[Gos85] Ardeshir Goshtasby. Description and discrimination of planar shapes using shape matrices. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):738–743, 1985.

[Gos95] James Gosling. Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). En *ACM Sigplan Notices*, tomo 30, páginas 111–118. ACM, 1995.

[GPVW95] Rob Gerth, Doron Peled, Moshe Y Vardi, y Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. En *International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[GVNM+48] Herman Heine Goldstine, John Von Neumann, Hungary Mathematician, John von Neumann, y John von Neumann. *Planning and coding of problems for an electronic computing instrument*. Institute for Advanced Study, 1948.

[GW14] Henning Günther y Georg Weissenbacher. Incremental bounded software model checking. En *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, páginas 40–47. ACM, 2014.

[H+85] Charles Antony Richard Hoare et al. *Communicating sequential processes*, tomo 178. Prentice-hall Englewood Cliffs, 1985.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, y Hasan Ural. Data flow testing as model checking. En *Software Engineering, 2003. Proceedings. 25th International Conference on*, páginas 232–242. IEEE, 2003.

[HDMR04] Grégoire Hamon, Leonardo De Moura, y John Rushby. Generating efficient test sets with a model checker. En *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, páginas 261–270. IEEE, 2004.

[HG04] Mats PE Heimdahl y Devaraj George. Test-suite reduction for model based tests: Effects on test quality and implications for testing. En *Proceedings of the 19th IEEE international conference on Automated software engineering*, páginas 176–185. IEEE Computer Society, 2004.

[HGS93] M Jean Harrold, Rajiv Gupta, y Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993.

[HGW04] Mats Per Erik Heimdahl, Devaraj George, y Robert Weber. Specification test coverage adequacy criteria= specification test generation inadequacy criteria. En *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, páginas 178–186. IEEE, 2004.

[HJ03] Martin Hofmann y Steffen Jost. Static prediction of heap space usage for first-order functional programs. En *ACM SIGPLAN Notices*, tomo 38, páginas 185–197. ACM, 2003.

[HJBJ77] VE Hoggatt Jr y Marjorie Bicknell-Johnson. Fibonacci convolution sequences. *Fibonacci Quart*, 15(2):117–122, 1977.

[HLSU02] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, y Hasan Ural. A temporal logic based theory of test coverage and generation. En *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 327–341. Springer, 2002.

[HM85] Matthew Hennessy y Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32(1):137–161, 1985.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hol90] Gerard J Holzmann. Design and validation of protocols. En *Tutorial Computer Networks and ISDN Systems*. Citeseer, 1990.

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, tomo 1003. Addison-Wesley Reading, 2004.

[HPBLG05] Manuel V Hermenegildo, Germán Puebla, Francisco Bueno, y Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the ciao system preprocessor). *Science of Computer Programming*, 58(1):115–140, 2005.

[HR01] Klaus Havelund y Grigore Rosu. Monitoring programs using rewriting. En *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, páginas 135–143. IEEE, 2001.

[HRV+04] Mats PE Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, y Jimin Gao. Auto-generating test sequences using model checkers: A case study. En *Formal Approaches to Software Testing*, páginas 42–59. Springer, 2004.

[HU05] Hyoung Seok Hong y Hasan Ural. *Using model checking for reducing the cost of test generation*. Springer, 2005.

[JH03] James A Jones y Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *Software Engineering, IEEE Transactions on*, 29(3):195–209, 2003.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, y David Pichardie. A formally-verified c static analyzer. En *POPL 2015-42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2015.

[Kam68] Hans Kamp. Tense logic and the theory of linear order. 1968.

[Kel76] Robert M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[KGWT11] Hadas Kress-Gazit, Tichakorn Wongpiromsarn, y Ufuk Topcu. Mitigating the state explosion problem of temporal logic synthesis. *IEEE Robotics & Automation Magazine*, 2011.

[KKSV05] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, y Helmut Veith. Detecting malicious code by model checking. En *Detection of Intrusions and Malware, and Vulnerability Assessment*, páginas 174–187. Springer, 2005.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical computer science*, 27(3):333–354, 1983.

[KPK87] Ho Sung Kim, Kyu Ho Park, y Myunghwan Kim. Shape decomposition by collinearity. *Pattern recognition letters*, 6(5):335–340, 1987.

[Kri63] Saul A Kripke. Semantical considerations on modal logic. 1963.

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler09b] Xavier Leroy. A formally verified compiler backend. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[Lew12] Clarence Irving Lewis. Implication and the algebra of logic. *Mind*, páginas 522–531, 1912.

[LLL59] Clarence Irving Lewis, Cooper Harold Langford, y P Lamprecht. *Symbolic logic*. Dover Publications New York, 1959.

[LMS02] François Laroussinie, Nicolas Markey, y Philippe Schnoebelen. Temporal logic with forgettable past. En *Logic in Computer Science, Symposium on*, páginas 383–383. IEEE Computer Society, 2002.

[Loh07] Peter Lohmann. Pspace-completeness of ltl/ctl model checking. 2007.

[LP85] Orna Lichtenstein y Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. En *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, páginas 97–107. ACM, 1985.

[LS92] Janusz Laski y Wojciech Szermer. Identification of program modifications and its applications in software maintenance. En *Software Maintenance, 1992. Proceerdings., Conference on*, páginas 282–290. IEEE, 1992.

[LS09] Martin Leucker y Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[LYBB13] Tim Lindholm, Frank Yellin, Gilad Bracha, y Alex Buckley. *The Java Virtual Machine Specification*. Addison-Wesley, 2013.

[Mar90] John Markoff. Computer intruder is put on probation and fined 10,000. *New York Times*, 5, 1990.

[McM93] Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

[MFG04] Robert Meolic, Alessandro Fantechi, y Stefania Gnesi. *Witness and counterexample automata for ACTL*. Springer, 2004.

[MFS12] Florian Merz, Stephan Falke, y Carsten Sinz. Llbmc: Bounded model checking of c and c++ programs using a compiler ir. En *Verified Software: Theories, Tools, Experiments*, páginas 146–161. Springer, 2012.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

[Mil89] Robin Milner. *Communication and concurrency*, tomo 84. Prentice hall New York etc., 1989.

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus*. Cambridge university press, 1999.

[MP92] Zohar Manna y Amir Pnueli. *The temporal logic of reactive and concurrent systems: specifications*, tomo 1. springer, 1992.

[MS70] John C Mott-Smith. Medial axis transformations. *Picture Processing and Psychopictorics*, páginas 267–283, 1970.

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[Neb00] Gary Nebbett. *Windows NT/2000 native API reference*. Sams Publishing, 2000.

[NS05] James Newsome y Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[NS07] Nicholas Nethercote y Julian Seward. How to shadow every byte of memory used by a program. En *Proceedings of the 3rd international conference on Virtual execution environments*, páginas 65–74. ACM, 2007.

[NVB10] Aleksandar Nanevski, Viktor Vafeiadis, y Josh Berdine. Structuring the verification of heap-manipulating programs. En *ACM Sigplan Notices*, tomo 45, páginas 261–274. ACM, 2010.

[OBY03] Vadim Okun, Paul E Black, y Yaacov Yesha. Testing with model checker: Insuring fault visibility. En *Proceedings of 2002 WSEAS international conference on system science, applied mathematics & computer science, and power engineering systems*, páginas 1351–1356. 2003.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[OWA13] The open web Application Security Proyect OWASP. Owasp top 10 - 2013, los riesgos más críticos en aplicaciónes web, 2013.

[PC98] Nuno Paiva y Carlos Caleiro. Temporal logic in coq. Informe técnico, Citeseer, 1998.

[PCJD07] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, y Saumya Debray. A semantics-based approach to malware detection. En *ACM SIGPLAN Notices*, tomo 42, páginas 377–388. ACM, 2007.

[Pea87] Giuseppe Peano. *Applicazioni geometriche del calcolo infinitesimale*, tomo 2. Fratelli Bocca, 1887.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. En *Computer Aided Verification*, páginas 409–423. Springer, 1993.

[PF77] Eric Persoon y King-Sun Fu. Shape discrimination using fourier descriptors. *Systems, Man and Cybernetics, IEEE Transactions on*, 7(3):170–179, 1977.

[Pnu77] Amir Pnueli. The temporal logic of programs. En *Foundations of Computer Science, 1977., 18th Annual Symposium on*, páginas 46–57. IEEE, 1977.

[Pri03] Arthur N Prior. *Time and modality*. Oxford University Press, 2003.

[QS82] Jean-Pierre Queille y Joseph Sifakis. Specification and verification of concurrent systems in cesar. En *International Symposium on Programming*, páginas 337–351. Springer, 1982.

[RBB13] Petr Rockai, Jiri Barnat, y Lubos Brim. Improved state space reductions for ltl model checking of c and c++ programs. En *NASA Formal Methods*, páginas 1–15. Springer, 2013.

[RBH+11] Thomas Reinbacher, Jörg Brauer, Martin Horauer, Andreas Steininger, y Stefan Kowalewski. Past time ltl runtime verification for microcontroller binary code. En *Formal Methods for Industrial Critical Systems*, páginas 37–51. Springer, 2011.

[Rey67] John C Reynolds. Automatic computation of data set definitions. IFIP Congress, 1967.

[RFS+13] Mikhail Ramalho, Mauro Freitas, Felipe Sousa, Hendrio Marques, Lucas Cordeiro, y Bernd Fischer. Smt-based bounded model checking of c++ programs. En *Engineering of Computer Based Systems (ECBS), 2013 20th IEEE International Conference and Workshops on the*, páginas 147–156. IEEE, 2013.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, y Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. En *Software Maintenance, 1998. Proceedings., International Conference on*, páginas 34–43. IEEE, 1998.

[RHVRH02] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, y Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[RRH03] Silvius Rus, Lawrence Rauchwerger, y Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251–283, 2003.

[RUCH99] Gregg Rothermel, Roland H Untch, Chengyun Chu, y Mary Jean Harrold. Test case prioritization: An empirical study. En *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, páginas 179–188. IEEE, 1999.

[RW68] Bertrand Russell y Alfred North Whitehead. *Principia mathematica*. Cambridge University Press, 1968.

[SBPV12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, y Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. En *USENIX ATC 2012*. 2012.

[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, y Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. En *Information systems security*, páginas 1–25. Springer, 2008.

[Sha14] Adrian Mark Shaw. *Partial Order Reduction with Compositional Verification*. Tesis Doctoral, University of Waikato, 2014.

[Som12] F Somenzi. Cudd: Cu decision diagram package, release 2.4. 2. department of electrical, computer, and energy engineering, university of colorado at boulder, 2012.

[Spr98] Christoph Sprenger. A verified model checker for the modal μ-calculus in coq. En *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 167–183. Springer, 1998.

[SS13] Marcelo Sousa y Alper Sen. Llvmvf: A generic approach for verification of multicore software. *Journal of Electronic Testing*, 29(5):635–646, 2013.

[ST12] Fu Song y Tayssir Touili. Efficient malware detection using model-checking. En *FM 2012: Formal Methods*, páginas 418–433. Springer, 2012.

[ST13a] Fu Song y Tayssir Touili. Ltl model-checking for malware detection. En *Tools and Algorithms for the Construction and Analysis of Systems*, páginas 416–431. Springer, 2013.

[ST13b] Fu Song y Tayssir Touili. Pommade: pushdown model-checking for malware detection. En *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, páginas 607–610. ACM, 2013.

[ST14] Fu Song y Tayssir Touili. Model-checking for android malware detection. En *Programming Languages and Systems*, páginas 216–235. Springer, 2014.

[TPC+13] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, y Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. En *Fundamental Approaches to Software Engineering*, páginas 210–225. Springer, 2013.

[TS02] Timothy Tsai y Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. En *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, página 541. IEEE, 2002.

[Tur49] Alan Turing. Checking a large routine. En *The early British computer conferences*, páginas 70–72. MIT Press, 1949.

[TW07] Ming-Hsien Tsai y Bow-Yaw Wang. Formalization of ctl* in calculus of inductive constructions. En *Advances in Computer Science-ASIAN 2006. Secure Software and Related Issues*, páginas 316–330. Springer, 2007.

[VCK14] Manuel Valdiviezo, Cristina Cifuentes, y Padmanabhan Krishnan. A method for scalable and precise bug finding using program analysis and model checking. En *Programming Languages and Systems*, páginas 196–215. Springer, 2014.

[VW86] Moshe Y Vardi y Pierre Wolper. An automata-theoretic approach to automatic program verification. En *1st Symposium in Logic in Computer Science (LICS)*, páginas 322–331. IEEE Computer Society, 1986.

[Wai08] Nicolas Waisman. Apology of 0days, 2008.

[WASF07] Duminda Wijesekera, Paul Ammann, Lingya Sun, y Gordon Fraser. Relating counterexamples to test cases in ctl model checking specifications. En *Proceedings of the 3rd international workshop on Advances in model-based testing*, páginas 75–84. ACM, 2007.

[WHLM95] W Eric Wong, Joseph R Horgan, Saul London, y Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. En *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, páginas 41–41. IEEE, 1995.

[XDR04] Lihua Xu, Marcio Dias, y Debra J Richardson. Generating regression tests via model checking. En *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, páginas 336–341. IEEE, 2004.

[XF12] Cheng Xu y Philip WL Fong. The specification and compilation of obligation policies for program monitoring. En *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, páginas 77–78. ACM, 2012.

[XSCM04] J-Y Xu, Andrew H Sung, Patrick Chavez, y Srinivas Mukkamala. Polymorphic malicious executable scanner by api sequence analysis. En *Hybrid Intelligent Systems, 2004. HIS'04. Fourth International Conference on*, páginas 378–383. IEEE, 2004.

[YCGK07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, y Robert M Kirby. Distributed dynamic partial order reduction based verification of threaded software. En *Model Checking Software*, páginas 58–75. Springer, 2007.

[Yel95] Frank Yellin. Low level security in java. En *Proceedings of the 4th International World Wide Web Conference, Boston, Massachusetts*. 1995.

[YJPVdE12] Yves Younan, Wouter Joosen, Frank Piessens, y Hans Van den Eynden. Improving memory management security for c and c++. *Security-Aware Systems Applications and Software Development Methods*, página 190, 2012.

[YSE+07] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, y Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. En *Proceedings of the 14th ACM conference on*

Computer and communications security, páginas 116–127. ACM, 2007.

[ZML07] Hongwei Zeng, Huaikou Miao, y Jing Liu. Specification-based test generation and optimization using model checking. En *Theoretical Aspects of Software Engineering, 2007. TASE'07. First Joint IEEE/IFIP Symposium on*, páginas 349–355. IEEE, 2007.

[ZRZM12] Yingying Zhang, Emmanuel Rodriguez, Hao Zheng, y Chris Myers. An improvement in partial order reduction using behavioral analysis. En *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, páginas 100–107. IEEE, 2012.

[ZZM06] Hao Zhong, Lu Zhang, y Hong Mei. An experimental comparison of four test suite reduction techniques. En *Proceedings of the 28th international conference on Software engineering*, páginas 636–640. ACM, 2006.