Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Tesis Doctoral

Detección de Vulnerabilidades y Análisis de Fallos con Técnicas de Aprendizaje Automatizado

Gustavo Grieco

Director: Dr. Guillermo L. Grimblat

Codirector: Dr. Laurent Mounier

Tesis presentada en la Facultad de Ciencias Exactas, Ingeniería y Agrimensura en cumplimiento parcial de los requisitos para optar al título de

Doctor en Informática

Marzo 2018

Certifico que el trabajo incluido en esta tesis es el resultado de tareas de investigación originales y que no ha sido presentado para optar a un título de posgrado en ninguna otra Universidad o Institución.

Gustavo Grieco

Resumen

La gran mayoría de los programas que utilizamos diariamente contienen numerosos errores, los cuales causan desde dificultades menores de uso hasta su terminación anormal y pérdida de información. Desafortunadamente, algunos fallos pueden ser aprovechados para atacar la integridad, confidencialidad o disponibilidad de un sistema. Debido a que ciertos sistemas informáticos cumplen un rol central en la vida moderna, es muy importante no sólo encontrar errores en los programas sino también identificar cuáles pueden resultar en vulnerabilidades que afectan a la seguridad de los mismos.

La presente tesis comienza introduciendo conceptos básicos sobre aprendizaje automatizado y seguridad en programas para ser utilizados extensivamente durante la misma. Luego, se presentan varias herramientas y técnicas novedosas para la detección de vulnerabilidades de manera automática. La primera de ellas es *QuickFuzz*, una herramienta de *fuzzing* que utiliza la generación de entradas malformadas para producir fallos en programas que procesan varios formatos de archivos complejos. La segunda es *XCraft*, una herramienta utilizada para la evaluación de fallos de seguridad a gran escala mediante técnicas de caja negra. También se presenta *VDiscover*, una novedosa herramienta para la identificación de vulnerabilidades a partir de fallos suministrados por el usuario. La misma utiliza varias técnicas de aprendizaje automatizado para estimar la probabilidad de que un fallo pueda esconder una vulnerabilidad que un atacante podría aprovechar fácilmente.

Finalmente, se exponen las conclusiones y las ideas futuras para continuar con la investigación y el desarrollo de nuevas técnicas de detección de fallos y vulnerabilidades en programas.

Abstract

Most of the software we use daily contains numerous issues, causing from minor annoyances in the usage to abnormal termination and data loss. Unfortunately, some of these can be exploited to attack the integrity, confidentiality or disponibility of a system. Some of the software affected is also essential in our society because it manages our infrastructure. Therefore, it is quite important to identify the issues they have, as well as the potential security vulnerabilities that an attacker could *exploit*.

This thesis starts reviewing some basic concepts on machine learning and computer security to be used during the following chapters. Then, it presents several novelty tools and techniques to detect automatically detect vulnerabilities in software. The first one is *QuickFuzz*, a tool to that uses fuzzing to generate malformed inputs in order to uncover issues in programs parsing complex file-types. The second one is *XCraft*, a tool created to identify security related issues using blackbox techniques. The last one is *VDiscover*, a novelty tool designed to uncover potentially exploitable memory corruptions in crashes. It uses machine learning to discover patterns in the program execution to estimate how likely a crash can be exploited by an attacker.

The final chapter presents the conclusions and future work ideas in order to continue researching and developing new techniques to discover issues and vulnerabilities in software.

Agradecimientos

A mi familia, mis amigos y mi novia Polly.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la actualidad, a pesar del progreso tecnológico que ofrecen los lenguajes de programación y las técnicas de ingeniería de software, la gran mayoría de los programas que utilizamos diariamente contienen numerosos errores. Estos se encuentran tanto en componentes críticos de los sistemas operativos como en las aplicaciones de usuario (por ejemplo, en procesadores de palabras y sistemas web). No obstante, no todos los fallos son iguales: algunos son más críticos que otros porque pueden comprometer la seguridad de todo el sistema. Nos referimos a estos últimos como *vulnerabilidades de software*.

Debido a que estas vulnerabilidades son cada vez más relevantes en los sistemas que usamos en nuestra vida diaria, es muy importante no sólo encontrar fallos en los programas sino también identificar cuáles pueden resultar en vulnerabilidades que afectan a la seguridad y ponen en peligro a los usuarios. De esta manera, se pueden solucionar antes de que los sistemas sean atacados. En este sentido, es importante destacar que los efectos producidos en los últimos años por las vulnerabilidades de software que afectan a un número masivo de usuarios han sido catastróficos. Por ejemplo, ha permitido que el software malicioso tome el control de los sistemas informáticos para sabotear desde procesos industriales muy específicos [85] hasta sistemas de usuarios hogareños y comerciales a escala planetaria [75, 7, 51].

Desafortunadamente, la detección de fallos y vulnerabilidades puede resultar una tarea muy compleja, tal como el libro *Why programs fail: A guide to systematic debugging* [98] destaca:

> *"El **defecto** causó una **afección**, que provocó una **falla**: cuando vimos la falla, rastreamos la afección y finalmente solucionamos el defecto."*

En el contexto de la detección de vulnerabilidades, una falla se puede definir como un comportamiento incorrecto o inesperado por parte de un programa. Por ejemplo, la terminación inesperada de un programa o la ejecución de un bucle sin salida pueden ser considerados fallos. Una afección podría ser la escritura de memoria más allá del espacio reservado. Su rastreo puede realizarse monitoreando las entradas de un programa y su código hasta que se observa el defecto (por ejemplo, en el uso de código inseguro para copiar memoria).

En la actualidad, existen varias técnicas para la detección de ciertos tipos de fallos y vulnerabilidades como desbordamientos de *buffers* o acceso a memoria mediante punteros nulos. A grandes rasgos, estas pueden ser clasificadas entre las que utilizan análisis estático y las que utilizan análisis dinámico.

El **análisis estático** es un método basado en la inspección de las instrucciones del programa sin ejecutarlas para deducir si existe alguna falla potencial [16, 17]. Esta metodología se utiliza efectivamente en el contexto de ciertas aplicaciones de dominio específico, por ejemplo, en sistemas embebidos o software aeroespacial. Desafortunadamente, el uso de estas técnicas para analizar software de propósito general no resulta suficientemente preciso y puede dar lugar a un gran número de falsos positivos: es decir, potenciales fallos o vulnerabilidades que en realidad nunca podrían suceder.

En cambio, el **análisis dinámico** ejecuta el programa sistemáticamente con distintas entradas para verificar si falla o no. Uno de los enfoques más efectivos para la detección de vulnerabilidades en software complejo se basa en el *fuzzing*: la generación de datos de entrada con el objetivo de lograr un comportamiento inesperado en un programa. De todas maneras, debido a la gran complejidad del software, las pruebas que utilizan análisis dinámico pueden requerir gran cantidad de recursos computacionales y de tiempo para realizarse.

El problema de detectar fallos y vulnerabilidades en el software se agrava debido a las escalas de tamaño y complejidad de los programas actuales: una instalación típica de un sistema operativo provee acceso a miles de aplicaciones distintas. A modo de ejemplo, el sistema operativo *Debian GNU/Linux* dispone de repositorios de paquetes con más de 30.000 programas: aplicaciones para visualizar o convertir multimedia,

navegadores web, herramientas para desarrollar programas, entre muchas otras. Este enorme volumen de aplicaciones genera grandes cantidades de reportes de errores: Debian posee más de 80.000 reportes de posibles fallos. Muchos de estos fallos deben ser analizados manualmente por expertos.

Para mitigar este problema, existen algunas metodologías y herramientas para realizar pruebas de software a gran escala, pero no siempre resultan adecuadas. Como ejemplo de esto, podemos citar el ciclo de desarrollo de software seguro de *Microsoft* (o SDL [55], por sus siglas en inglés), un proceso incremental por el cual las pruebas de software para detectar fallos y vulnerabilidades se realizan en todos los niveles del desarrollo. Sin embargo, su uso para mejorar sistemas operativos ya desarrollados está fuera del alcance de SDL.

De todas maneras, descubrir fallos y vulnerabilidades no es suficiente para lograr un software seguro. Es de igual (o mayor) importancia demostrar qué tan grave es cada fallo: si resulta crítico para la seguridad de un programa o no. El proceso para determinar la severidad de un fallo depende fuertemente del programa en sí mismo: por ejemplo, para sistemas críticos, la sola existencia de un fallo que detenga el programa se suele considerar crítica. En otras aplicaciones, como el almacenamiento de documentos confidenciales, una potencial filtración de información resulta crítica.

La presente tesis propone varios objetivos en la dirección de automatizar y acelerar la búsqueda de fallos y vulnerabilidades en programas. El primer objetivo de este trabajo consiste en indagar, proponer y desarrollar nuevas técnicas y herramientas para la detección de vulnerabilidades y fallos en programas. Dada la complejidad y dificultad de las tareas involucradas en la detección de fallos y vulnerabilidades, en esta tesis se define un segundo objetivo: el uso de distintas técnicas de aprendizaje automatizado y minería de datos para aprender relaciones entre defectos, afecciones y fallos en distintos componentes de software. Estas técnicas ya han sido usadas con éxito en una gran variedad de aplicaciones [22, 37, 28]. En particular, se busca poder inferir la probabilidad de que los fallos de un programa puedan afectar la seguridad de un sistema; una tarea compleja, incluso para expertos humanos.

A continuación, se detalla la estructura del resto de la tesis. En el Capítulo 2, se explican las bases y los conceptos fundamentales del aprendizaje automatizado en sus distintos enfoques. Asimismo, se definen técnicas útiles para la minería de texto, y se presenta un ejemplo de clasificación de grupos de noticias según su tópico.

El Capítulo 3 introduce y formaliza conceptos básicos sobre seguridad de programas con el enfoque utilizado en el resto de la tesis. En particular, se abordan los fallos de desbordamientos de *buffers* y de enteros, junto con ejemplos mínimos donde se muestran los defectos, las afecciones y las fallas que estos producen. También se detalla cómo estos pueden afectar la seguridad de un sistema. Finalmente, se cierra el capítulo con un ejemplo más complejo, donde se muestran múltiples fallas y vulnerabilidades en un código más complejo y real: un conversor de archivos de *Microsoft PowerPoint*™.

En el Capítulo 4, se presentan distintas técnicas de detección de vulnerabilidades y fallos, desde las más tradicionales hasta las más modernas. Este capítulo se enfoca, en particular, en dos de las herramientas desarrolladas en el transcurso de esta tesis:

- *QuickFuzz*: una herramienta de *fuzzing* que utiliza la generación de entradas malformadas para producir fallos en programas que procesan varios formatos de archivos complejos. Esta herramienta ha sido particularmente efectiva en la detección de fallos y vulnerabilidades en software complejo, tal como navegadores web, librerías de conversión de imágenes y compresores de archivos.

- *XCraft*: una herramienta utilizada para la evaluación de fallos de seguridad a gran escala. Esta herramienta no solo sirve para descubrir nuevos fallos, sino que además se centra en identificar las vulnerabilidades críticas, de manera que sea posible priorizar este tipo de errores en el proceso del desarrollo de software. *XCraft* no requiere gran cantidad de recursos computacionales, por lo cual puede ser utilizada para identificar vulnerabilidades en una gran cantidad de programas distintos.

Para ambos casos, se presentan los resultados obtenidos, destacando sus fortalezas y debilidades, respecto a los enfoques que utilizan.

A continuación, en el Capítulo 5, se presenta y desarrolla *VDiscover*: una nueva técnica para la identificación de vulnerabilidades a partir de fallos utilizando aprendizaje automatizado. Esta técnica apunta a estimar la probabilidad de que el fallo pueda esconder una vulnerabilidad que un atacante podría aprovechar fácilmente.

Finalmente, el Capítulo 6 presenta las conclusiones de los capítulos anteriores y expone las ideas futuras para continuar con la investigación y el desarrollo de nuevas técnicas de detección de fallos y vulnerabilidades en programas.

Los contenidos de esta tesis se basan en las siguientes publicaciones y trabajos inéditos:

- Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist y Laurent Mounier. "Toward Large-Scale Vulnerability Discovery using Machine Learning". En los anales de la "Sixth ACM Conference on Data and Application Security and Privacy" (CODASPY). 2016.

- Gustavo Grieco, Martín Ceresa y Pablo Buiras. "QuickFuzz: an automatic random fuzzer for common file formats". En los anales de la "9th International Symposium on Haskell". 2016.

- Gustavo Grieco, Martín Ceresa, Agustín Mista y Pablo Buiras. "QuickFuzz Testing for Fun and Profit". Aceptado para su publicación en el "Journal of Systems and Software". 2017.

Además, durante el transcurso de este trabajo, se realizaron las siguientes colaboraciones:

- Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco y David Brumley. "Optimizing seed selection for fuzzing". En los anales de la "23rd USENIX conference on Security Symposium". 2014.

- Shubham Tripathi, Gustavo Grieco y Sanjay Rawat. "Exniffer: Learning to Rank Crashes by Assessing the Exploitability from Memory Dump". Aceptado para su publicación en la "24th Asia-Pacific Software Engineering Conference". 2017.

Capítulo 2

Aprendizaje Automatizado

2.1. Conceptos fundamentales

Una de las definiciones más aceptadas de **aprendizaje automatizado** fue dada por Tom Mitchell en su libro *Machine Learning* [61] (1997):

> "Se dice que un programa de computadora aprende de la **experiencia E** sobre un tipo de **tarea T** y **medida de desempeño P**, si su desempeño en la tarea de tipo T, medido en términos de P, mejora con la experiencia E"

Todas las tareas del aprendizaje automatizado requieren **datos de entrenamiento** de los cuales aprender. No obstante, dentro de este campo, se reconocen distintas áreas especializadas según las características de la experiencia, el tipo de tarea y la medida de desempeño. La primera distinción se basa en la noción de supervisión durante el aprendizaje. El **aprendizaje supervisado** requiere información precisa sobre los datos que se utilizan durante el entrenamiento en forma de **etiquetas** a predecir. El objetivo de este tipo de aprendizaje es obtener una buena aproximación de una función F, donde X son los datos de entrada e Y sus etiquetas:

$$F(X) = Y.$$

La etiqueta que se busca aprender puede ser una clase proveniente de un número

finito de alternativas discretas (también llamadas clases o categorías). En este caso, el problema se denomina de **clasificación**. Por ejemplo, pertenecen a este tipo de tareas: la predicción de los dígitos a partir de imágenes de números escritos a mano [97] o la detección de objetos en imágenes naturales [47].

En cambio, si la etiqueta corresponde a un valor real de alguna magnitud a predecir, el problema a resolver se denomina de **regresión**. Por ejemplo, pertenece a este tipo de tareas: la estimación del precio de las propiedades en un área metropolitana, en función de las características de la propiedad y su zona [35].

Por otro lado, la otra gran área del aprendizaje automatizado se denomina **aprendizaje no supervisado**. Dicho aprendizaje se centra en acumular experiencia durante la aplicación de la tarea, sin utilizar la información de las etiquetas. De esta manera, el programa aún puede aprender a realizar distintas tareas, como la reducción de dimensionalidad o la visualización de los datos, conservando sus propiedades intrínsecas. Por ejemplo, pertenece a este tipo de tareas: la detección de anomalías para la identificación de intrusiones informáticas en una red militar de computadoras [49].

En todas las áreas del aprendizaje automatizado, los modelos se utilizan siguiendo un proceso de dos fases: **entrenamiento** y **predicción**. En la fase de entrenamiento, los modelos aprenden de los datos mediante el ajuste de sus **parámetros** internos, basándose en los datos que reciben. Además, existe una gran variedad de modelos que también poseen parámetros especiales que pueden ajustar el modelo de alguna forma. Estos parámetros especiales son llamados **hiperparámetros** y son necesarios para determinar el número de parámetros a aprender, la rigidez del modelo y la tolerancia a errores, entre otras características. Finalmente, en la fase de predicción, los modelos pueden usarse para predecir información de datos desconocidos, de acuerdo con la aplicación específica de aprendizaje automatizado que se esté desarrollando.

A lo largo de esta tesis, se hará uso de conceptos básicos de aprendizaje automatizado, en particular de los métodos supervisados y no supervisados de clasificación que se detallan en este capítulo. En contrapartida, no se abordarán tópicos relacionados con las tareas de regresión.

2.1.1. Medidas de desempeño

La evaluación de los modelos entrenados, utilizando aprendizaje supervisado, es un aspecto fundamental para su correcto funcionamiento. En el caso de las tareas de clasificación, la medida natural de desempeño consiste en el error en la clasificación, utilizando un conjunto de datos nunca antes visto. Dicha medida suele expresarse mediante el porcentaje de los datos con predicciones incorrectas.

Para poder entender mejor los aciertos y los errores cometidos por los predictores, es útil separar los datos previamente por clase y computar el porcentaje de errores para cada clase. Una forma muy utilizada de mostrar un resumen del desempeño de un predictor ya entrenado es mediante una **matriz de confusión**. En dicha matriz, cada fila representa a la clase real de los datos a predecir, mientras que cada columna muestra las clases predichas por el clasificador a evaluar. De esta manera, el número de ejemplos correctamente clasificados se sitúa en la diagonal principal, mientras que los ejemplos erróneamente clasificados se encuentran fuera de dicha diagonal. Por ejemplo, si se tienen 3 clases, la Tabla 2.1 refleja el resultado de un predictor para las mismas, mostrando cuál es el número de ejemplos correcta e incorrectamente clasificados para todas las clases.

		Clase predicha		
		Clase 1	Clase 2	Clase 3
Clase real	Clase 1	$\mathbf{C_{11}}$	C_{12}	C_{13}
	Clase 2	C_{21}	$\mathbf{C_{22}}$	C_{23}
	Clase 3	C_{31}	C_{32}	$\mathbf{C_{33}}$

Tabla 2.1: Matriz de confusión para 3 clases

Esta matriz, brinda información importante sobre el desempeño de un clasificador, ya que permite identificar rápidamente si el clasificador comete errores al confundir una clase con otra. Para obtener el error de predicción por clase, simplemente calculamos el porcentaje de aciertos y errores por cada fila.

2.1.2. Consideraciones Importantes

El aprendizaje automatizado debe realizarse cuidadosamente para evitar los problemas potenciales presentes durante la fase de entrenamiento. Las soluciones a estos

problemas definen metodologías y técnicas esenciales para que sea posible realizar predicciones precisas en un gran número de aplicaciones distintas.

2.1.2.1. Sobreajuste y Subajuste

Todos los métodos de aprendizaje automatizado son susceptibles al **sobreajuste**. Dicho fenómeno se produce cuando el modelo intenta explicar patrones presentes en los datos de entrenamiento que disminuyen la efectividad en la predicción de otros datos nunca vistos. En general, este problema se manifiesta como una estimación muy optimista del error en los datos de entrenamiento.

Para evitar este fenómeno, se deberán separar los datos para entrenar un modelo de aprendizaje supervisado en tres subconjuntos: **entrenamiento**, **validación** y **prueba**:

- Una muestra de datos para ajustar los parámetros del modelo supervisado. Este subconjunto se denomina *conjunto de entrenamiento*.
- Una muestra nunca antes utilizada de datos a predecir, de manera que sea posible obtener una estimación del error de predicción sin el sesgo causado por el sobreajuste [4]. Este subconjunto se denomina *conjunto de prueba*.
- Si el modelo tuviera hiperparámetros a determinar, sería necesaria una segunda muestra, nunca antes utilizada, de datos a predecir, de manera que sea posible estimar los mejores hiperparámetros sin producir sobreajuste. Este subconjunto se denomina *conjunto de validación*. En el caso de utilizar modelos de aprendizaje iterativos, es decir, aquellos que aprenden ajustando sus parámetros paso a paso, los datos de validación se pueden utilizan para monitorear el error de clasificación durante el entrenamiento. Una técnica muy utilizada para evitar el sobreajuste de hiperparámetros consiste en detener el entrenamiento cuando el error de validación alcance un mínimo. Esta técnica se conoce como *detención temprana* [4].

Otro problema potencial en varias técnicas de aprendizaje automatizado es el **subajuste**. Dicho fenómeno se produce cuando el modelo resulta demasiado rígido para poder aprender razonablemente los patrones presentes en los datos de entrenamiento. En

este caso, es importante poder ajustar los hiperparámetros para flexibilizar el modelo a entrenar.

En la Figura 2.1, se pueden apreciar los fenómenos de sobreajuste y subajuste cuando se utilizan polinomios de distintos grados para aproximar datos ruidosos.

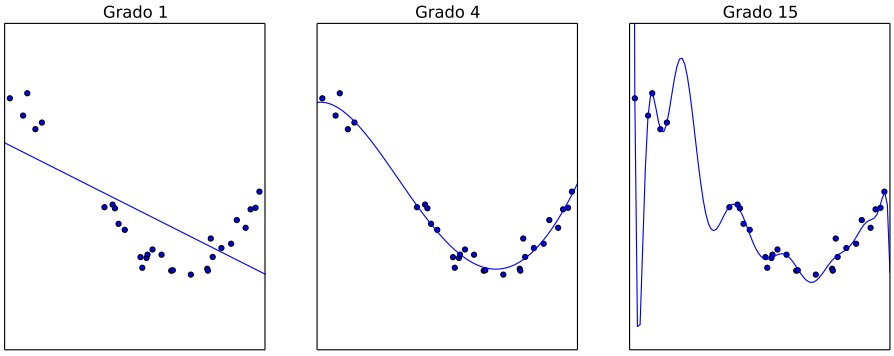


Figura 2.1: Polinomios de distintos grados aproximando datos ruidosos.

Por un lado, el polinomio de 1er grado no resulta en un modelo lo suficientemente preciso, dado los datos que se utilizan para el entrenamiento. Por otro lado, el polinomio de 15avo grado ajusta perfectamente a la gran mayoría de los datos disponibles para entrenamiento. No obstante, al utilizar dicho modelo en nuevos datos, el error de predicción se incrementará significativamente debido al sobreajuste. Finalmente, el polinomio de 4to grado resulta un balance razonable entre las situaciones de subajuste y sobreajuste.

2.1.2.2. Desbalanceo de Datos

El desbalanceo de datos es un problema serio para el aprendizaje automatizado. El mismo se produce cuando la distribución de las etiquetas en los datos para entrenamiento no es uniforme. Típicamente, los datos desbalanceados poseen un gran subconjunto de los mismos etiquetados con una misma etiqueta, la clase mayoritaria. El uso de datos altamente desbalanceados requiere un cuidado adicional cuando se procede a entrenar y evaluar un predictor. Por ejemplo, en el conjunto de datos de detección de fraude mediante transacciones bancarias con tarjetas de crédito [2], las operaciones fraudulentas apenas representan el 0,172 % del total de las operaciones.

Si se intenta entrenar un clasificador para que separe las transacciones fraudulentas y las legítimas, nos encontraremos con un problema. Este severo desbalance afecta al procedimiento de entrenamiento, ya que el clasificador tiende a aprender la información de la clase mayoritaria. Desafortunadamente, utilizar de manera directa datos altamente desbalanceados para el aprendizaje deriva en un clasificador trivial que siempre predice dicha clase. Por ende, en este ejemplo, acabaría por indicar que las transacciones son siempre legítimas y no resultaría útil para detectar fraudes.

Para poder facilitar el correcto aprendizaje de datos desbalanceados, existen dos técnicas muy utilizadas en la literatura del aprendizaje automatizado: el sobremuestreo aleatorio [36] y el submuestreo aleatorio [41]. De esta manera, el sobremuestreo balancea el conjunto de datos, repitiendo aleatoriamente los datos de la clase minoritaria, hasta que su número se equipare con la clase mayoritaria. En cambio, el submuestreo balancea el conjunto de datos, seleccionando aleatoriamente datos de la clase mayoritaria, hasta que su número se equipare con la clase minoritaria. Estas técnicas se realizan exclusivamente en el conjunto de entrenamiento, por lo que los conjuntos de validación y prueba permanecen inalterados.

Adicionalmente, para realizar una correcta evaluación de efectividad de los predictores ya entrenados, cuando los datos están desbalanceados, hay que tener en cuenta que el uso de error de clasificación no es una medida adecuada. Siguiendo el ejemplo, un clasificador trivial sólo tiene el 1 % de error. Con este error tan acotado, se puede concluir erróneamente que el predictor aprendió efectivamente a separar ambas clases. Sin embargo, para poder usar una medida coherente y útil en el proceso de evaluación, es necesario separar los datos previamente por clase y computar el porcentaje de errores en cada clase. Por este motivo, el concepto de matriz de confusión, introducido en la Sección 2.1.1, resulta fundamental para evaluar los predictores en datos altamente desbalanceados.

2.2. Técnicas supervisadas

Una gran variedad de modelos de aprendizaje automatizado se enfoca en resolver el problema de clasificación de manera supervisada. A lo largo de esta tesis, se utilizaron diversas técnicas de aprendizaje supervisado. Las mismas se explican, de manera general, en esta sección.

2.2.1. Regresión logística

La regresión logística [4] es el modelo que utiliza una regresión lineal para separar dos clases con unos pocos parámetros, resultando fácil de entrenar. Este modelo aprende ajustando la probabilidad condicional de un conjunto de categorías presente en los datos, utilizando una transformación afín. Para transformar los valores resultantes de la regresión en una distribución de probabilidades de cada clase, utiliza la función logística:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Para utilizar este modelo para la clasificación de más de dos clases, se suele utilizar un enfoque conocido como *one-vs-rest [4]* que entrena múltiples regresiones logísticas para separar cada clase de las demás.

2.2.2. Redes Neuronales Artificiales

El modelo de regresión logística puede convertirse en no lineal utilizando una o más transformaciones, conocidas como *capas ocultas*, entre los datos de entrada y la transformación afín de la regresión logística. Estas capas utilizan una transformación afín junto con una función no lineal; por ejemplo: la tangente hiperbólica o la rectificación lineal (*ReLu*) [65]. El modelo resultante se conoce como red neuronal artificial o perceptrón multicapa [61]. Dicho modelo aprende mediante una optimización de los parámetros de la red neuronal artificial. Para poder realizar dicha optimización, se suele utilizar el algoritmo de descenso estocástico por el gradiente (o SGD, por sus siglas en inglés) [4]. Dicho algoritmo resulta efectivo para entrenar redes neuronales artificiales con varias capas utilizando un conjunto de datos etiquetados.

La configuración estructural de las redes neuronales (por ejemplo, el número de unidades por capa) y el comportamiento del optimizador para lograr el entrenamiento (por ejemplo, la velocidad o inercia de la minimización) constituyen los hiperparámetros de este tipo de modelos.

Es importante remarcar que estos modelos pueden ser muy susceptibles al sobreajuste. Por esta causa, existen técnicas específicas para evitar este problema. En particular, se destaca la conocida como *dropout* [38]. Dicha técnica ha resultado muy

exitosa en la reducción del sobreajuste, en una gran variedad de aplicaciones [47, 86].

2.2.3. *Random Forest*

Los árboles de decisión son modelos que dividen el espacio de los datos, de manera que sea posible combinar el uso de distintas variables para aprender a clasificar. Estos modelos poseen dos ventajas importantes: resultan fáciles de entrenar y son directamente interpretables en función de las variables de los datos que utilizan. Desafortunadamente, un único árbol de decisión no alcanza para capturar la complejidad de muchos tipos de datos, por lo cual no es un método que se suela utilizar por sí sólo.

Por consiguiente, Leo Breiman definió un método de aprendizaje automatizado llamado *Random Forest* [8]. Este clasificador utiliza un conjunto de árboles de decisión (llamado usualmente *ensemble*) convenientemente entrenados con una selección aleatoria de datos. De esta manera, cada árbol se especializa en ciertos patrones de datos, de modo que sea posible realizar una buena predicción cuando trabajen en conjunto. El resultado final es un modelo robusto [11] muy utilizado en diversos problemas de clasificación, en virtud de la resistencia al sobreajuste y su efectividad en la predicción.

2.3. Técnicas no supervisadas

Otra gran familia de técnicas de aprendizaje automatizado se denomina no supervisada. Estas técnicas no requieren la existencia de etiquetas para aprender patrones en los datos. En esta área, existen dos tareas principales que las técnicas no supervisadas nos permiten abordar: la visualización de los datos, utilizando una adecuada reducción de la dimensionalidad, y el agrupamiento de datos según su similaridad, definida por los patrones de los datos mismos.

2.3.1. Reducción de dimensionalidad y proyección de los datos

Las técnicas de proyección permiten transformar los datos y reducir su dimensionalidad, con el fin de poder visualizarlos (en dos o tres dimensiones) y entenderlos mejor.

2.3.1.1. Análisis de Componentes Principales

El análisis de componentes principales (o PCA, por sus siglas en inglés) es una técnica lineal clásica de transformación de datos y reducción de dimensionalidad, introducida por Karl Pearson, a principios del siglo XX [82]. El PCA se basa en el uso de *valores propios* de la matriz de covarianza de los datos para detectar las dimensiones principales. Las mismas contribuyen significativamente a explicar la varianza de los datos, y por eso son consideradas las más relevantes.

En la práctica, dicha técnica permite transformar, mediante rotaciones y escalado, los datos de su espacio original a un espacio vectorial donde se maximiza la separación de los datos en cada dimensión. Finalmente, el PCA reordena y selecciona las dimensiones con el objetivo de poder ofrecer una proyección en un número reducido de dimensiones (usualmente, se utilizan dos o tres dimensiones). Esta técnica se encuentra implementada en una gran variedad de herramientas de análisis de datos [71, 32] y se utiliza en una multitud de campos de aplicación [78].

2.3.1.2. Análisis de Componentes Semánticos Latentes

El análisis de componentes semánticos latentes (o LSA, por sus siglas en inglés) es una técnica no lineal de transformación de datos, utilizada en procesamiento del lenguaje natural y aprendizaje automatizado, para capturar la semántica de los documentos y sus términos. Fue introducida por Scott Deerwester et al. [19] a finales de los años 80, como una mejora respecto a PCA, de tal forma que resultase más apropiada para los datos producidos a partir de documentos. Dicha técnica utiliza las matrices que representan documentos en cada una de sus filas para reducir su dimensionalidad, pero preservando las relaciones semánticas entre los documentos originales. Para lograrlo, se factoriza la matriz utilizando su descomposición en valores singulares (o SVD, por sus siglas en inglés) y, luego, se reduce su dimensionalidad mediante una aproximación matricial de bajo rango.

2.4. Minería de texto

A lo largo de los capítulos siguientes, serán necesarias no sólo técnicas de aprendizaje automatizado, sino también algunas otras provenientes de la minería de texto. En consecuencia, en esta sección se detallan conceptos básicos de minería de texto junto con un ejemplo de su aplicación en la clasificación de grupos de noticias según

su tópico.

La **minería de texto** es un conjunto de técnicas que se encargan del análisis, el agrupamiento y la clasificación de documentos. En general, se utilizan métodos provenientes del aprendizaje automatizado. Debido a que los texto son secuencias de palabras de longitud variable y no acotada, un problema fundamental, que la minería de texto debe resolver, es el de obtener una representación adecuada de estos datos. Gran parte de las técnicas de aprendizaje automatizado requieren utilizar representaciones de los datos provenientes de un espacio vectorial $\mathbb{R}^n$ donde el valor de n se encuentra fijo.

Luego de obtener una representación vectorial adecuada, es factible aplicar cualquiera de los métodos ya mencionados de aprendizaje automatizado sobre las representaciones vectoriales de los textos para descubrir patrones en los mismos. A continuación, se explicarán algunas técnicas clásicas para la representación de textos provenientes de la minería de datos. En virtud de que esta área es relativamente nueva y está en constante expansión, algunas de las técnicas de preprocesamiento y representación se referirán por sus nombres en inglés, ya que no poseen una traducción al español aceptada.

2.4.1. Bolsa de palabras

La *bolsa de palabras* (o *bag-of-words* en inglés) es una técnica clásica para representar textos como vectores utilizando únicamente la información de las frecuencias de las palabras. En este enfoque, cada texto se reduce a un conjunto con las palabras únicas que utiliza (sin repetición). De esta forma, se pueden representar textos que utilizan n palabras distintas en el espacio vectorial $\mathbb{R}^n$. Al utilizar la *bolsa de palabras*, la información relacionada con el orden de las palabras se descarta. Esta simplificación de los datos permite procesar grandes volúmenes de textos de manera rápida, pero también limita las aplicaciones en las que puede ser utilizado.

Existen dos variantes clásicas para obtener la frecuencia de las palabras en un texto. La primera obtiene la información de la frecuencia utilizando el **conteo directo**: la componente del vector que representa un texto contiene el número de ocurrencias de la palabra correspondiente. Formalmente, para cada $palabra_i$:

$$v_i = \quad conteo(palabra_i, texto)$$

Es importante notar que, en los lenguajes naturales, las palabras más comunes no son necesariamente informativas, sino que sirven de soporte a las construcciones lingüísticas, por ejemplo: los artículos o las preposiciones. Por lo tanto, se ideó una variante de la bolsa de palabras utilizando la información de la **frecuencia inversa**. Así, las palabras menos comunes reciben valores más grandes. Formalmente, sea n el número de palabras distintas, para cada $palabra_i$:

$$v_i = \quad \frac{n}{conteo(palabra_i, texto) + 1}$$

Ambas técnicas son ampliamente utilizadas, y su uso depende principalmente de las propiedades de los datos y de la aplicación concreta.

2.4.2. *Word2vec*

Word2vec es una técnica de preprocesamiento y vectorización diseñada para aprender las representaciones vectoriales continuas [57] de cada palabra en grandes corpus de texto. Se seleccionó esta técnica porque fue utilizada con éxito en una variedad de aplicaciones de minería de texto [94, 73]. Adicionalmente, una implementación eficiente de esta técnica está disponible en forma de código libre, gracias al trabajo de investigadores e ingenieros de software de *Google* [58].

2.4.3. *FastText*

FastText es una técnica de preprocesamiento y vectorización presentada recientemente [43]. La misma fue diseñada para mejorar substancialmente los resultados de un clasificador lineal, extendiendo la representación de bolsa de palabras con información adicional. Se seleccionó dicha técnica porque proporciona una representación vectorial semántica de manera rápida y eficiente. Otro detalle importante es que se encuentra disponible una implementación muy eficiente de código libre, creada por investigadores de *Facebook* [25].

2.4.4. Ejemplo: Minería de textos de grupos de noticias

A continuación, se introduce un ejemplo de uso de técnicas de minería de textos y aprendizaje automatizado sobre datos de grupos de noticias en inglés, catalogados según su temática. Para este ejemplo, se seleccionaron los mensajes de cuatro grupos de noticias del conjunto de datos de *20news* [45]:

- clase 1: *alt.atheism*
- clase 2: *talk.religion.misc*
- clase 3: *comp.graphics*
- clase 4: *sci.space*

Estos grupos de noticias fueron utilizados como categorías para cada uno de los 3.387 mensajes de dichos grupos. Para este ejemplo, primero se comparan las distintas formas de representaciones de los textos utilizando variantes de la bolsa de palabras junto con distintas tareas de aprendizaje no supervisado para reducir la dimensionalidad y presentar el resultado en un gráfico bidimensional. Dichos gráficos exponen cómo los datos se agrupan naturalmente según su similitud. Es notable que la existencia de distintos grupos de noticias se manifiesta más allá de su etiqueta en el conjunto de datos. Por esta razón, este tipo de técnicas resultan útiles en el análisis de datos. Finalmente, se utilizaron distintos algoritmos de aprendizaje supervisado para entrenar diversos clasificadores que puedan predecir el grupo al cual pertenece un determinado texto. De esta manera, queda constituido un sencillo clasificador de temas de grupos de noticias entrenado de acuerdo a las palabras presentes en los mensajes.

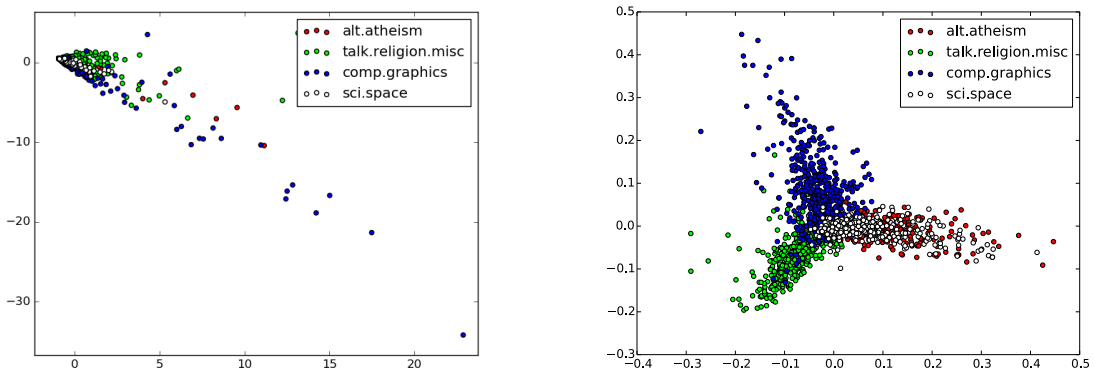
Para la implementación de este ejemplo, se utilizó *scikit-learn* [71], un robusto paquete de software de código libre para realizar una variedad de tareas de aprendizaje automatizado que incluye preprocesamiento, entrenamiento y evaluación de predictores.

Las técnicas más modernas de preprocesamiento de textos en conjuntos de datos pequeños como *20news* no resultan efectivas, por eso no se las incluyen en este ejemplo. Usualmente, se requieren al menos 100.000 documentos para obtener buenas representaciones utilizando técnicas como *word2vec* y *fastText*.

Preprocesamiento. Antes de comenzar con las técnicas de minería de datos, se filtraron las líneas correspondientes a los encabezados y remitentes de cada texto. Por consiguiente, el análisis de los textos se centra en las palabras utilizadas en el cuerpo de los mensajes, ya que la información de los encabezados y remitentes está fuertemente correlacionada con los autores de cada grupo de noticias y puede provocar un sesgo en los resultados.

Para representar los textos como vectores, sería impráctico utilizar todas las palabras del lenguaje, ya que el número de palabras distintas en los textos en inglés es muy alto: por ejemplo, el diccionario de *Oxford* contiene 171.476 entradas. De esta manera, para cada texto, se consideraron hasta 10.000 palabras distintas.

Reducción de dimensionalidad. Las figuras a continuación indican el uso de las técnicas de representación de textos, utilizando las dos variantes de la bolsa de palabras: conteo directo y frecuencia inversa. Los gráficos fueron generados reduciendo la dimensionalidad con el PCA.



(a) Utilizando el conteo directo.

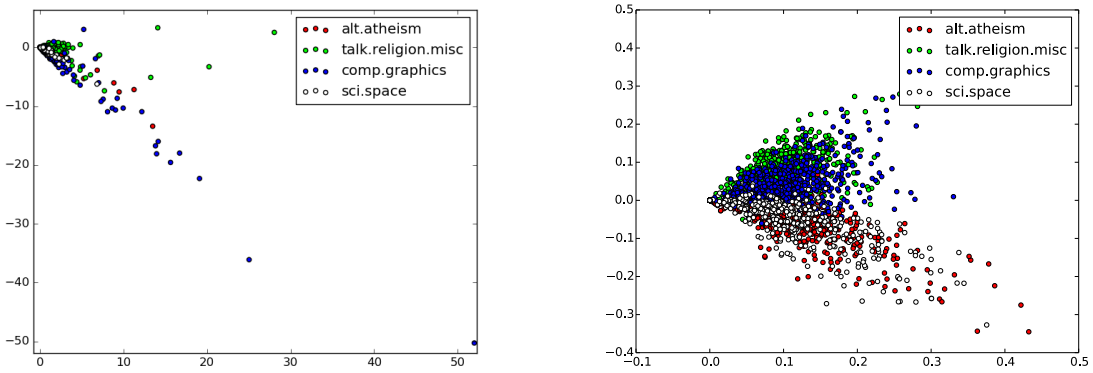
(b) Utilizando la frecuencia inversa.

Figura 2.2: Textos de noticias visualizados con PCA.

Es importante recordar que las técnicas de reducción de dimensionalidad no requieren el uso de etiquetas para producir resultados. A pesar de esto, en la Figura 2.2b se muestra que el uso de la frecuencia inversa en la representación de bolsa de palabras preserva razonablemente las etiquetas de los distintos grupos del cual se extrajeron los textos. Se puede apreciar también una fuerte superposición de las

categorías *alt.atheism* y *sci.space.* Por otro lado, en la Figura 2.2a, se aprecia que las categorías de los textos se encuentran mucho más superpuestas y es notable la existencia de valores atípicos (o *outliers*, según su denominación en inglés).

En contraste, en las Figuras 2.3a y 2.3b se utilizó el LSA para reducir la dimensionalidad.



(a) Utilizando el conteo directo.

(b) Utilizando la frecuencia inversa.

Figura 2.3: Textos de noticias visualizados con LSA.

Al igual que en los gráficos anteriores, la Figura 2.3b revela que el uso de la frecuencia inversa en la representación de bolsa de palabras preserva mejor las etiquetas de los distintos grupos del cual se extrajeron los textos, que el conteo directo de la Figura 2.3a. Sin embargo, comparando los resultados en todas las figuras presentadas anteriormente, se evidencia una mejor separación por temas en la reducción utilizando el PCA.

Aprendizaje supervisado: Luego, utilizando un clasificador, es posible entrenar con los datos de los grupos de noticias y predecir la temática de un nuevo texto, si se lo procesa de la misma manera que los datos de entrenamiento. En estos experimentos, se han utilizado los datos procesados con frecuencia inversa por exhibir propiedades útiles para la separación de los textos según los tópicos de cada grupo de noticias. No se aplicaron técnicas de rebalanceo de datos, debido a que los datos originales no presentaban un desbalance significativo.

Para esta etapa, se hicieron uso de todos los clasificadores previamente presentados: *random forest*, regresión logística y redes neuronales de una y dos capas ocultas.

	Clase 1	Clase 2	Clase 3	Clase 4
Clase 1	**201**	21	30	67
Clase 2	22	**320**	29	18
Clase 3	54	34	**268**	38
Clase 4	121	23	18	**89**

(a) *Random Forest con* error de prueba de 36 %

	Clase 1	Clase 2	Clase 3	Clase 4
Clase 1	**202**	10	31	76
Clase 2	10	**345**	19	15
Clase 3	18	20	**329**	27
Clase 4	62	8	18	**163**

(b) Regresión logística con un error de prueba de 24 %.

	Clase 1	Clase 2	Clase 3	Clase 4
Clase 1	**207**	13	38	61
Clase 2	10	**354**	24	1
Clase 3	25	20	**347**	2
Clase 4	73	12	24	**142**

(c) Redes neuronales artificiales de una capa con un error de prueba de 23 %.

	Clase 1	Clase 2	Clase 3	Clase 4
Clase 1	**199**	7	40	73
Clase 2	6	**350**	25	8
Clase 3	18	15	**357**	4
Clase 4	57	15	21	**158**

(d) Redes neuronales artificiales de dos capas con un error de prueba de 22 %.

Tabla 2.2: Matrices de confusión de la predicción de los grupos de noticias utilizando distintos métodos de aprendizaje supervisado.

Los resultados en forma de matrices de confusión se muestran en las Tablas 2.2a, 2.2b, 2.2c y 2.2d. Estos se elaboraron utilizando los datos de prueba, nunca antes utilizados para clasificación o validación, especificados en el conjunto de datos original [45]. Los experimentos indican que el clasificador más preciso resultó ser una red neuronal artificial de dos capas ocultas, alcanzando el 22 % de error de prueba promedio entre todas las clases.

Capítulo 3

Seguridad en programas

3.1. Conceptos fundamentales

Antes de comenzar a definir de manera precisa los distintos tipos de fallos y vulnerabilidades en programas que se investigaron en la presente tesis, es importante establecer algunos conceptos generales sobre la seguridad de los sistemas informáticos que serán utilizados en los siguientes capítulos.

3.1.1. Atacantes de un sistema informático

Los atacantes de un sistema pueden ser definidos como agentes externos que intentan alterar la integridad, confidencialidad o disponibilidad de un sistema informático. Los sistemas informáticos ponen a disposición de los usuarios sus recursos propios, como los ciclos de procesamiento y la memoria temporal o permanente, aunque con ciertas restricciones. Dentro de este contexto, un atacante intentará utilizar todos los recursos de un sistema bajo su control para producir un comportamiento disruptivo. Esta suposición introduce un concepto importante: la controlabilidad de recursos por parte de un atacante. Por este motivo, es primordial delimitar claramente cuáles son las restricciones en las entradas y los recursos de un sistema que están bajo el control de un atacante, para poder establecer los posibles ataques.

Adicionalmente, existen distintos roles que un atacante puede asumir para atacar un sistema. Por ejemplo, en el contexto de un sistema multiusuario, un usuario puede tener acceso a ciertos permisos pero, bajo ciertas circunstancias, puede realizar una operación reservada solo para los administradores. En este caso, un usuario malicioso

puede realizar un ataque con dichos privilegios.

En particular, se hará enfoque en la seguridad de los programas que se ejecutan localmente en un sistema operativo, por ejemplo: programas para procesar texto, audio o video; o visualizar páginas web. En este escenario, un posible atacante tendrá el control de las entradas más comunes de un programa ejecutable, como la interfaz de línea de comandos, los archivos provistos por el usuario o la entrada de teclado estándar.

3.1.2. Errores de programa

Los programas que se utilizan no siempre se comportan de la manera esperada por parte del usuario final o, incluso, de su programador. Dichos comportamientos inesperados se denominan **errores o fallos de programa**. Prácticamente, todos los programas que una computadora ejecuta contienen errores.

En primera instancia, para poder formalizar la noción de error, se definirá el concepto de ejecución de un programa. La ejecución del mismo puede terminar normalmente o con un fallo: sea un programa P y una entrada I, notamos $P(I)$ a la ejecución del programa P utilizando la entrada I. En general, los programas pueden ser analizados únicamente desde el punto de vista de sus entradas y salidas, debido a que los detalles de la implementación son demasiado complejos o simplemente irrelevantes. Dicha función se nota $P : \mathbb{I} \to \mathbb{O}$, donde $\mathbb{I}$ designa el espacio de entrada de un programa y $\mathbb{O}$ designa al conjunto de los estados finales de la ejecución, definido como la unión de los siguientes valores:

$$\texttt{Salida} \,|\, \texttt{Aborto} \,|\, \texttt{Fallo}$$

donde el valor `Salida` indica que el programa ha terminado sin errores. Por otro lado, un `Aborto` indica que la ejecución ha terminado abruptamente a causa de algún error grave, pero previsto en el código. Finalmente, un `Fallo` se produce cuando la ejecución de un programa termina de manera abrupta por una circunstancia no prevista por el programador.

Atributos de Ejecución: Se introduce esta noción para hacer referencia a los componentes específicos del estado de ejecución de un programa. Dado un programa

P y una entrada I, denotamos $P_{at}(I)$ al valor de un atributo at al final de la ejecución fallida. Los atributos pueden ser registros específicos de la arquitectura, donde el programa se ejecuta, o direcciones de memoria. Por ejemplo, en muchas arquitecturas de hardware, el registro que contiene el puntero a la próxima instrucción se denomina como `IP`, por sus siglas en inglés. En ese caso, se notará $P_{IP}(I)$ al valor del puntero de instrucción al final de la ejecución fallida.

3.1.3. Vulnerabilidad

Un error de programa que podría ser utilizado por un atacante para obtener algún provecho del sistema se denomina **vulnerabilidad**. Existe un amplio espectro de vulnerabilidades conocidas en los programas, dependiendo de cuál sea la tarea que el programa realice y en qué contexto. Entre los ejemplos comunes, figuran:

- Elevación de permisos: un atacante utiliza un usuario sin privilegios para obtener permisos de administrador sin ser debidamente autorizado (por ejemplo, mediante una contraseña del sistema).
- Ejecución arbitraria de código: un atacante inyecta datos en la memoria que son interpretados como instrucciones por la CPU.
- Filtración de datos: un atacante logra obtener datos internos del sistema que deben mantenerse en secreto (por ejemplo, la contraseña de un usuario o los números de tarjetas bancarias).
- Denegación de servicio: un atacante logra detener o demorar el procesamiento normal de un sistema, afectando su disponibilidad a otros usuarios.

Aunque no hay una escala aceptada globalmente sobre la gravedad de las vulnerabilidades en sistemas informáticos, es importante destacar que la ejecución arbitraria de código se considera como la vulnerabilidad más grave, en vista de la flexibilidad que su uso le da al atacante. La misma permite realizar potencialmente cualquier tarea en el sistema atacado y, por lo tanto, puede utilizarse fácilmente para lograr los efectos de otras vulnerabilidades como, por ejemplo, producir una filtración de datos o una denegación de servicio.

Existen proyectos de documentación de libre participación enfocados en catalogar y ejemplificar distintos tipos de errores y vulnerabilidades comunes en el software. Uno de los más importantes es la enciclopedia electrónica *Community Weakness Enumeration* [63], conocida por sus siglas en inglés como CWE. Utilizaremos esta base de conocimiento para referirnos a las definiciones más aceptadas de algunas vulnerabilidades comunes.

3.1.4. *Exploit*

Un ***exploit*** es un caso de prueba o herramienta diseñada para aprovecharse de una vulnerabilidad en un sistema informático concreto. No todas las vulnerabilidades pueden ser aprovechadas de manera práctica para el beneficio de un atacante. En estos casos, resulta imposible crear un *exploit* y la importancia de la vulnerabilidad se reduce.

Sin importar la técnica empleada para aprovecharse de una determinada vulnerabilidad, la ejecución arbitraria de código tiene una estructura común fundamental: dado un programa P, un atacante necesita encontrar una entrada particular $\tilde{I}$, donde P falle en un estado, donde el puntero de instrucciones tenga un valor concreto X. Podemos formular la tarea de generación de *exploits* como un problema de inversión de atributos de ejecución: $P_{IP}(\tilde{I}) = X$.

Inversión de Atributos: Dado un programa P, un espacio de entradas $\mathbb{I}$ de dicho programa, un atributo at y un valor de dicho atributo X, el problema de inversión de atributos se enfoca en encontrar una entrada particular $\tilde{I} \in \mathbb{I}$ tal que:

$$P_{at}(\tilde{I}) = X$$

Cualquier solución a dicha ecuación se denota $P_{at}^{-1}(X)$. Usando esta definición, podemos formalizar el concepto de *exploit* que nos permita una ejecución remota del código: $P_{IP}^{-1}(\&shellcode)$. En otras palabras, el atacante intenta encontrar una entrada de P que cause que el puntero de instrucción apunte al código inyectado por el atacante. Este código se denomina usualmente como *shellcode*. De esta manera, este tipo de *exploit* se completa redireccionando el `IP` a la dirección donde se encuentra el *shellcode*.

3.2. Errores y vulnerabilidades comunes de los programas

3.2.1. Desbordamiento de *buffer*

La memoria es un recurso fundamental en el software. Los sistemas operativos modernos permiten un manejo explícito de la memoria y se orientan hacia las regiones contiguas de la misma, comúnmente denominadas ***buffers***. Estas regiones se reservan y liberan mediante funciones provistas por el sistema operativo. Adicionalmente, se distinguen dos regiones generales de memoria básica donde se puede localizar la memoria reservada y utilizada por un proceso particular. Por un lado, existe **la pila**, donde reside la memoria local, empleada por rutinas o funciones. En contraste, en **el *heap*** es donde se almacena la memoria reservada dinámicamente. Cada proceso posee su propia memoria y, por lo tanto, sus propias regiones de pila y *heap*.

Históricamente, el manejo de memoria ha sido un foco de vulnerabilidades por medio de operaciones inseguras de memoria, donde se lee o escribe fuera de los *buffers* reservados [67, 76]. Tal fallo se conoce como **desbordamiento de *buffer***. La directiva CWE-120 define los desbordamientos de *buffer* como:

> "Una condición en la ejecución de un programa donde se escriben datos pasado el final o antes del principio de un determinado *buffer*"

Este tipo de errores se manifiesta comúnmente en lenguajes de programación que permiten un manejo explícito de la memoria como ensamblador, C o C++. En esta sección, se presentará un sencillo ejemplo de cómo este fallo puede afectar la seguridad de un sistema de permisos multiusuario. En la Figura 3.1 se muestra el código C para ejemplificar este tipo de errores.

```
1  struct info_usuario {
2     char usuario[32];
3     int  permiso;
4  };
5
6  char *id_usuario(char *nombre) {
7    char *usuario = (char*) malloc(32);
8    strcpy(usuario, "usr:");
9    if (!nombre || strlen(nombre) > 32)
10     return NULL;
11   return strcat(usuario, nombre);
12 }
13
14 int es_admin(char *id) {
15   struct info_usuario info;
16   info.permiso = 0;
17   strcpy(info.usuario, id);
18   if (info.usuario[0] == 'a' &&
19       info.usuario[1] == 'd' &&
20       info.usuario[2] == 'm' &&
21       info.usuario[3] == ':')
22     info.permiso = 1;
23   return info.permiso;
24 }
25
26 int main(int argc, char *argv[]) {
27   char *id = id_usuario(argv[1]);
28   if (id) {
29     int resultado = es_admin(id);
30     printf("%s %s es admin\n", id, resultado ? "" : "no");
31   }
32   return 0;
33 }
```

Figura 3.1: Código C con una vulnerabilidad de desbordamiento de *buffer*.

En dicho programa, se presenta un sistema multiusuario muy simplificado que asocia a todo usuario con un identificador y un número entero que representan los permisos que este posee. Los usuarios administradores tienen un identificador con

el prefijo `adm:` y un permiso valuado en 1. En contraste, los usuarios sin privilegios tienen un identificador con el prefijo `usr:` y un permiso valuado en 0.

Además, resulta esencial que los usuarios sin privilegios solo puedan obtener su identificador utilizando la función `id_usuario` (Líneas 6–12). En esta función, el programador ha colocado una condición para revisar el tamaño del identificador, de manera que retorne error si el identificador es demasiado largo (Líneas 9–10). No obstante, bajo ciertas condiciones, es posible escribir más allá de la memoria reservada para el identificador (Línea 11).

Una vez obtenido el identificador de usuario, la función `es_admin` debe determinar si el usuario es administrador o no, revisando si el prefijo del identificador resulta `adm:` (Líneas 14–24). Esta función solo debe utilizarse bajo la condición de que los identificadores tengan, a lo sumo, 32 bytes.

En este sencillo ejemplo, se muestra únicamente el código correspondiente a los usuarios sin privilegios. Por esta razón, el programa siempre debería informar que el usuario no es administrador.

Explotabilidad. Se hará foco, especialmente, en errores relacionados con el manejo inseguro de memoria. Todos los sistemas operativos modernos protegen la memoria de los procesos para que estos no puedan interferir entre sí. De este modo, en el caso de un error en un programa, este no podrá producir fallos de memoria en los demás al escribir memoria sin control. No obstante, en el caso de un programa complejo con varios módulos de software funcionando en el mismo espacio de memoria, el fallo de un solo componente puede ocasionar que el programa completo termine erróneamente.

En particular, un desbordamiento de *buffer* puede tener varios efectos en un programa en ejecución como, por ejemplo, hacerlo abortar si intenta escribir un área protegida de otro programa, o marcada como solo lectura; también puede afectar visiblemente la lógica del mismo programa, haciéndolo entrar en un bucle sin salida, o corrompiendo información interna. En algunos casos, puede no tener efectos visibles, si las áreas de memoria están suficientemente separadas. La importancia real de este tipo de vulnerabilidades es la posibilidad de ser utilizada por un atacante para producir la ejecución arbitraria de código [67].

A pesar de que en los últimos años, muchos sistemas operativos han adoptado

protecciones y mitigaciones efectivas contra esta familia de vulnerabilidades (forzando que el código se ejecute únicamente desde la memoria de solo lectura [88], por nombrar un ejemplo), este tipo de errores sigue siendo relevante, a causa de las técnicas de explotación avanzada, tal como ROP [76].

Siguiendo con el ejemplo, se analizará cómo utilizar un desbordamiento de *buffer* para cambiar el estado del programa, de manera que permita identificar a un usuario como administrador. En circunstancias normales, un programa o el sistema operativo pueden consultar si un usuario tiene permisos de administrador o no, de la siguiente manera:

```
$ ./es_admin juan
usr:juan no es admin
$ ./es_admin juaaaaaaaaaaaaaaaaaaaaaaaaaa ... aaaaaaaan
$
```

En este primer caso de prueba, el usuario `juan` no resulta estar en el grupo de los administradores. En la siguiente ejecución, el nombre de usuario resulta anormalmente largo y el programa no responde la consulta por resultar inválida. No obstante, si un atacante puede controlar efectivamente su nombre de usuario, puede sobreescribir la variable `info.permiso` con un valor no nulo y, por lo tanto, elevar sus permisos a administrador:

```
$ ./es_admin aaaaaaaaaaaaaaaaaaaaaaaaaaaa      # 28 caracteres
usr:aaaaaaaaaaaaaaaaaaaaaaaaaaaa no es admin
$ ./es_admin aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  # 34 caracteres
usr:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa es admin
```

Tal como se aprecia en este sencillo ejemplo, el atacante aprovecha el desbordamiento de *buffer* para escribir más allá de la región de memoria que corresponde, alterando información importante del sistema.

Este tipo de errores no será detectado por las protecciones más comunes, tales como la protección de memoria de pila implementada por varios compiladores de C. Solamente en el caso de que el código se compile con la opción `-fsanitize=address`, se incluirá una protección efectiva para detectar este tipo de errores y abortar la ejecución en el momento que suceda el primer desbordamiento:

```
==24350== ERROR: AddressSanitizer: unknown-crash on address 0x60060000efe4
WRITE of size 29 at 0x60060000efe4 thread T0
...
```

Sin embargo, los programas no pueden utilizar esta protección en los sistemas de producción, ya que incurre en una alta penalidad de eficiencia al requerir la verificación de todas las lecturas y escrituras en la memoria antes de que se realicen.

3.2.2. Desbordamiento de entero

El uso de aritmética modular de enteros también puede resultar en errores y vulnerabilidades en los programas, si no se lo realiza cuidadosamente. Estos casos son producidos por condiciones imprevistas por los programadores cuando las operaciones aritméticas resultan en números más grandes o más pequeños de lo que los registros pueden representar. Estos errores se conocen como ***desbordamiento*** (*overflow*) y ***subdesbordamiento*** (*underflow*) **de enteros**. La directiva CWE-190 define el desbordamientos de números enteros como:

> "Una condición ocurrida en la ejecución de un programa donde un valor entero que se incrementa resulta demasiado grande para la representación asociada con tal valor y la lógica del programa asume que el resultado siempre será mayor que el valor original"

A continuación se presenta un ejemplo con aritmética entera. Es bien sabido que un entero sin signo de 16-bits no podrá representar un número superior a 65535. Entonces, si $x_{16} = 65520$ (`0xFFF0`), la siguiente operación en aritmética entera de 16-bits desborda la variable y_{16}:

$$y_{16} = x_{16} + 32$$

Luego de esta asignación, la variable y_{16} contendrá 16 (`0x10`). Si alguna parte del código que se ejecuta posteriormente utiliza el resultado para alguna operación de memoria, el programa puede ser vulnerable a un desbordamiento de enteros. Puesto que la implementación de controles en cada operación aritmética puede incurrir en una pérdida de velocidad sustancial, los lenguajes de nivel bajo o medio no implementan este tipo de controles de manera obligatoria. En esta sección, se presenta

un sencillo ejemplo de cómo este fallo afecta a un programa. Para ejemplificar ese tipo de errores, se utilizará un código de ejemplo de lo que podría ser un sistema que procesa imágenes a color. En la Figura 3.2 se muestra el código C, que reserva memoria para imágenes a color:

```
1 char *crear_imagen(unsigned alto, unsigned ancho) {
2   unsigned total = 3*alto*ancho+16;
3   if (alto*ancho == 0)
4     return NULL;
5   return (char*) malloc(total);
6 }
```

Figura 3.2: Código C con una vulnerabilidad de desbordamiento de entero.

En este ejemplo, se muestra únicamente una función que reserva la memoria para contener imágenes a color, de tamaño $N \times M$. Estas imágenes se implementan como arreglos de caracteres: debido a que cada píxel se representa por una tripleta de bytes de color (por ejemplo, utilizando RGB, por las siglas de los colores rojo, verde y azul, en inglés), es necesario reservar un arreglo de al menos $3 * N * M$ bytes. Adicionalmente, se agregan 16 bytes extra para almacenar información suplementaria de la imagen, como sus dimensiones de alto y ancho.

En el código de `crear_imagen`, el cálculo del tamaño de memoria a reservar se encuentra en la Línea 2. El programador implementa un sencillo control de tamaño para evitar reservar imágenes de tamaño nulo en las Líneas 3–4. Finalmente, la Línea 5 reserva efectivamente la memoria, si esta estuviera disponible en el sistema operativo. No obstante, `crear_imagen` no posee ninguna verificación de desbordamiento de enteros, en el caso de que la operación aritmética utilizada para calcular el tamaño se desborde. Particularmente, si se invoca a esta función con determinados parámetros de alto y ancho, por ejemplo:

```
char *imagen = crear_imagen(65534, 21846)
```

el resultado será un puntero a *buffer* de 12 bytes de tamaño, en vez de contener 4,294,967,308 bytes (∼4GB). Solamente en el caso de que el código se compile con la opción `-fsanitize=signed-integer-overflow`, se incluirá una advertencia para detectar este tipo de errores:

```
runtime error: unsigned integer overflow: 4294967292 + 16 cannot be
    represented in type 'unsigned int'
```

Sin embargo, los programas no pueden utilizar esta protección en los sistemas de producción, ya que incurre en una penalidad de eficiencia al requerir la verificación de los resultados de todas las operaciones aritméticas antes de que se realicen.

Explotabilidad. En general, para aprovecharse de un desbordamiento de enteros, es necesario utilizar algún otro fallo. Por ejemplo, los desbordamientos de enteros suelen crear condiciones propicias para las vulnerabilidades de manejo de memoria, ya que permiten alterar los tamaños de los *buffers* para hacerlos más pequeños de lo esperado. De esta manera, la memoria leída o escrita por un programa resulta en una operación insegura.

En la próxima sección, se describirá en detalle cómo un desbordamiento de enteros afecta al código de manera que permita la ejecución arbitraria de código por parte de un atacante.

3.2.3. Una vulnerabilidad en un conversor de archivos *PowerPoint*

`ppthtml` es una utilidad de línea de comandos de código libre para convertir archivos de *Microsoft PowerPoint*™ (ppt) en páginas web (html). La Figura 3.3 muestra un fragmento de código fuente de `ppthtml` relacionado con una grave vulnerabilidad, conocida como CVE-2013-4565 [77], que permite la ejecución arbitraria de código. Se utilizará dicho fragmento para demostrar cómo interactúan varios errores presentados anteriormente en este programa para conformar dicha vulnerabilidad.

La vulnerabilidad ocurre durante la ejecución de `__OLEdecode`, la función responsable de decodificar el *Object Linking and Embedding*, es decir, la información interna de un archivo de *Microsoft PowerPoint*™. Esta función recibe un nombre de archivo (por el primer argumento). Dicha función lee el número de bloques del archivo (Líneas 15-16), reserva un espacio de memoria en el *heap* (Línea 17) y finalmente copia el contenido del archivo en ella.

El *exploit* utilizan varios pasos y distintos tipos de vulnerabilidades para lograr la ejecución arbitraria de código:

1. **Desbordamiento de enteros.** Un desbordamiento de enteros se puede producir en la Línea 17, cuando el programa reserva memoria en el *heap*, utilizando la función `malloc`. El argumento de tamaño de la función `malloc` es directamente controlado por un posible atacante (a través de un archivo ppt para convertir) y, utilizando un valor suficientemente grande, su valor se desborda. Por ejemplo, en el *exploit* de estas vulnerabilidades, `num_bbd_blocks = 0x4` y `num_xbbd_blocks = 0x1010101`, reservando solamente `0x2020a00` bytes (aproximadamente, 32MB) para la variable `BDepot` en vez de 0x202020a00 bytes (aproximadamente, 8GB).

2. **Desbordamiento de *buffer* Nº 1.** Un desbordamiento de *buffer* es posible en el primer bucle del código (Líneas 19–24), debido al desbordamiento de enteros mencionado anteriormente. Esta vulnerabilidad no se utilizará para conformar el *exploit*, pero sin embargo está presente.

3. **Desbordamiento de *buffer* Nº 2.** Durante el ciclo anidado (Líneas 26–34), la función `__OLEDecoder` intenta copiar `0x1010101`×`0x200` bytes (aproximadamente, 8GB) al *buffer* de 32MB de tamaño `BDepot`. Esta operación resulta en un desbordamiento de `BDepot`.

Es esperable que, al realizar esta serie de operaciones erróneas, el programa finalice inmediatamente con un error de segmentación. Sin embargo, bajo ciertas condiciones, esta serie de errores puede ser suficiente para controlar el registro IP del sistema y ejecutar código arbitrario. El *exploit* se completa mediante la escritura de la estructura de control de hilos (o TCB, por sus siglas en inglés). Normalmente, esta información se escribe antes de iniciar el programa y contiene datos que el sistema operativo utiliza al ejecutar el hilo. En particular, la TCB contiene los punteros hacia funciones que se utilizan para ejecutar llamadas al sistema. Debido a que esta vulnerabilidad sobreescribe gran parte de la memoria, un atacante puede utilizar la TCB para lograr ejecución arbitraria de código.

Ejecución arbitraria de código Finalizada la sobreescritura de la memoria (incluyendo la TCB en uso), el programa sigue su marcha. Sin embargo, cuando intente

utilizar nuevamente la llamada a `fseek` (Línea 30), el sistema operativo consultará la TCB, ya que esta función requiere el uso de una llamada del sistema. El atacante reemplazó el puntero de llamada del sistema utilizado por uno propio, que apunta al código malicioso. De esta manera, es posible inducir un control completo del `IP` mediante los datos de entrada de `ppthtml`, y forzar la ejecución de código arbitrario. Para poder realizar la ejecución de código arbitrario en sí misma, solo resta inyectar el código deseado en algún lugar de la memoria y apuntar el `IP` a dicha dirección de memoria para lograr la ejecución cuando se llame a `fseek`.

3.2.4. Observaciones importantes

Incluso para un grupo de expertos en seguridad informática, entender esta vulnerabilidad y cómo explotarla no es una tarea trivial[1]. El *exploit* detallado aquí muestra algunos de los desafíos en el proceso de encontrar y catalogar errores de seguridad críticos. En especial, los aspectos claves de este proceso son:

- **La lógica de la aplicación y el sistema operativo es importante.** Aprovecharse de las vulnerabilidades presentes en `ppthtml` es un proceso delicado, ya que involucra múltiples vulnerabilidades consecutivas: un desbordamiento de enteros y *buffers* en distintas líneas de código. También es necesario contar con un gran número de intrincados detalles técnicos sobre el manejo de memoria y el funcionamiento de la TCB.

- **La automatización es necesaria.** Aunque la inspección manual por parte de expertos puede revelar la existencia de vulnerabilidades cuando las aplicaciones no son muy complejas, este enfoque es complicado y costoso. Auditar centenares o miles de aplicaciones es impráctico sin utilizar técnicas y herramientas especializadas que analicen el comportamiento de los programas. Por ejemplo, en este caso, `ppthtml` espera encontrar archivos estructurados como presentaciones de *PowerPoint*™. Sin cumplir esta condición, resulta imposible alcanzar el sector de código vulnerable.

[1] El autor pasó un día entero intentando entender este código junto a estudiantes e investigadores de la universidad de Carnegie Mellon, considerada una de las mejores en el mundo en el área de seguridad informática.

- **Explorar condiciones inseguras de los programas puede conducir a la vulnerabilidad.** El desbordamiento de enteros de la Línea 17 es solo el primer paso para aprovecharse de estas vulnerabilidades. Se suceden varios desbordamientos en memoria y una sobrescritura de punteros a función. Muchas veces los expertos (y algunas herramientas), detienen su análisis del programa cuando se encuentran con la primera falla. Usualmente, para poder encontrar vulnerabilidades, es necesario explorar el estado del programa, especialmente cuando es erróneo, para detectar si la falla puede ser aprovechada.

En el próximo capítulo, exploraremos estas ideas con el fin de abordar la detección de fallos y vulnerabilidades.

```
1  static char* Block;
2  static char* Blockx;
3  static FILE* input;
4  static char* BDepot;
5
6  int __OLEdecode( char* filename, ... )
7  {
8    char* s;
9    unsigned i, j;
10   unsigned num_bbd_blocks, num_xbbd_blocks;
11   input = fopen( filename, "rb" );
12   Block = (char*) malloc( 0x200 );
13   fread( Block, 0x200, 1, input );
14   rewind( input );
15   num_bbd_blocks = (unsigned) (Block + 0x2c);
16   num_xbbd_blocks = (unsigned) (Block + 0x48);
17   BDepot = (char*) malloc( 0x200*(num_bbd_blocks+
       num_xbbd_blocks) );
18   s = BDepot;
19   for ( i = 0;
20         i < MIN( num_bbd_blocks, 0x200/4 - 19 );
21         i++ ) {
22     fread( s, 0x200, 1, input );
23     s += 0x200;
24   }
25   Blockx = (char*) malloc( 0x200 );
26   for ( i = 0; i < num_xbbd_blocks; i++ ) {
27     fseek( ... );
28     fread( Blockx, 0x200, 1, input );
29     for ( j = 0; j < 0x200 / 4 - 1; j++ ) {
30       fseek( ... );
31       fread( s, 0x200, 1, input );
32       s += 0x200;
33     }
34   }
35 }
```

Figura 3.3: Una función vulnerable de `ppthtml` simplificada.

Capítulo 4

Detección de fallos y vulnerabilidades

4.1. Estado del arte

A lo largo de los últimos años, distintos procedimientos de detección de vulnerabilidades (abreviado en esta tesis como PDV) han sido propuestos en la literatura de la seguridad informática. Cada una de estas metodologías poseen distintos requerimientos, costos computacionales y sesgos al identificar vulnerabilidades en software. En este capítulo se destacan los PDV más utilizados, propuestos por varios autores, así como las nuevas herramientas que influyeron en el desarrollo, en esta tesis, de metodologías novedosas para detectar fallos y vulnerabilidades.

4.1.1. Análisis estático

El análisis estático consta de un conjunto de técnicas para estimar el comportamiento del código de los programas para identificar los valores posibles en cada uno de sus estados, sin requerir su ejecución. En particular, existen metodologías y herramientas específicas para la detección de errores y vulnerabilidades en programas que utilizan análisis estático. Su ventaja principal es proveer un análisis coherente de un conjunto de ejecuciones posibles de un programa. No obstante, estas sufren de limitaciones intrínsecas: ninguna de estas técnicas puede utilizar información sólo disponible durante la ejecución (por ejemplo, direcciones de memoria específicas o datos provistos por el usuario).

Históricamente, las herramientas de análisis estático fueron utilizadas para probar

la ausencia de errores en un programa [16][17]. Estas técnicas fueron particularmente efectivas en el contexto de ciertas aplicaciones de dominio específico, por ejemplo sistemas embebidos o software aeroespacial. Sin embargo, el uso del análisis estático para analizar software de propósito general, tal como se plantea en esta tesis, es difícil y costoso en función de la memoria utilizada y el tiempo de procesamiento. Esto es esencialmente causado por las sobreaproximaciones necesarias para calcular las diferentes ejecuciones que un programa podría realizar, manteniendo la coherencia del análisis. Este fenómeno reduce notoriamente la utilidad del análisis estático en este tipo de situaciones.

Recientes trabajos han propuesto el uso de técnicas de análisis estático en el contexto de la detección de vulnerabilidades. Se proponen algoritmos muy eficientes pero potencialmente incoherentes [24, 96, 74] que requieren del uso de código fuente o del programa compilado, pero se enfocan en detectar una clase particular de vulnerabilidades.

A pesar de que las técnicas basadas en el análisis estático no resultan lo suficientemente precisas para ser usadas como un procedimiento efectivo de detección de vulnerabilidad, sí pueden brindar información útil para combinarse con otros enfoques [1, 79].

4.1.2. Análisis dinámico

El análisis dinámico se centra en ejecutar un programa sistemáticamente con distintas entradas para verificar si falla o no. Actualmente, uno de los enfoques más efectivos para la detección de vulnerabilidades en software complejo utilizando esta técnica es el *fuzzing*. El mismo se define como la generación de datos de entrada de manera de lograr un comportamiento inesperado en un programa. Por ejemplo, si un programa esperase una entrada de la siguiente forma:

```
var=1
```

la tarea principal de un *fuzzer* sería inducir al programa a fallar en distintas partes su código. En particular, podría utilizar alguna de las siguientes expresiones potencialmente inválidas:

```
=1
v ar=1
var=-2817324908701279755
var=~1
```

Este tipo de técnicas se utilizan desde hace varias décadas [59]. Existen dos enfoques distintivos para realizar *fuzzing* [60]: el enfoque **mutacional** y el enfoque **generacional**.

4.1.2.1. *Fuzzers* tradicionales

La primera generación de *fuzzers* utilizaron un enfoque de caja negra, es decir, no utilizaban la información proveniente del programa para producir entradas inválidas.

Fuzzers Mutacionales. Los *fuzzers* mutacionales producen entradas corruptas o inválidas para probar la robustez y seguridad de los programas mediante la alteración aleatoria de entradas válidas. Los mismos funcionan típicamente produciendo pequeñas mutaciones a nivel de bits o bytes completos. Actualmente existe una gran variedad de *fuzzers* mutacionales. Por ejemplo *zzuf* [9] es una herramienta creada por *Caca Labs* que produce mutaciones en las entradas de un programa de manera automática. Este *fuzzer* altera bits aleatorios de los datos que el mismo lee desde archivos o interfaces de red. *zzuf* puede configurarse de manera de corromper un pequeño porcentaje de los bits que un programa está leyendo. Otro *fuzzer* emblemático es *radamsa* [68]. Esta herramienta fue desarrollada por el grupo de programación segura de la universidad de Oulu (Filandia). A diferencia de los *fuzzers* a nivel de bits, que simplemente cambian el estado de los bits de manera aleatoria, *radamsa* agrega, elimina o modifica secuencias de bytes completos de la entrada. Esta herramienta recopila un gran variedad de mutaciones que resultaron útiles para descubrir errores y vulnerabilidades en el pasado.

A pesar de que los *fuzzers* mutacionales son herramientas simples y rápidas para encontrar errores y vulnerabilidades en programas, podemos destacar dos serias limitaciones: (1) requieren un buen conjunto inicial de entradas válidas para ser realmente efectivos y (2) gran parte de las entradas corruptas pueden ser descartadas por programas robustos en las primeras etapas de análisis (por ejemplo, porque se ha alterado una suma de verificación).

***Fuzzers* Generacionales.** Por otro lado, tenemos los *fuzzers* generacionales que producen entradas para probar programas mediante una especificación o modelo de las mismas. Estas herramientas buscan superar la limitaciones de los *fuzzers* mutacionales incorporando conocimiento específico, por ejemplo generando entradas utilizando una gramática [**?**]. Los *fuzzers* generacionales no son nuevos. Una de las herramientas más maduras y soportadas comercialmente es *Peach* [20]. Fue originalmente escrita en *Python* y luego reescrita en *C#* en sus más recientes versiones. Provee un amplio conjunto de funciones para la generación, mutación y monitoreo de fallos de programas. Sin embargo, para poder empezar a generar entradas de prueba, requiere la especificación de dos componentes principales:

- Modelos de Datos: una descripción formal de cómo los datos se componen de manera de poder generar entradas para un programa.
- Objetivo: un descripción formal de cómo las entradas para un programa pueden ser mutadas y cómo monitorear posibles fallos.

El inconveniente principal de *Peach* es el necesidad de especificar estos componentes en forma de archivos de configuración, debido al conocimiento de dominio específico requerido. Otra herramienta es *Sulley* [70], un formalismo para escribir *fuzzers* en *Python*. *Sulley* se presenta como una alternativa simplificada de *Peach* ya que los modelos requeridos para su funcionamiento se pueden especificar en código *Python*. Un *fuzzer* más reciente, de código libre creado por *Mozilla* es *Dharma* [64]. Esta herramienta fue diseñada por el equipo de seguridad de *Mozilla* para verificar la robustez del código de *Firefox*. Internamente, utiliza gramáticas libres de contexto y también se encuentra programada en *Python*. Tal como las demás, requiere al usuario la especificación de un modelo en forma de gramática, pero utiliza un formato sencillo de entender basado en texto plano. Además provee varios modelos para generar formatos de archivos específicos tales como *Canvas2D* y código de *Node.js*.

Las herramientas que funcionan por medio de técnicas de caja negra resultan sencillas de implementar y pueden utilizarse aun en proyectos de software grandes y complejos. Sin embargo, poseen evidentes limitaciones al no utilizar la información del programa para mejorar la mutación o generación de entradas inválidas. Además, no permiten distinguir entre distintos casos de prueba fallidos, por lo que estos deben ser analizados posteriormente por expertos de manera manual.

4.1.2.2. *Fuzzers* inteligentes

Para superar las limitaciones de los *fuzzers* tradicionales se plantearon distintas técnicas para darle inteligencia a estas herramientas.

Ejecución Simbólica. Las técnicas de ejecución simbólica utilizan el código de un programa para monitorear y analizar su ejecución de manera de permitir explorar todas las *trazas* posibles de su ejecución [46]. En general, la exploración se realiza mediante la computación y resolución de una fórmula lógica donde se almacenan todas las condiciones necesarias para una determinada ejecución, aquí denominada Π_{ej}. Una vez que se resuelve la formula, la solución determina las variables de entrada de manera de poder reconstruir la entrada. En otras palabras, estas técnicas permiten descubrir cada una de las posibles entradas que los programas aceptan.

Es importante destacar que las herramientas actuales realizan una variante de la ejecución simbólica que mantiene una parte de la ejecución con valores concretos en vez de simbólicos. El objetivo es mantener controlados los recursos utilizados en la exploración de las distintas entradas posibles de un programa. Sin este tipo de restricciones, el tiempo de cómputo y memoria requerido para la ejecución simbólica resultaría impráctico.

Caja Blanca. En los últimos años se desarrollaron técnicas para realizar *fuzzing* de *caja blanca* [31, 27] basadas en algunas variantes de la ejecución simbólica. El objetivo que estas técnicas persiguen es el de enumerar todas las entradas posibles de un programa y luego verificar si estas producen un fallo o no. Adicionalmente, algunas herramientas van más allá de la detección de fallas de manera de poder descubrir cierto tipo de vulnerabilidades [13, 39, 3]. La idea general es imponer condiciones en una ejecución concreta que garanticen que la vulnerabilidad se pueda aprovechar de alguna manera. Las condiciones particulares de explotabilidad de cierta vulnerabilidad se modelan con una formula, aquí denominada Π_{exp}. Finalmente, la técnica se encarga de verificar si es posible cumplir la conjunción de ambas fórmulas: $\Pi_{ej} \wedge \Pi_{exp}$. Si resulta posible, la herramienta obtiene una prueba de concepto de la vulnerabilidad que se busca. La herramienta más conocida que utiliza esta técnica es *Mayhem* [13] pero recientemente se incorporaron herramientas de código libre como *Angr* [83] y *Manticore* [90].

Caja Gris. Debido a que el *fuzzing* de caja blanca aún no resulta práctico de utilizar en aplicaciones complejas, surgieron distintas técnicas de *fuzzing* denominadas de *caja gris*. Estas técnicas se basan en el uso de cierta información extraída directamente de la ejecución de los programas para realizar *fuzzing* guiado. No resultan enteramente de caja blanca ni de caja negra, por eso se consideran de caja gris.

Entre las herramientas que utilizan estas técnicas, se destaca *American Fuzzy Lop* (*AFL*) [53], un *fuzzer* guiado por la respuesta de los programas. Dado un programa y un pequeño conjunto de entradas, *AFL* funciona produciendo un gran número de mutaciones sobre las entradas y ejecutando el programa que se quiere probar. Durante cada ejecución, la herramienta recolecta eficientemente información resumida sobre la ejecución y selecciona nuevas entradas básandose en algoritmos genéticos. Las trazas se usan para retroalimentar y guiar el *fuzzing*, de manera de seleccionar las entradas más variadas y así extender el proceso de mutación iteración tras iteración. Esta herramienta posee un impresionante número de fallos y vulnerabilidades descubiertas durante los últimos años [53]. Otro *fuzzer* que funciona con una técnica similar es *honggfuzz* [33]. Esta herramienta fue desarrollada por *Google* y se destaca por utilizar las nuevas características de hardware para extraer información de trazas de los programas eficientemente (e.g, utilizando la tecnología de *Intel BTS* [40]).

4.2. *QuickFuzz*: un *fuzzer* generacional en *Haskell*

Tal como se detalló en las secciones anteriores, los *fuzzers* sufren de varias limitaciones. En particular, estos resultan poco útiles para encontrar errores en programas que manipulen formatos de archivos complejos tales como documentos o imágenes vectoriales. Incluso los *fuzzers* inteligentes tienen grandes dificultades para aprender correctamente este tipo de formatos porque utilizan la información de las ejecuciones para ir adaptando sus mutaciones. En general, dicha información resulta de muy bajo nivel para poder recrear tipos de archivos complejos. Para intentar resolver ese problema se idearon los *fuzzers* generacionales. Éstos requieren que el usuario especifique un modelo o generador del formato de archivo que se quiere utilizar. No obstante, este requerimiento es demasiado costoso para la mayoría de los casos.

Una opción poco explorada en el estado del arte es la de añadir la información del formato de las entradas del programa mediante el análisis detallado del código de

terceros. Por ejemplo, si se quisieran generar documentos de tipo PDF, sería necesario analizar el código de un programa que pueda leer o escribir este tipo de archivos de manera de extraer automáticamente una especificación y un procedimiento de generación.

En esta sección, introducimos *QuickFuzz*: una herramienta que utiliza *QuickCheck* [14], el formalismo para realizar pruebas de programas de manera aleatoria basadas en propiedades, y *Hackage* [34], el repositorio de software de la comunidad de Haskell, para realizar *fuzzing* generacional de varios formatos de archivos complejos.

Adicionalmente, *QuickFuzz* integra el uso de *fuzzers* mutacionales y herramientas de detección de errores externas para descubrir errores en programas y aislar los casos de prueba resultantes. A diferencia de otras herramientas, *QuickFuzz* no requiere el desarrollo de modelos o generadores para producir los datos de entrada por parte de los usuarios, sino que utiliza las librerías de *Hackage* para generar un gran número de formatos de archivos de imágenes, documentos, archivadores y multimedia.

A grandes rasgos, para lograr generar los casos de prueba, se requieren librerías que provean: (a) un tipo de datos de *Haskell* al que llamaremos `T`, que sirve como una especificación del tipo de archivo que se quiere generar y (b), una función `encode` para almacenar los elementos del tipo `T` en archivos según el formato que corresponda. La utilización de *Hackage* permite ahorrar el enorme costo de crear el tipo de dato y la función `encode` correspondiente.

El punto clave detrás del éxito de *QuickFuzz* es la generación aleatoria de valores del tipo `T` utilizando los generadores de *QuickCheck*, la maquinaria especializada para producir valores a partir de un tipo de *Haskell*. Luego, los valores generados se almacenan en un archivo, que será mutado utilizando *fuzzers* tradicionales. Este procedimiento produce archivos inválidos pero con cierta estructura interna correcta. En el paso siguiente, el programa a probar se ejecuta utilizando estos archivos corruptos.

Finalmente, si se detecta una terminación anormal en la ejecución del programa (por ejemplo, un fallo de segmentación), *QuickFuzz* reporta dicho error junto con el archivo que permite replicar el fallo.

Utilizando el código de las librerías para manejar distintos formatos de archivos disponibles en *Hackage*, *QuickFuzz* permite generar y mutar un gran conjunto de tipos distintos de archivos. Sin embargo, es también posible para un usuario experto

agregar un nuevo tipo de dato así como las funciones para escribir archivos en el nuevo formato. De esta manera, *QuickFuzz* puede extenderse para descubrir errores en todo tipo de aplicaciones.

A pesar de que la herramienta está escrita íntegramente en *Haskell*, se destaca que puede utilizarse para encontrar errores en todo tipo de software ejecutable, sin importar el lenguaje en el que esté programado.

4.2.1. Conceptos Preliminares

Antes de explicar el proceso por el cual *QuickFuzz* funciona, es importante introducir algunos conceptos provenientes de la programación funcional.

Haskell. *Haskell* es un lenguaje de programación funcional de propósito general [50]. Este lenguaje provee un poderoso sistema de tipado y estructuras de datos algebraicas definidas por el usuario. Los mismos pueden utilizar parametrización, definiciones recursivas y otras características importantes para la creación de especificaciones de valores.

Tipos de Datos. *Haskell* permite definiciones de tipos polimórficos, por ejemplo para listas de valores de tipos de datos genéricos. Dichas listas pueden representarse usando dos constructores, `Nil` para representar las listas vacías y `Cons` para representar la listas con al menos un elemento:

```
data List a = Nil | Cons a (List a)
```

Nótese que la definición del tipo `List` es recursiva. A modo de ejemplo, definimos también algunas funciones que se utilizarán en esta sección:

```
length :: List a -> Int
length Nil = 0
length (Cons x xs) = 1 + length xs

snoc :: a -> List a -> List a
snoc x Nil = Cons x Nil
snoc x (Cons y ys) = Cons y (snoc x ys)

reverse :: List a -> List a
reverse Nil = Nil
reverse (Cons x xs) = snoc x (reverse xs)
```

donde `length` computa el tamaño de una lista, `snoc` agrega un elemento en el final de la lista y finalmente `reverse`, revierte el orden de los valores de la lista completa.

Sistema de Clases. *Haskell* provee también un poderoso sistema de clases. Informalmente, un sistema de clases provee mecanismos para garantizar que ciertas operaciones se podrán realizar para determinados tipos de valores. En la práctica, un sistema de tipos define una interfaz abstracta que deberá ser inicializada con funciones o valores concretos para poder utilizarse. Por ejemplo, en *Haskell* se define una clase muy utilizada llamada `Eq`. Esta clase define la relación de igualdad de los valores de un tipo dado.

```
instance Eq a => Eq (List a) where
Nil == Nil = True
(Cons x xs) == Nil = False
Nil == (Cons x xs) = False
(Cons x xs) == (Cons y ys) = (x == y) && (xs == ys)
```

Es interesante notar que la instancia de igualdad de `List a` se define suponiendo que el tipo paramétrico `a` ya posee una instancia de igualdad.

Hackage. Otra parte importante de esta herramienta proviene de *Hackage*, un repositorio centralizado de paquetes de software de la comunidad de *Haskell*. *QuickFuzz* utiliza una gran variedad de librerías de código de terceros para generar y mutar distintos formatos de archivos. Por ejemplo, la librería *JuicyPixels* está disponible en *Hackage* [92] y provee soporte para la lectura y escritura de distintos formatos de imágenes.

4.2.2. *QuickCheck*

QuickCheck es una herramienta que asiste al programador en formular y probar propiedades de los programas creada por Koen Claessen y John Hughes [14]. *QuickCheck* ofrece mecanismos para la generación aleatoria de valores de un tipo dado por medio de una serie de simples y modulares primitivas de *Haskell*. Una vez que los programadores definieron generadores adecuados, las propiedades se verifican mediante la generación aleatoria de un gran número de valores.

Propiedades. Para poder usar *QuickCheck*, un programador debe definir propiedades que su código debe satisfacer. Esta librería define las propiedades como predicados: funciones que retornan valores boleanos. Por ejemplo, siguiendo las definiciones básicas de listas dadas previamente, podemos probar que el tamaño de las listas se preserva cuando la lista se invierte:

```
propreverseSize :: List a -> Bool
propreverseSize xs = length xs == length (reverse xs)
```

La tarea de *QuickCheck* consiste en tratar de encontrar un contraejemplo para esta propiedad mediante la generación de un gran número de valores de `List a`. Si *QuickCheck* no puede detectar un contraejemplo luego de generar un gran número de valores, la propiedad se considera *probada*.

Generadores. *QuickCheck* requiere que el programador implemente un generador de valores del tipo `List a` para poder probar propiedades que involucren a dicho tipo, tales como `propreverseSize`. Para poder orquestar la generación de cada valores, *QuickCheck* utiliza instancias de clase, en particular de una clase especial llamada `Arbitrary`:

```
class Arbitrary a where
  arbitrary :: Gen a
```

QuickCheck provee instancias para tipos básicos de *Haskell* tales como `Integer` o `String`. Por el contrario, es responsabilidad del programador definir una instancia adecuada de `Arbitrary` para `List a` usando las funciones provistas por *QuickCheck*. Por ejemplo:

```
instance Arbitrary a => Arbitrary (List a) where
  arbitrary = genList
    where genList = oneof [ genNil, genCons ]
          genNil = return Nil
          genCons = do
                      x <- arbitrary :: Gen a
                      xs <- arbitrary :: Gen (List a)
                      return (Cons x xs)
```

La función `oneof` combina generadores de valores de manera de elegir uno de ellos con la misma probabilidad. En este código, uno de los generadores, `genNil`, siempre produce la lista vacía. El otro, `genCons`, genera una lista con al menos un elemento aleatorio.

Es importante destacar que esta implementación de `arbitrary` para `List a` no realiza ningún control sobre el tamaño de la lista que genera. Si bien en este ejemplo puntual la probabilidad es muy baja, es posible que la función generadora se ejecute sin terminación. Es por eso que *QuickCheck* provee funciones específicas para limitar el tamaño de los elementos generados aleatoriamente. Dado un valor entero n, las funciones `sized` y `resize` se encargan de contabilizar y controlar el tamaño de los elementos generados.

Utilizando una implementación alternativa de `arbitrary` para `List a` podemos garantizar su terminación debido a que cuando el tamaño llega a cero, la generación siempre debe resulta en una lista vacía, el valor menos complejo:

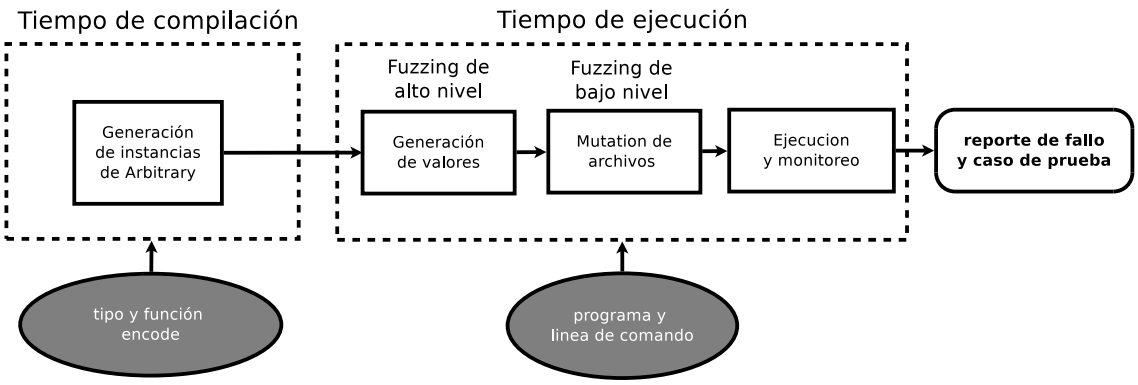


Figura 4.1: Arquitectura de *QuickFuzz*.

```
instance Arbitrary a => Arbitrary (List a) where
  arbitrary = sized genList
    where genList n = oneof [ genNil, genCons n]
          genNil = return Nil
          genCons 0 = genNil
          genCons n = do
                      x <- resize (n-1) (arbitrary :: Gen a)
                      xs <- resize (n-1) (arbitrary :: Gen (List a))
                      return (Cons x xs)
```

Finalmente, usando esta instancia, *QuickCheck* puede generar valores para *probar* esta propiedad de la siguiente forma:

```
quickCheck propreverseSize
```

Si la prueba resulta exitosa, *QuickCheck* reporta la cantidad de valores generados:

```
++++ OK, passed 100 tests
```

4.2.3. Un recorrido por *QuickFuzz*

En esta sección, mostramos como *QuickFuzz* funciona con un sencillo ejemplo donde descubrimos errores en *giffix*, una aplicación de línea de comandos que perte-

nece a la librería *giflib* [30]. Este programa intenta reparar imágenes de tipo "*Graphics Interchange Format*" (comúnmente abreviado *gif*), un antiguo pero aún popular formato de codificación de imágenes y animaciones utilizado en páginas web y aplicaciones móviles. Nuestra herramienta tiene soporte para la generación de archivos gif utilizando la librería *JuicyPixels* [92].

Para poder producir casos de prueba que revelan errores en el programa objetivo, nuestra herramienta sólo requiere del usuario:

- Un nombre de formato de archivo a generar.
- Una línea de comando para ejecutar el programa objetivo.

El programa objetivo se ejecutará sin modificaciones, por lo cual esta herramienta no está restringida en ese sentido. Por ejemplo, para iniciar una serie de pruebas de programa aleatorias de *giffix* ejecutamos:

```
$ QuickFuzz Gif 'giffix @@' -a radamsa -s 10
```

En este caso, nuestra herramienta reemplaza `@@` por un nombre de archivo aleatorio que contiene el archivo gif generado para probar *giffix*. El siguiente parámetro es el nombre del *fuzzer* utilizado y el último es el tamaño máximo del valor que representa el archivo gif tal como se definió en 4.2.1. Esta parámetro limita efectivamente los recursos computacionales como el tiempo y la memoria utilizados en la generación de casos de pruebas.

Después de ejecutarse por unos segundos, *QuickFuzz* encuentra un caso de prueba que induce un fallo de segmentación en *giffix*. En este punto, podemos examinar el directorio de salida para ver el archivo gif que produce el fallo.

La Figura 4.1 muestra la arquitectura de *QuickFuzz*. Los nodos oscuros indican que información es proporcionada por el usuario y los nodos en negrita, la salida producida por la herramienta. Para empezar, en tiempo de compilación, se selecciona una lista de tipos de datos correspondientes a formatos de archivos para la generación según su disponibilidad en *Hackage*. Luego, en tiempo de ejecución, se distinguen tres etapas: *fuzzing* de alto nivel, *fuzzing* de bajo nivel y ejecución. Veamos en detalle qué sucede en cada etapa con el ejemplo de *giffix*.

Generación de casos de prueba de alto nivel. Durante esta fase, *QuickFuzz* genera valores del tipo de dato `T` que representa el tipo de archivo que se desea generar para detectar fallos en un programa. La generación utiliza las herramientas provistas por *QuickCheck*. Explicaremos como se producen los valores utilizando `GifFile` un tipo que representa a los archivos gif tomado de *Juicy.Pixels*:

```
data Looping = LoopingNever
             | LoopingForever
             | LoopingRepeat Int
data GifFile = GifFile Header Images Looping
```

Un `GifFile` se compone de una cabecera con información básica de la imagen de tipo `Header`, una lista de imágenes de mapa de bits de tipo `Images` e información adicional sobre el número de repeticiones que se especifica en tres constructores del tipo `Looping`. Sin pérdida de generalidad, en este ejemplo omitiremos las definiciones de `Header` y `Images`.

Para obtener generadores de valores de los tipos que componen `GifFile`, simplemente seguimos las definiciones de los constructores de manera de requerir valores arbitrarios de sus parámetros cuando fuera necesario (por ejemplo el número entero de `LoopingRepeat`). Para la generación de código *Haskell*, utilizamos una librería auxiliar desarrollada específicamente para este propósito llamada **MeGaDeTH** [52] que se encarga de producir mecánicamente las instancias de `Arbitrary`. Esta librería utiliza las primitivas de *Template Haskell*, una librería para generar código *Haskell* fácilmente. Volviendo al ejemplo, las instancias necesarias para generar valores de tipo de `GifFile` incluyen:

```
instance Arbitrary Looping where
  arbitrary = sized gen
    where gen n = oneof [ return LoopingNever,
                          return LoopingForever,
                          return LoopingRepeat
                          <*> (resize (n-1) arbitrary :: Gen Int) ]
```

```
instance Arbitrary GifFile where
  arbitrary = sized gen
    where gen n = do
            header  <- resize (n-1) (arbitrary :: Gen Header)
            images  <- resize (n-1) (arbitrary :: Gen Images)
            looping <- resize (n-1) (arbitrary :: Gen Looping)
            return (GifFile header images looping)
```

Luego de generar un valor del tipo `GifFile`, utilizamos la función `encode` provista por *Juicy.Pixels* para transformar ese valor en una secuencia de bytes de acuerdo al estándar definido en el formato de archivos gif. El resultado de esta etapa es un archivo gif generado aleatoriamente, muy probablemente con valores internos inválidos tales como el tamaño o la longitud.

Fuzzing de bajo nivel. Usualmente, el uso de generación de valores para el *fuzzing* de alto nivel no es suficiente para detectar ciertos tipos de errores comunes. Es por eso que la siguiente fase se encarga de utilizar distintas herramientas que realizan *fuzzing* sobre archivos para introducir mutaciones a nivel de bits o bytes. En particular, la versión actual de *QuickFuzz* puede utilizar *zzuf*, *radamsa* y *honggfuzz*.

Al final de esta fase, el resultado será un archivo de tipo gif con un gran número de mutaciones y valores internos inconsistentes gracias a la combinación de *fuzzing* de alto y bajo nivel.

Ejecución de casos de prueba. La última fase del proceso de detección de vulnerabilidades y fallos consiste esencialmente en la ejecución del programa utilizando el archivo de entrada previamente generado. Si la ejecución termina anormalmente, el caso de prueba se almacena para análisis posterior en el directorio de salida. Para detectar ejecuciones terminadas anormalmente, necesitamos un procedimiento efectivo para determinar si se produjo un fallo o no. Para esto, utilizamos las señales de sistema enviadas a un programa, tales como `SIGSEGV`, `SIGABRT` y otras para identificar un fallo en una ejecución, por medio del código de retorno de programa. Tradicionalmente en los sistemas *GNU/Linux*, un proceso que termina normalmente

retorna un valor nulo, mientras que valores positivos refieren a distintos tipos de errores. En particular, cuando un proceso termina debido a una señal asociada con un número n, el sistema establece el código de retorno en $128 + n$. Capturamos esta condición utilizando el siguiente predicado de *Haskell*:

```
hasFailed :: ExitCode -> Bool
hasFailed (ExitFailure n) = (n < 0 || n > 128) && n /= 143
hasFailed ExitSuccess = False
```

Excluimos la señal `SIGTERM` (que utiliza el código de salida 143) ya que esta se utiliza usualmente para terminar un proceso que ha excedido su tiempo máximo de ejecución. De esta manera, cuando los programas quedan *colgados* no son considerados como fallos. Además, `hasFailed` nos servirá para definir la propiedad específica que *QuickCheck* intentará falsear para descubrir fallos y vulnerabilidades en programas. En particular *QuickFuzz* utiliza la siguiente propiedad:

```
propNoFail :: Cmd -> (a -> ByteString) -> a -> Property
propNoFail cmd encode x = do run (write filename (encode x))
                             ret <- run (execute cmd)
                             assert (not (hasFailed ret))
```

La propiedad `propNoFail` recibe una línea de comando a ejecutar (`cmd`), una función para preparar el archivo con una entrada inválida (`encode`) y el valor generado que se desea probar (`x`). La propiedad almacena el archivo con la entrada en algún archivo (`filename`), ejecuta el programa y finalmente verifica si el código de retorno indica un fallo con `hasFailed`. Cualquier valor que falsee esta propiedad se guardará para posterior análisis.

En nuestro ejemplo, luego de generar y probar algunos archivos gif, *QuickFuzz* encuentra un caso de prueba para reproducir una vulnerabilidad de desbordamiento de *buffer* conocida en *giffix* (CVE-2015-7555). Esta vulnerabilidad está causada por la falta de validación en el tamaño de las estructuras internas que definen la secuencias de imágenes. La identificación e individualización de fallos no forma parte de la herramienta, aunque puede ser implementada en el futuro. Adicionalmente, nuestra herramienta puede utilizar programas auxiliares para mejorar la detección de fallos tales como *Valgrind* [66] y *Address Sanitizer* [80] para detectar errores más sutiles que no causan terminación anormal de los programas.

Código
Javascript
Python
Html
Lua
Xml
Css
Sh
GLSL
Dot
Regex

Imágenes
Bmp
Gif
Png
Jpeg
Svg
Eps
Ico
Tga
Tiff
Pnm

Multimedia
Ogg
ID3
Midi
TTF
Wav

Documentos
PDF
PS
Docx
Odt
Rtf
Ical

PKI
ASN.1
X509
CRL

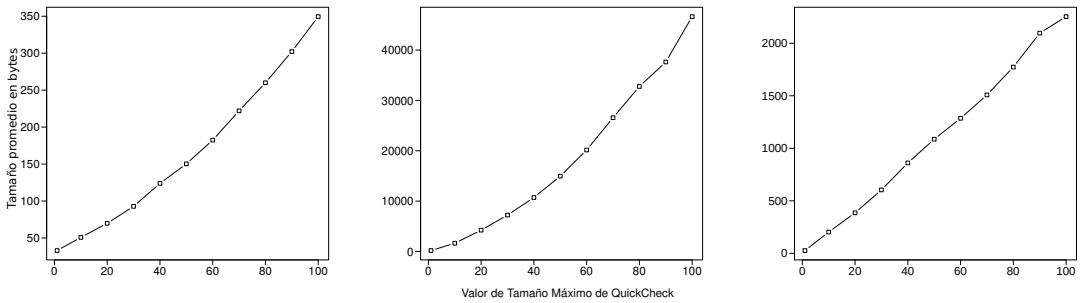
Archivadores
Zip
Gzip
Tar
CPIO

Figura 4.2: Lista de formatos disponibles en *QuickFuzz*.

4.2.4. Evaluación

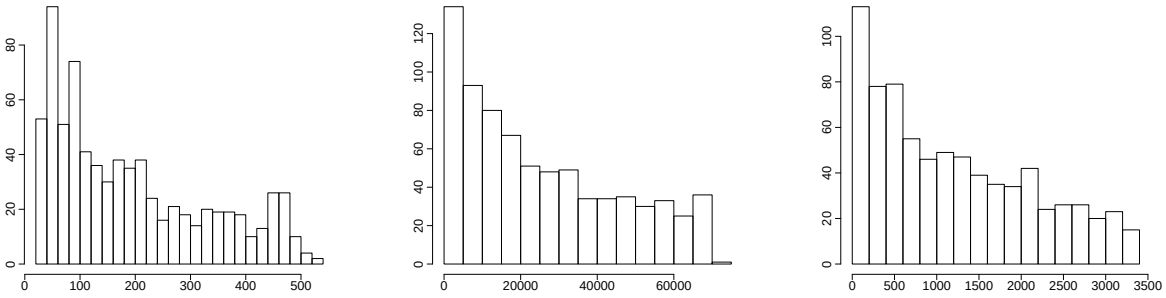
En esta sección describimos diferentes experimentos realizados y sus resultados para entender cómo *QuickFuzz* genera y altera entradas de programas reales. De la extensa lista de formatos soportados por *QuickFuzz* expuesta en la Figura 4.2 hemos seleccionado cinco formatos distintos para nuestros experimentos: zip, png, jpeg, xml y svg. Estos han sido seleccionados debido a que representan distintos tipos de especificaciones de archivos para diferentes aplicaciones. Por ejemplo, algunos de estos son formatos binarios y otros lenguajes de marcado que pueden ser entendidos por personas.

En estos experimentos se apunta a dilucidar como *QuickFuzz* se comporta durante el proceso de *fuzzing*. Debido a que este proceso es inherentemente aleatorio, cada medida experimental que se detalla en esta sección fue promediada sobre 10 repeticiones en un procesador dedicado de un Intel i7 (3,40GHz).



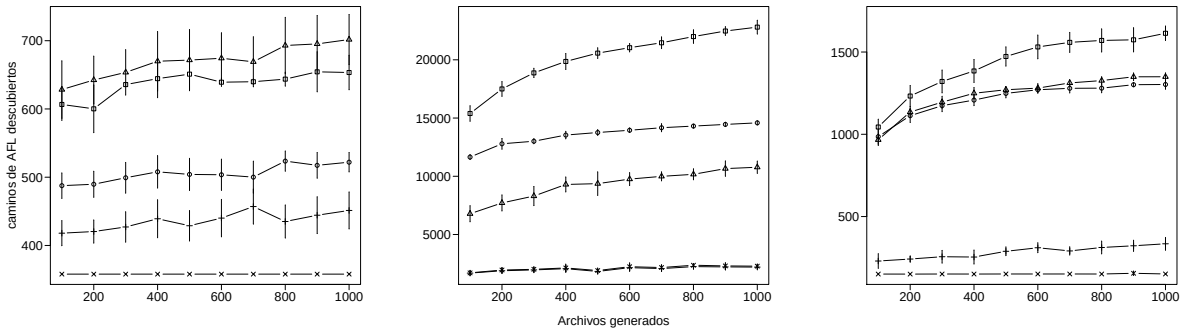
(a) png (b) svg (c) zip

Figura 4.3: Tamaño promedio en bytes de los archivos generados.



(a) png (b) svg (c) zip

Figura 4.4: Frecuencia de los tamaños en bytes de los archivos generados.



(a) libpng 1.2.50 (b) libxml 2.9.1 (c) libjpeg-turbo 1.3.0

Figura 4.5: Promedio y desviación estándar de *caminos* descubiertos por cada formato de archivo dado un número de entradas generadas utilizando distintas variantes de la configuración de *QuickFuzz*: los círculos (◯) representan los caminos generados por la ejecución directa de los archivos generados, los triángulos (△) son ejecuciones que incluyen mutación por parte de *zzuf* y los cuadrados (□) son ejecuciones que incluye mutaciones por parte de radamsa. Mases (+) y cruces (×) representan los archivos generados con bytes aleatorios con el uso de los números mágicos correspondientes a cada formato y sin utilizarlos, respectivamente.

Tamaño de la generación. Un importante parámetro en la generación de entradas es el tamaño máximo de los archivos resultantes. Este valor debe ser cuidadosamente controlado permitiendo al usuario adaptarlo en función de los recursos disponibles para las pruebas de programa. Si no se ajusta debidamente este parámetro se podría generar un gran número de archivos muy pequeños o muy grandes que no contribuya a la detección de fallos.

Para evitar este problema, *QuickFuzz* utiliza código para generar valores que mantiene acotado el tamaño de los mismos usando la función `resize` provista por *QuickCheck*. Las Figuras 4.3a, 4.3b y 4.3c muestran como los tamaños promedios de los archivos varían cuando el tamaño de *QuickCheck* se incrementa. Los resultados muestran que el promedio de los tamaño de los archivos resultantes crece linealmente.

También es importante indagar en la distribución de tamaños de los archivos generados, considerando que un sesgo hacia los tamaños pequeños es útil para detectar fallos más eficientemente. De hecho, existe un doble beneficio de este sesgo: (1) contribuye a la rápida ejecución de los programas y (2) tiende a generar casos de prueba con un bajo nivel de redundancia. Los casos de pruebas reducidos en tamaño son útiles para los desarrolladores del programa que se prueba ya que suelen ser más fáciles de entender. Esto permite reparar el código que produce el fallo más rápidamente.

En nuestros experimentos, analizamos las distribuciones de los archivos generados utilizando *QuickFuzz* tal como se muestra en las Figuras 4.4a, 4.4b y 4.4c. Las figuras muestran un sesgo hacia los archivos más pequeños.

Efectividad de la Generación. Idealmente un *fuzzer* debería generar entradas para producir un gran número de ejecuciones distintas. Cuando se quiere probar efectivamente un programa, se busca alcanzar el mayor número de líneas de código posible con el objetivo de descubrir fallos en el mismo.

Para poder explorar la efectividad en la generación de entradas de *QuickFuzz*, evaluamos el número de ejecuciones distintas que se producen en el funcionamiento de los programas de prueba. Para nuestros experimentos utilizamos una medida de cobertura de código de programa conocida como *camino.* Dicha medida es empleada por *AFL* [54]. A grandes rasgos, esta métrica representa al conteo de cuantas veces se visita cada línea de código. Por esta razón, esta medida de cobertura de código

utilizada por *AFL* podría asociar diferentes ejecuciones a un mismo *camino*. Sin embargo, utilizamos esta métrica debido a que:

- Fue diseñada para resultar útil en el proceso de detección de fallos y vulnerabilidades: la aparición de más caminos está altamente correlacionada con el descubrimiento de más fallos [5].
- Fue desarrollada utilizando un enfoque modular: podemos reutilizar fácilmente los componentes *AFL* que sólo extraen y cuentan la cantidad de *caminos* distintos.
- Fue desarrollada para resultar muy eficiente: la implementación actual extrae la información de los caminos prácticamente sin penalización en la ejecución.

En nuestros experimentos utilizamos *QuickFuzz* para generar y mutar archivos de tipo png, jpeg y xml. Luego, ejecutamos cada uno utilizando librerías de código abierto para poder procesarlos. Específicamente, utilizamos *libjpeg-turbo 1.3.0* para leer archivos jpeg, *libpng 1.2.50* para leer archivos png y *libxml 2.9.1* para leer archivos *xml*. Las Figuras 4.5a, 4.5c y 4.5b muestran la cantidad de caminos que se descubren mediante la ejecución de los casos de prueba generados por la herramienta utilizando *fuzzers* de bajo nivel (*zzuf* o *radamsa*) o sin ellos.

Adicionalmente incluimos dos medidas de base para comparar como las técnicas utilizadas por *QuickFuzz* mejoran el número de *caminos* descubiertos. La primera medida muestra el uso de archivos puramente aleatorios y la segunda medida utiliza también archivos aleatorios, pero con los correspondientes números mágicos para cada formato. *QuickFuzz* supera ampliamente las medidas de base gracias a la información que posee sobre los distintos formatos de archivos. En las Figuras 4.5a, 4.5c y 4.5b, la simbología refiere a distintas variantes de la configuración de *QuickFuzz*. Los círculos ($\bigcirc$) representan los caminos generados por la ejecución directa de los archivos generados. Los triángulos ($\triangle$) son ejecuciones que incluyen mutación por parte de *zzuf* y los cuadrados ($\square$) son ejecuciones que incluye mutaciones por parte de *radamsa*. Finalmente, mases ($+$) y cruces ($\times$) representan los archivos generados con bytes aleatorios con el uso de los números mágicos correspondientes a cada formato y sin utilizarlos, respectivamente.

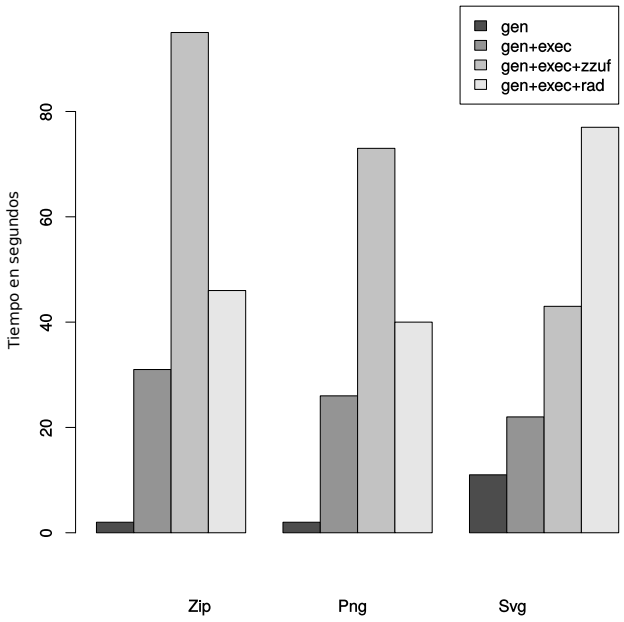


Figura 4.6: Tiempo empleado por *QuickFuzz* para realizar *fuzzing* de distintos tipos de archivos.

Respecto a las figuras en sí, es apreciable el incremento de los *caminos* descubiertos a medida que se generan más archivos. No obstante, el número de caminos distintos crece muy lentamente al acercarse a los mil casos de prueba. Este fenómeno es esperable debido a que *QuickFuzz* funciona como un *fuzzer ciego* que simplemente genera y ejecuta sin utilizar ninguna información del comportamiento del programa para guiar el *fuzzing*.

Adicionalmente, el efecto del *fuzzing* de bajo nivel depende del tipo de formato y el programa que lo procesa. Por ejemplo, el caso de los archivos xml usando *libxml2* es emblemático. Por un lado, el uso de *radamsa* para mutar los archivos mejora notablemente los caminos descubiertos, en comparación con la ejecución sin *fuzzing* externo. Por otro lado, la utilización de *zzuf* produce el efecto opuesto: el número de caminos descubiertos decrece significativamente cuando se lo utiliza. Este comportamiento puede deberse a que la mutación aleatoria de bits causa que el archivo se corrompa demasiado, forzando al programa a descartarlo de manera temprana en la ejecución.

Velocidad de Generación, Mutación y Ejecución. La velocidad es un atributo crítico de cualquier *fuzzer*: este tipo de herramientas deben utilizar el menor tiempo posible en la generación y la mutación de las entradas. Para una evaluación de la velocidad de generación de *QuickFuzz*, medimos los tiempos empleados por la herramienta al realizar:

- Generación de archivos utilizando *fuzzing* de alto nivel referido como `gen`.
- Generación de archivos utilizando *fuzzing* de alto nivel y luego ejecución de un programa referido como `gen+exec`.
- Generación de archivos utilizando *fuzzing* de alto nivel, mutación con *fuzzing* de bajo nivel utilizando *zzuf* y luego ejecución de un programa referido como `gen+exec+zzuf`.
- Generación de archivos utilizando *fuzzing* de alto nivel, mutación con *fuzzing* de bajo nivel utilizando *radamsa* y luego ejecución de un programa referido como `gen+exec+radamsa`.

Para poder realizar un medición objetiva del tiempo empleado en la ejecución de un programa a probar, pero sin depender del programa que se utilice, seleccionamos la utilidad `echo` (usualmente localizada en `/bin/echo` en GNU/Linux). Este sencillo programa no lee archivos y por lo tanto debería ejecutarse en tiempo constante, sin importar lo que se genere.

La Figura 4.6 muestra una comparativa del tiempo que *QuickFuzz* necesita para realizar cada paso del proceso de *fuzzing* para tres tipos de distintos de archivos. Nuestros experimentos parecen indicar que el tiempo consumido por el código generado por *MegaDeTH* (`gen`) es bajo en comparación al requerido por las otras tareas de nuestra herramienta tales como la ejecución y la mutación. Por ejemplo, se puede observar una pérdida de velocidad notable en la ejecución del programa mismo. A pesar de que `echo` ejecuta unas pocas instrucciones, es probable que la mayor parte del tiempo de vida del proceso se utilice en las operaciones de `fork` y `exec`: esta es una de las razones por las cuales algunos *fuzzers* implementan una alternativa eficiente para la ejecución de los programas sin utilizar estas primitivas.

Otro efecto interesante que se puede observar está relacionado con el uso de los *fuzzers* de bajo nivel. Es esperable que estos programas realicen mutaciones aleatorias

Programa	Formato	Referencia	Programa	Formato	Referencia
Firefox	Gif	CVE-2016-1933	Cairo	Svg	CVE-2016-9082
Firefox	Zip	CVE-2015-7194	libgd	Tga	CVE-2016-6132
Firefox	Svg	1297206	libgd	Tga	CVE-2016-6214
Firefox	Gif	1210745	GraphicsMagick	Svg	CVE-2016-2317
mujs	Js	CVE-2016-9109	GraphicsMagick	Svg	CVE-2016-2318
Webkit	Js	CVE-2016-9642	Mini-XML	Xml	CVE-2016-4570
Webkit	Regex	CVE-2016-9643	libical	Ical	CVE-2016-9584
gif2webp	Gif	CVE-2016-9085	Mini-Xml	Xml	CVE-2016-4571
VLC	Wav	CVE-2016-3941	GDK-pixbuf	Bmp	CVE-2015-7552
Jasper	Jpeg	CVE-2015-5203	GDK-pixbuf	Gif	CVE-2015-7674
libXML	Xml	CVE-2016-4483	GDK-pixbuf	Tga	CVE-2015-7673
libXML	Xml	CVE-2016-3627	GDK-pixbuf	Ico	CVE-2016-6352
Jq	Json	CVE-2016-4074	mplayer	Wav	CVE-2016-5115
Jasson	Json	CVE-2016-4425	mplayer	Gif	CVE-2016-4352
cpio	CPIO	CVE-2016-2037	libTIFF	Tiff	CVE-2015-7313

Tabla 4.1: Algunas de las vulnerabilidades encontrada por *QuickFuzz* al utilizarlo sobre programas complejos.

a los datos sin importar como estos sean. Por ejemplo, en el caso de *zzuf*, un *fuzzer* que sólo realiza mutaciones aleatorias a nivel de bits utilizando operaciones lógicas tales como XOR, el tiempo empleado en este proceso es constante. Sin embargo, radamsa funciona de manera distinta. Este *fuzzer* sí realiza mutaciones de acuerdo a los datos que recibe. De hecho, algunas mutaciones se aplican a casos muy especializados [69] tales como los lenguajes de marcado. Esto explica porque se observa una pérdida de velocidad significativa al utilizarse con archivos svg, un formato basado en xml, uno de los lenguajes de marcado más populares.

4.2.5. Resultados

Utilizamos *QuickFuzz* para poder generar y mutar una gran cantidad de tipos de archivos distintos, que se muestran en la Figura 4.2. Esto nos permitió detectar y reportar un gran número de vulnerabilidades y fallos en software complejo tales como navegadores web, librerías para el procesamiento de imágenes y compresores de archivos, entre otros. Todas las vulnerabilidades descubiertas por *QuickFuzz* eran previamente desconocidas y fueron reportadas a los correspondientes desarrolladores

del software afectado. Los resultados se muestran en la Tabla 4.1.

Una de las vulnerabilidades encontradas más notorias fue CVE-2016-1933 causada por un desbordamiento de enteros en el código que procesa los archivos gif de *Firefox*. Dado un caso de prueba identificado por *QuickFuzz* que producía un fallo completo en *Firefox*, el equipo de seguridad de *Mozilla* identificó la línea que producía el desbordamiento:

```
MakeUnique(mImageSize.width * mImageSize.height * sizeof(uint32_t)))
```

donde `MakeUnique` es una función que reserva memoria de acuerdo al tamaño que recibe.

El caso de prueba que causaba el fallo era un archivo gif con 65534 píxeles de ancho y 65531 píxeles de alto. El desbordamiento de enteros estaba causado por varios factores: por un lado, los tamaños de ancho y alto fueron declarados como enteros de 32-bits con signo, con lo cual su multiplicación directa causaría un desbordamiento. Por otro lado, también se realiza una multiplicación por 4 por un valor de tipo `size_t`, normalmente un entero largo sin signo en arquitecturas de 64-bits. Es esperable que la multiplicación de estos valores resulte en 17178034216 bytes reservados en memoria. Pero en este caso, el valor es mucho más grande.

Para entender este fallo es crítico analizar el orden de la operaciones: la operación de multiplicación es asociativa a derecha, con lo cual el código ejecuta de la siguiente manera:

1. Se multiplica `mImageSize.width * mImageSize.height` donde el resultado es entero con signo de 32-bits. Aquí se produce el desbordamiento y el resultado es negativo. El valor resultante es `0xfff9000a`.

2. Se multiplica el resultado anterior por 4 donde el resultado es un entero sin signo de 64-bits. Aquí el valor del entero con signo de 32-bits se convierte a entero sin signo por las reglas de promoción de enteros en C. Para esta conversión de 32-bits a 64-bits, se realiza la extensión de signo. Entonces el entero con signo `0xfff9000a` se convierte en `0xfffffffffff9000a` para luego multiplicarse `por 4.`

3. Finalmente, *Firefox* intenta reserva una exorbitante cantidad de memoria: 18446744073707716648 bytes, lo que equivale aproximadamente a 16777215 terabytes, causando que el programa aborte su ejecución por falta de memoria.

Además, se reportaron algunos fallos de software muy utilizado que si bien no representaban problemas de seguridad, podía afectar a un gran número de usuarios. Por ejemplo un fallo que detenía la compilación de un programa de Python y más de 20 errores de memoria en GNU Bash y Busybox.

4.2.6. Limitaciones

El uso de librerías de código de terceros para la generación de valores arbitrarios acarrea algunas limitaciones. Para empezar, algunos de los módulos que generan tipos de archivos complejos no implementan las especificaciones completas. Por ejemplo, el código para escribir archivos bmp en `Juicy.Pixels` no posee soporte para compresión tal como se define en las especificaciones de dicho formato. Por lo tanto, esa variante de archivos bmp nunca será probada efectivamente por *QuickFuzz*. En este sentido, los tipos de datos provistos por librerías de terceros actúan como especificaciones incompletas. De todas maneras, cuando los desarrolladores de `Juicy.Pixels` amplíen el soporte del tipo bmp, sólo se requiere recompilar *QuickFuzz* utilizando la nueva versión para aprovechar estas mejoras.

Otra limitación relacionada con el uso de código de terceros reside en las funciones `encode` provistas, debido a que estas pueden resultar parciales, o sea pueden no estar definidas para todas las entradas posibles. En el caso de que la función no esté definida para algún valor, se produce una excepción al intentar ejecutar dicho caso. Por ejemplo, si la función `encode` requiere alguna clase de restricción poco probable de satisfacer de manera aleatoria tal como adivinar un número mágico, puede suponer un problema para *QuickFuzz*. Veamos un ejemplo en el formato gif:

```
encodeHeader :: Int -> ByteString
encodeHeader version = | version == 87 = "GIF87"
                       | version == 89 = "GIF89"
                       | otherwise = error "invalid version"
```

En esta función, el proceso de construir archivos gif se define para dos números de versiones distintas: 87 y 89. No obstante, el enfoque de generación aleatoria detallado

en la Sección 4.2.3 no resulta efectivo dado que la probabilidad de seleccionar un número válido de versión es de 1 en $2,147,483,647$. Actualmente, las librerías que se utilizan de *Hackage*, se examinan manualmente antes de ser incorporadas en nuestra herramienta para evitar este tipo de problemas.

Finalmente, el código de librerías de terceros incluye sus propios fallos. No resulta sorprendente que algunos de estos fallos pueden ser detectados utilizando la generación aleatoria de valores usando *QuickCheck*. Reportamos los fallos encontrados [26] a los desarrolladores del código y estos fueron solucionados a la brevedad. En los casos de que no se haya podido incorporar una solución a los errores y limitaciones del código de `encode`, *QuickFuzz* simplemente captura e ignora las excepciones generadas y continúa el *fuzzing* con el próximo valor aleatorio.

4.2.7. Comparativa

Comparar distintas herramientas que realizan *fuzzing* es una tarea compleja. Para empezar, solo es posible comparar *fuzzers* que utilizan técnicas similares. Por ejemplo, en el caso de los *fuzzers* generativos, la especificación de los valores a generar debería ser similar o equivalente; de otra manera la generación de entradas completas puede requerir más tiempo en un determinado *fuzzer*. Si esto fuera así, los *fuzzers* que generen valores más complejos puede ser vistos como más lentos e ineficientes.

Además, tal como se mencionó en la Sección 4.1.2.1, algunos *fuzzers* como *Peach* necesitan ser configurados adecuadamente con los modelos de los datos a generar antes de iniciar el proceso de descubrimiento de fallos y vulnerabilidades. Existen librerías que recopilan especificaciones, formatos y protocolos, pero suelen ser de acceso pago [21].

Afortunadamente, *Dharma* [64], uno de los *fuzzers* ya presentados, resulta un buen candidato para una comparativa con *QuickFuzz* debido a estar disponible pública y gratuitamente e implementar la generación de archivos gráficos vectoriales [93] (o SVG por sus siglas en inglés). *QuickFuzz* puede generar este tipo de imágenes mediante los tipos de datos y las funciones del paquete `svg-tree` [91].

De todas maneras, una comparativa directa de los errores y vulnerabilidades descubiertos por ambos *fuzzers* no es posible: no pudimos encontrar información pública sobre la cantidad de vulnerabilidades descubiertas por *Dharma*, aunque creemos que

ya ha sido utilizado extensivamente por *Mozilla* para mejorar la estabilidad del código de *Firefox*.

Lo que si resulta posible es comparar el rendimiento de ambas herramientas generando archivos svg. Para esto se midieron los rendimientos en la generación de 10,000 archivos svg utilizando la siguiente fórmula:

$$\text{rendimiento} = \frac{\text{tamaño total de los archivos}}{\text{tiempo de generación}}$$

QuickFuzz produce archivos svg 1,9 veces más rápido que *Dharma*. A pesar de que esta sencilla comparativa no refleja el panorama completo debido a que no podemos comparar la velocidad de generación de otros tipos de archivos, sí indica que nuestra herramienta se encuentra optimizada para la generación de archivos aún en formatos tan complejos como svg.

4.2.8. Implementación

QuickFuzz se implementó íntegramente en *Haskell* utilizando *Template Haskell* [81] y librerías de terceros disponibles en *Hackage*. Su código es software libre (GPL3) y está disponible gratuitamente en su sitio web oficial. La librería utilizada para la generación de código de instancias de clases `Arbitrary` también fue implementada para esta herramienta y se denomina *MeGaDeTH*. La misma también cuenta con su repositorio propio. También es destacable que debido a que *QuickFuzz* utiliza gran cantidad de código de terceros, fue necesario elaborar un sistema de compilación personalizado, donde el usuario puede decir qué módulos desea utilizar antes de empezar. De esta manera el usuario puede compilar *QuickFuzz* para generar únicamente ciertos tipos de archivos y así ahorrar tiempo y memoria en su uso.

4.3. *XCraft*: un generador de *exploits* de caja negra

Descubrir fallos y vulnerabilidades no es suficiente para lograr software seguro. Igual o más importante es demostrar qué tan grave es cada fallo: si resulta crítico para la seguridad de un programa o no. Esta sección introduce ***XCraft***, un sistema de evaluación rápida de fallos de seguridad a gran escala. Esta herramienta no sólo busca encontrar fallos, sino que se encarga de identificar las vulnerabilidades críticas

de manera de poder priorizar este tipo de errores en el proceso de desarrollo de software. Por ejemplo, si un programa tiene una vulnerabilidad que potencialmente permite la ejecución arbitraria de código ya identificada, *XCraft* puede generar un reporte que incluya un caso de prueba de concepto que logre la ejecución de un comando de sistema, efectivamente mostrando su peligrosidad. Adicionalmente, las técnicas utilizadas en esta herramienta no requieren gran cantidad de recursos, por lo cuales pueden ser utilizadas para identificar vulnerabilidades en una gran cantidad de programas distintos.

Uno de los desafíos más importantes que intenta resolver *XCraft* es la prueba de explotabilidad de las vulnerabilidades de programas mediante la generación de *exploits*. Se han propuesto varios trabajos previos en generación automática de *exploits* (o AEG por sus siglas en inglés) basados en técnicas de caja blanca [39, 3] que han sido usados para identificar vulnerabilidades críticas en los programas. Sin embargo, el problema más común de los sistemas de AEG que utilizan técnicas de caja blanca tales como la ejecución simbólica, es el excesivo uso de recursos que requieren en tiempo de procesamiento y memoria para poder construir *exploits*. Por ejemplo, este tipo de técnicas deben tener en cuenta la complejidad de las llamadas a sistemas, las condiciones y los bucles en trazas con millones de instrucciones. En la práctica, estos problemas hacen que dichas técnicas tengan largos tiempos de ejecución, que reducen notablemente la utilidad de las mismas.

En contrapartida, *XCraft* utiliza un enfoque de caja negra para modelar lo que el programa parece hacer, intentar identificar una vulnerabilidad y reportar un caso de prueba que muestre su peligrosidad. Este enfoque no requiere razonar sobre comportamientos complejos, aunque podría no funcionar para todos los casos de pruebas vulnerables. En otras palabras, es rápido pero potencialmente incompleto.

4.3.1. Arquitectura

La Figura 4.7 muestra la arquitectura general de *XCraft*. La herramienta debe ser provista con al menos un caso de prueba y un programa ejecutable para analizar. A grandes rasgos, el enfoque utilizado consiste de dos etapas: (1) el descubrimiento de fallos mediante mediante el uso de *fuzzing* y (2) la generación automática de *exploits*. Ambas etapas pueden descubrir nuevos fallos que se almacenan en una base de datos

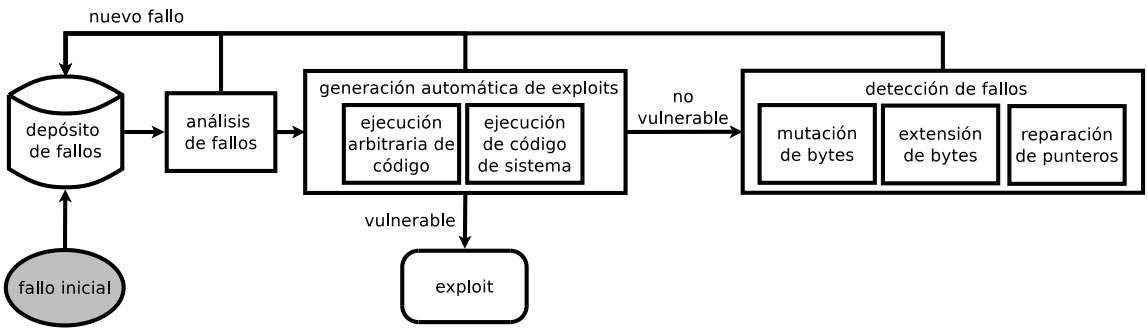


Figura 4.7: Arquitectura general de *XCraft*

para analizar eventualmente por la misma herramienta.

Es importante destacar que *XCraft* fue diseñado para realizar un análisis totalmente automático en ambas etapas, pudiendo funcionar en paralelo si se tienen diferentes casos de prueba para analizar. Adicionalmente, la herramienta fue pensada para ser modular y fácilmente extensible de manera de incorporar nuevas técnicas específicas para descubrir errores o generar *exploits* particulares.

4.3.2. Análisis de Fallos

Antes de realizar cualquier tipo de análisis o detección de nuevos fallos, *XCraft* necesita recopilar información útil sobre el fallo.

Análisis de Causa Directa. En este primer paso, *XCraft* analiza la instrucción donde se produce el fallo. Este tipo de análisis brinda información de rápida extracción sobre la causa directa del fallo. *XCraft* pone particular énfasis en identificar los registros o posiciones de memoria responsables directos del fallo en la última instrucción ejecutada. Específicamente, los fallos se dividen en dos categorías:

- **Puntero a instrucción inválido**: este fallo es causado por un valor inválido del registro de puntero a la instrucción a ejecutar. Es posible que el puntero apunte a una página de memoria inexistente, sin permisos de ejecución o simplemente que contenga una instrucción inválida.

  Por ejemplo, una instrucción de salto directo puede causar este tipo de fallos: `jmp *%eax` donde `eax` contiene un valor nulo.

- **Acceso a memoria inválido**: este fallo es causado por una instrucción que intenta leer o escribir una página de memoria inválida o sin los permisos adecuados.

  Por ejemplo, una instrucción de movimiento de memoria puede causar este tipo de fallos: `mov -0x8(%ebx),%eax` donde la dirección apuntada por `ebx-8` se encuentra en una página sin permisos de lectura.

Recordemos que esta herramienta está limitada a un análisis de caja negra y por eso no puede determinar causas elaboradas de fallos, por ejemplo "la memoria reservada por el procedimiento P no resulta suficiente y produce un desbordamiento".

Individualización de Fallos. Los fallos que se descubran se individualizan y clasifican para evitar la redundancia y la repetición en el análisis. Para obtener un identificador de cada fallo, se utiliza la pila de llamados a función, el tipo de fallo y la información referida a los atributos que causan el fallo. Esta información está disponible incluso en programas compilados sin símbolos de depuración y puede ser extraída rápidamente. El paso siguiente es almacenar todos los fallos descubiertos en una base de datos para su rápido acceso en las siguientes etapas.

Análisis de Entrada. Adicionalmente, resulta de suma importancia analizar cuál es la relación entre la entrada del programa y los atributos involucrados en el fallo que se produce: un caso de prueba generalmente resulta innecesariamente grande para su análisis. Por ejemplo, dada una entrada de 10000 bytes de un programa que falla, no todos los bytes van a estar necesariamente relacionados con la falla. De hecho, rara vez todos los bytes de la entrada son necesarios para demostrar un fallo, a menos que el caso de prueba pase un proceso conocido como simplificación de entradas [99].

La herramienta utiliza un sencillo proceso para determinar la relación de cada byte de la entrada con el fallo: se realiza la mutación aleatoria de cada byte para determinar si el programa deja de fallar o produce algún efecto observable en el fallo mismo (por ejemplo, cambian los valores de los registros). Utilizando esta idea, *XCraft* elabora un *mapa* con la información recolectada de cada byte de la entrada. Específicamente, se clasifica a cada byte en una de la siguientes categorías:

1. **C**: La alteración de estos bytes causa que el programa deje de fallar o cambie

el tipo de falla producida. Los análisis posteriores deben evitar modificar estos sectores de la entrada.

2. **N**: La alteración de estos bytes no causa ningún efecto en el estado final del programa o el tipo de falla producida. Los análisis posteriores pueden modificar estos sectores de la entrada, si fuera necesario.

3. **M**: La alteración de estos bytes no evita que el programa falle, pero causa efectos directos en los atributos del programa de manera de que permite controlar el valor de algunos registros o posiciones de memoria.

Este sencillo algoritmo es lineal sobre el tamaño de la entrada del programa que causa el fallo y puede ser restringido a ciertos sectores de la entrada que se presupone son relevantes al fallo (por ejemplo, la cabecera de los archivos de imágenes o sonido).

Luego de que el proceso concluye, el *mapa* de la entrada se utiliza en el resto de las etapas de la herramienta para mejorar y agilizar la detección de nuevos fallos y construir pruebas de concepto de vulnerabilidades. Además, si durante este proceso se descubre un caso de prueba que produzca una falla nunca antes vista, el mismo se almacena en la base de datos de fallos, para su utilización en otras etapas.

Mapeo de Entrada-Salida. En el caso de los bytes que pueden ser modificados para controlar atributos como registros y memoria en un fallo, es esencial determinar cual es la relación entre los bytes marcados con **M** (aquí llamados *entrada*) y los bytes de los atributos de ejecución del fallo que pueden ser controlables (aquí llamados *salida*). Formalmente, sean I y O arreglos de bytes de entrada y salida respectivamente, dado un conjunto S de pares de dichos arreglos, es necesario inferir la familia de funciones f_i para todo i tal que:

$$\forall (I, O) \in S : O[i] = f_i(I)$$

Este análisis también se realiza utilizando un enfoque caja negra que aproxima o estima el comportamiento del programa en función de lo que se observa. Este enfoque se opone diametralmente al análisis simbólico, que realiza el cómputo exacto de f pero que puede resultar innecesariamente costoso debido a la complejidad subyacente en los programas y la falta de código fuente.

Para poder mantener el cálculo necesario para este análisis acotado, es necesario restringir la variedad de las funciones f que se pueden inferir. La implementación actual permite deducir las funciones de la forma $f_i(I) = I[j + i]$. La metodología para determinar f también utiliza mutaciones de bytes individuales: si al modificar el byte $I[j + i]$ y relanzar el programa, el fallo muestra que el atributo $O[i]$ resulta idéntico, *XCraft* infiere que $f_i(I) = I[j + i]$. De todas maneras, la herramienta es fácilmente extensible para poder utilizar una variedad de funciones distintas.

Finalmente, en este punto, *XCraft* ya tiene toda la información necesaria del fallo para proceder a las siguientes etapas. La información recolectada sobre el comportamiento del programa al modificar su entrada resulta fundamental tanto para la detección de fallos y vulnerabilidades como para la generación automática de *exploits*.

4.3.3. Detección de Fallos

En la primera etapa, *XCraft* intenta detectar la mayor cantidad de casos de prueba que produzcan fallos en el programa analizado: es claro que a mayor cantidad de fallos, mayor será la posibilidad de generar un *exploit* funcional. La herramienta utiliza *fuzzing* de caja negra o caja gris para obtener fallos o explorar variantes de los que ya posee. Otras técnicas de detección de fallas pueden integrarse en *XCraft* con el único requerimiento que el resultado sea un caso de prueba que genere un fallo. Los fallos que se descubran se individualizan y clasifican para evitar la excesiva redundancia, utilizando la información de pila de llamados a función. Esta información está disponible incluso en programas sin símbolos de depuración, por eso se utiliza. Todos los fallos descubiertos se guardan en una base de datos para su rápido acceso.

Intuitivamente, los fallos de un programa indican que el mismo se encuentra en un estado inválido o inconsistente. Tal como se vio en el ejemplo de `ppthtml` en la Sección 3.2.3, un fallo de desbordamiento de enteros puede iniciar una cadena de errores que permita la ejecución de código arbitrario por parte de un atacante. Siguiendo esta idea, *XCraft* busca acumular un gran número de fallos debido a que alguno de ellos puede resultar en una vulnerabilidad fácilmente explotable. El resultado de esta etapa es un conjunto *enriquecido* de fallos. En la implementación actual, los fallos son detectados usando dos estrategias rápidas y eficientes de mutación de entradas:

- **Mutación aleatoria de bytes**: Dado una entrada, la herramienta produce

una mutación aleatoria de un byte elegido al azar. La única restricción es que la mutación nunca introduce el byte nulo, debido a que es frecuente que dicho valor produzca que las funciones de copiado de cadenas de texto acorten sus tamaños.

- **Extensión aleatoria de bytes:** Dada una entrada, la herramienta *extiende* un byte aleatorio un gran número de veces repitiendo el valor elegido. Por ejemplo, dado la cadena de texto "ab", esta mutación genera "aaaab" y "abbbb" si realiza una extensión de 3 bytes. Para poder generar fallos en programas, *XCraft* produce extensiones de entre 300 y 10000 bytes. Si la entrada extendida se copia en un *buffer* pequeño, producirá un desbordamiento. Este tipo de mutaciones además presenta las siguientes ventajas:

  - Si el byte extendido es parte de una cadena de texto, el resultado siempre será otra cadena de texto válida. A menos que se extienda el último byte ('\0'), esta mutación no producen bytes nulos y por lo tanto es probable que la cadena extendida sea preservada en las operaciones que manejan este tipo de datos (por ejemplo, `strcpy`).
  - Si los bytes extendidos sobreescriben un campo de 32-bits o 64-bits, el resultado será 0 en el caso de extender un byte nulo o un número muy grande, mayor o igual a `0x01010101`, o sea 16843009.
  - Si el byte extendido forma parte de un ruta de archivos válida, el resultado también resultará una ruta válida[1], pero de gran longitud.

Estas sencillas estrategias buscan explorar las distintas variantes de los mismos fallos, de manera de generar nuevos casos de pruebas que descubran vulnerabilidades.

Una técnica adicional se utiliza para descubrir nuevos errores que no hayan sido encontrados por el *fuzzing*: **la *reparación* de punteros inválidos**. Intuitivamente, si podemos controlar un registro con un puntero que está causando un fallo de la lectura o escritura de memoria, podríamos *repararlo* para poder permitir que el programa continúe ejecutando y quizá descubrir una vulnerabilidad. Esta técnica sólo se aplica si el programa exhibe un fallo de acceso de memoria inválido y se identificaron correctamente qué bytes de la entrada controlan el atributo que causa el

[1]Nótese que //// ... //// es una ruta válida que apunta al directorio raíz del sistema.

fallo tal como se explica en la Sección 4.3.2. La *reparación* de punteros es una técnica muy especializada de *fuzzing*, es por eso que se aplica en contextos limitados, sin embargo su uso resulta muy útil para descubrir vulnerabilidades que son obstruidas por un simple acceso de memoria inválido. Veamos un ejemplo de esto. Supongamos un programa que falla en la instrucción:

```
mov -0x8(%ebx),%eax
```

donde `%ebx` contiene `0x45444342`, y que la entrada que genera este fallo sea:

valor	...	0x40	0x41	0x42	0x43	0x44	0x45	0x46	...
mapa	...	C	C	**M**	**M**	**M**	**M**	N	...
posición	...	120	121	**122**	**123**	**124**	**125**	126	...

donde los bytes en las posiciones 122, 123, 124 y 125 determinan directamente el valor del registro `%ebx` durante el fallo. En este caso podemos aplicar la *reparación* de punteros para evitar el fallo de acceso a memoria redireccionando la lectura de la memoria en `0x4544433a` (este valor equivale a substraer `0x8` a `0x45444342`) al comienzo de la pila `0xffffaa00`. Debido a que no se posee más información sobre la operación que el programa realiza con el valor leído luego de la instrucción que falla, no es posible determinar que valor debe ser leído. Es por eso que el valor del nuevo puntero sólo debe contener memoria con permisos de lectura. Finalmente, la entrada resultante sería:

valor	...	0x40	0x41	0x08	0xaa	0xff	0xff	0x46	...
mapa	...	C	C	**M**	**M**	**M**	**M**	N	...
posición	...	120	121	**122**	**123**	**124**	**125**	126	...

Es claro que un programa que falla por un acceso inválido de memoria ya está en una estado inválido. Esta técnica apunta a forzar al programa a continuar ejecutando instrucciones con la esperanza de descubrir una vulnerabilidad que nos permita ejecutar código arbitrario, como explicaremos en la siguiente etapa.

4.3.4. Generación Automática de *Exploits*

Detección de Vulnerabilidades. *XCraft* se centra en la generación de casos de prueba que muestren una vulnerabilidad en un programa. Actualmente la herramienta posee dos módulos para la detección de distintos tipos de vulnerabilidades comunes: desbordamiento de *buffer* para la ejecución de código arbitrario e inyección de código de sistema.

Una vulnerabilidad de corrupción de memoria se detecta cuando un programa falla con un error de lectura de la siguiente instrucción y es posible controlar directamente el valor de dicho `IP` mediante ciertos bytes de la entrada. Por otro lado, una vulnerabilidad de inyección de código requiere que el atacante puede controlar al menos 4 caracteres consecutivos de un parámetro de una función que ejecute programas por medio del sistema operativo (por ejemplo, mediante las funciones `system` o `exec`).

Generación Automática de *Exploits*. Para poder mantener a *XCraft* como una herramienta eficiente en la tarea de generación automática de *exploits*, todos sus pasos deben resultar eficientes. Es por eso que técnicas más tradicionales para este tipo de tareas como la instrumentación de programas o el uso de demostradores de fórmulas lógicas no se utilizaron. En vez de eso, se utilizaron técnicas de caja negra como el *fuzzing* o el mapeo de entrada-salida de manera que se ejecuten a velocidad nativa.

Para poder generar un *exploit* de corrupción de memoria con *XCraft* que ejecute nuestro código *shellcode*, elaboramos un sencillo procedimiento basado en instructivos donde se explican cómo elaborar *exploits* de este tipo de vulnerabilidades [67]. Para empezar, es necesario que se cumplan las siguiente precondiciones para un caso de prueba denotado como $\tilde{I}$:

- El programa debe fallar con un puntero de instrucción inválido al ejecutar P con la entrada $\tilde{I}$.
- El registro del puntero de instrucción debe ser directamente controlable utilizando un determinado índice de $\tilde{I}$. Denotamos a este posición i_{ip}.

Si se cumplen dichas precondiciones, los pasos para generar un *exploit* dado un

programa P, un caso de prueba $\tilde{I}$ y un determinado *shellcode* a ejecutar C_{sc}, son los siguientes:

1. Primero se debe averiguar la dirección de memoria de C_{sc} donde se almacenarán las instrucciones inyectadas que el atacante quisiera ejecutar. En nuestra herramienta, la inyección se implementa con una variable de entorno llamada `SC` utilizada al lanzar el caso de prueba vulnerable, de esta manera nos aseguramos que nuestro *shellcode* esté localizado en la memoria del programa. Denotamos esta dirección de memoria como $\&sc$. Esta operación se automatiza mediante un depurador.

2. Se deben reemplazar los bytes que controlan la siguiente instrucción del puntero a instrucción en la posición correspondiente de $\tilde{I}$ por la dirección del código a ejecutar $\&sc$. El resultado es la entrada $\tilde{I}_{exp}$ definida de la siguiente manera:

$$\tilde{I}_{exp} = \begin{cases} \&sc & \text{si } i = i_{ip} \\ \tilde{I}[i] & \text{si } i \neq i_{ip} \end{cases}$$

3. Finalmente, el *exploit* generado puede ser verificado ejecutando $P(I_{exp})$ utilizando una variable de entorno `SC` que contiene C_{sc}. Por ejemplo:

```
$ SC=shellcode ./programa -arg1 -arg2 exploit.input
```

El proceso para la generación de un *exploit* de inyección de código de sistema es similar al procedimiento previamente descripto. Este proceso también fue definido ad hoc para funcionar en *XCraft* basado en los pasos necesarios para inyectar código que realizan los expertos. Para esto se deben tener en cuenta que se cumplan las siguiente precondiciones para el programa P y un caso de prueba denotado como $\tilde{I}$:

- El programa debe llamar a una función f_{cmd} que ejecute programas externos al utilizar la entrada $\tilde{I}$.

- La memoria que contiene el argumento con los comandos de la llamada de f_{cmd} deben tener al menos 4 bytes directamente controlables utilizando determinado índice de $\tilde{I}$. Denotamos a este posición i_{cmd}.

Si se cumplen dichas precondiciones, los pasos para generar un *exploit* dado un programa P, un caso de prueba $\tilde{I}$ y un determinado comando de sistema a ejecutar C_{sc}, son los siguientes:

1. Se deben reemplazar los bytes que controlan el argumento con los comandos de la llamada a f_{cmd} en la posición correspondiente de $\tilde{I}$ por el código a ejecutar C_{cs}. El resultado es la entrada $\tilde{I}_{exp}$ definida de la siguiente manera:

$$\tilde{I}_{exp} = \begin{cases} C_{cs} & \text{si } i = i_{cmd} \\ \tilde{I}[i] & \text{si } i \neq i_{cmd} \end{cases}$$

2. Finalmente, el *exploit* generado puede ser verificado ejecutando $P(\tilde{I}_{exp})$.

4.3.5. Resultados

Utilizando un conjunto inicial de fallos de programas de línea de comandos provenientes del sistema operativo *Debian*, fue posible detectar y comprobar un centenar de vulnerabilidades en base a los *exploits* generados por *XCraft*. Estos resultados se realizaron generando *exploits* que permitían ejecutar comandos de sistema y resultaron funcionales (es decir, se verificó que los comandos inyectados se ejecuten correctamente).

Las Tablas 4.2 y 4.3 muestran una selección de las vulnerabilidades encontradas y los *exploits* que se pudieron generar correctamente. También se incluye información importante sobre el tiempo que la herramienta utiliza para realizar cada etapa y el número de ejecuciones realizadas. Por un lado, el tiempo de exploración refiere al lapso que la herramienta utiliza en detectar nuevos fallos. Por otro lado, el tiempo de explotación se refiere al lapso que la herramienta utiliza en recolectar la información necesaria para la generación de un *exploit*. En base a los resultados obtenidos podemos afirmar que *XCraft* puede obtener *exploits* funcionales en cuestión de minutos.

4.3.6. Limitaciones

Tal como se mencionó en las secciones anteriores, el enfoque caja negra utilizado por *XCraft* tiene limitaciones inherentes. Para empezar, el análisis de la causa del fallo

Programa	Tamaño de entrada	Vulnerabilidad	Tiempo de exploración (seg)	Tiempo en explotación (seg)	Ejecuciones
worklog	2100	Corrupción de memoria	-	17.87	2340
usage	1224	Corrupción de memoria	3.15	0.12	816
testlp3_gmp	2100	Corrupción de memoria	10.93	0.32	2407
testlp3	2100	Corrupción de memoria	12.45	0.32	2447
testcdd1	2100	Corrupción de memoria	11.36	0.32	2439
trisetcmp	1224	Corrupción de memoria	2.17	13.73	1797
smujajgau	2100	Corrupción de memoria	17.33	16.92	4675
skyeye	2100	Corrupción de memoria	7.60	18.42	2655
decomment	10000	Corrupción de memoria	-	0.48	5
radacct	5000	Corrupción de memoria	551.15	46.22	69034
rstartd.real	10000	Corrupción de memoria	111.25	103.91	20616
ntpq	10000	Corrupción de memoria	1.48	183.95	10328
ntpcd	10000	Corrupción de memoria	399.57	190.82	32161
pprof	2100	Corrupción de memoria	-	0.17	5
pforth	324	Corrupción de memoria	36.83	17.09	6778
opldecode	2048	Corrupción de memoria	16.85	16.06	4578

Tabla 4.2: *Exploits* generados utilizando *XCraft*.

Programa	Tamaño de entrada	Vulnerabilidad	Tiempo de exploración (seg)	Tiempo en explotación (seg)	Ejecuciones
repeat-match	2100	Corrupción de memoria	12.12	45.45	4714
setupnash	2100	Corrupción de memoria	17.77	17.49	4691
setupnash2	1224	Corrupción de memoria	17.53	10.69	3670
hrepack	2400	Corrupción de memoria	10.48	21.86	5069
icon_ dump_ file	10000	Corrupción de memoria	-	0.38	5
gipddecode	2048	Corrupción de memoria	-	16.29	2286
usepackage	500	Corrupción de memoria	3.80	6.44	1333
e2ps	4	Inyección de comandos	-	-	1
gtk-theme-switch2	4	Inyección de comandos	-	-	1
icmake	4	Inyección de comandos	-	-	1
graphviz utils	10024	Corrupción de memoria	8.68	0.35	2360
index_ tar	2100	Corrupción de memoria	8.16	0.34	2367
mpeg3 utils	2100	Corrupción de memoria	-	18.55	2343
ppthtml	207460	Corrupción de memoria	-	65.05	1104
trueprint	2100	Corrupción de memoria	41.16	17.89	7041
trcsort	311810	Corrupción de memoria	-	419.90	11240
text2sf	2100	Corrupción de memoria	17.79	18.11	4694
xymon	5007	Corrupción de memoria	9.08	55.74	9720

Tabla 4.3: *Exploits* generados utilizando *XCraft*.

puede ser insuficiente para identificar errores donde el programa impone restricciones complejas sobre sus entradas. Si el análisis no tiene éxito, entonces la generación de *exploit* no es posible y es necesario buscar otra variante del mismo fallo que nos permita recolectar esta información. Otra posible limitación puede ser causada por una incorrecta suposición del mapeo de entrada-salida.

Incluso si todos los pasos previos a la generación de *exploits* pudieron realizarse exitosamente, el *exploit* resultante puede no resultar funcional. Por ejemplo, en uno de los experimentos realizados para generar un *exploit* de un programa que intentaba interpretar un archivo xml de manera simplista, *XCraft* identificó correctamente la parte de la entrada que permitía controlar el registro del `IP`. A continuación, procedió a generar un *exploit* con la información recolectada. No obstante, el resultado no pudo verificarse, porque el programa no se ejecutaba de la manera esperada. El mismo se detenía antes de llegar al código vulnerable porque detectaba cierto sector del archivo de entrada necesario para el *exploit* como xml inválido. De esta manera, el resultado es una vulnerabilidad potencialmente peligrosa detectada, pero imposible de verificar sin utilizar un enfoque de caja blanca que nos permita concluir si es realmente posible ejecutar código inyectado por un atacante o no.

También es importante aclarar que los *exploits* generados con *XCraft* no están hechos para ser utilizados con las medidas de protección más comunes de los sistemas operativos modernos tales como W^X y ASLR. Los *exploits* generados son considerados pruebas de concepto que tienen como objetivo evidenciar que la falla descubierta puede resultar en una vulnerabilidad y por lo tanto debe ser corregida lo antes posible.

4.3.7. Comparativa

El hecho de que las herramientas para la generación automática de *exploits* no están disponibles públicamente evita que se puede realizar una comparativa. Por lo general, este tipo de herramientas son vistas como demasiado valiosas para un potencial atacante de un sistema, por lo tanto rara vez se publican.

No obstante, en los últimos años, las competencias para comparar técnicas automáticas de seguridad defensiva y ofensiva tales como el *Cyber Grand Challenge* [18] organizadas por DARPA resultaron en un gran aumento del interés en el desarrollo

de este tipo de herramientas. En este contexto, podemos destacar que recientemente los creadores de *angr* [83], una librería de Python para realizar análisis estático y ejecución simbólica, publicaron una nueva herramienta llamada *rex* [15]. Esta permite la generación automática de *exploits* de algunos tipos de vulnerabilidades comunes. En este sentido, es similar a *XCraft*. Sin embargo, la implementación actual no permite la generación de *exploits* en programas que se ejecuten en sistemas como *Linux* o *Windows*, sino que se centra en un tipo de ejecutables específicos llamados *DECREE* sólo utilizados en el *Cyber Grand Challenge*.

4.3.8. Implementación

XCraft se implementó en *Ocaml*, un lenguaje que combina los paradigmas de programación funcional e imperativa con un sistema de tipado fuerte. Para obtener la información de los atributos de ejecución (registro, memoria, etc) cada vez que se descubre un nuevo fallo, se utilizó `ptrace`. En particular, resultó útil un paquete para interactuar con este formalismo denominado `otrace`.

Otra librería fundamental para la implementación fue *Binary Analysis Platform [10]* (BAP). Esta plataforma provee un extenso número de funcionalidades para un análisis formal de las instrucciones ensamblador de distintas arquitecturas. Por ejemplo la instrucción:

add %**eax** ,%**ebx**

es traducida por BAP formalmente al siguiente código:

```
t: u32 = R_EBX: u32
R_EBX: u32 = R_EBX: u32 + R_EAX: u32
CF: bool = R_EBX: u32 <t: u32
```

donde todos los operandos involucrados y sus tipos están explícitos. Este análisis nos permitió identificar fácilmente qué instrucciones de ensamblador leen o escriben direcciones de memoria o registros, de manera de implementar correctamente técnicas avanzadas tales como la *reparación* de punteros.

Actualmente, el código de *XCraft* es propiedad exclusiva de la universidad de

Carnegie Mellon y no está disponible públicamente.

4.4. Observaciones importantes

En este capítulo se desarrollaron una serie de nuevas metodologías y herramientas para la detección de fallos, cada una con sus respectivas ventajas y desventajas. En el próximo capítulo veremos un enfoque más genérico para abordar esta importante tarea que se acopla a un procedimiento de descubrimiento de vulnerabilidades y fallos para optimizar su uso mediante técnicas de aprendizaje automatizado.

Capítulo 5

VDiscover

En este capítulo definiremos y evaluaremos algunas técnicas de aprendizaje automatizado con el objetivo de predecir si un caso de prueba de software puede esconder algún tipo de vulnerabilidad. Esta sección introduce ***VDiscover***, una herramienta para la detección rápida de casos de prueba potencialmente vulnerables utilizando aprendizaje automatizado. Dicha herramienta utiliza una metodología de dos fases: entrenamiento y predicción.

En la fase de entrenamiento, un gran número de casos de prueba son recolectados de varios programas en un conjunto de datos para el entrenamiento. Estos se analizan usando algún proceso de detección de vulnerabilidades. De esta manera se averigua si los casos de prueba de entrenamiento deben ser marcados como vulnerables o no. Luego, estos casos de prueba son caracterizados por información estática extraída de los archivos ejecutables desensamblados y por información dinámica extraída del análisis de las ejecuciones mismas. Esta información se obtiene analizando los patrones de uso de la librería estándar de C. El objetivo de esta fase es utilizar la información del análisis estático y dinámico junto con el procedimiento de detección de vulnerabilidades (PDV) para entrenar un predictor usando aprendizaje automatizado supervisado.

Finalmente, en la fase de predicción, un predictor entrenado se utiliza para determinar si un caso de prueba nunca antes visto será probablemente clasificado como vulnerable o no por el PDV. Un caso de prueba

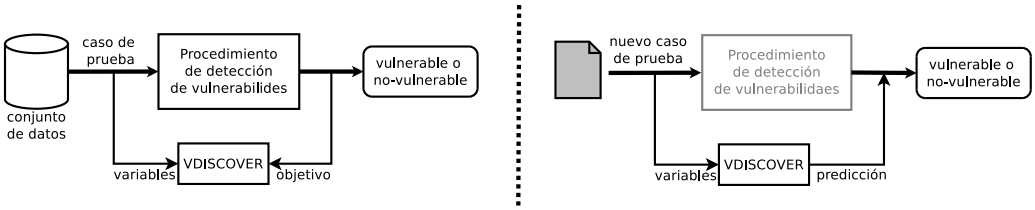


Figura 5.1: Diagrama de las fases de entrenamiento y predicción en *VDiscover*.

marcado como vulnerable en esta fase será priorizado para un análisis más profundo. Es importante destacar que este enfoque no reemplaza totalmente el uso de los procedimientos de detección de vulnerabilidades, sino que permite predecir qué casos de prueba son más probables que revelen vulnerabilidades de software y así permitir su identificación temprana. La Figura 5.1 sintetiza las dos fases de nuestra técnica.

5.1. Propiedades de los predictores

Nuestra herramienta se enfoca en poder utilizar un gran número de casos de prueba para decidir cuales deben ser estudiados más profundamente en el momento de analizar sus vulnerabilidades. Esto se logra mediante el entrenamiento de un predictor para distinguir entre casos de prueba vulnerables o no vulnerables de manera tan efectiva como sea posible. Es por eso que resulta fundamental definir adecuadamente la forma de determinar la efectividad en la predicción de casos de prueba vulnerables. Mediremos la efectividad de un predictor en términos de los errores prediciendo casos de prueba vulnerables (también conocidos como falsos negativos o errores de tipo II) y los errores prediciendo casos de prueba no vulnerables (también conocidos como falsos positivos o errores de tipo I). *VDiscover* intenta minimizar ambos tipos de errores durante la fase de entrenamiento: por un lado, la reducción de falsos positivos permite descubrir más vulnerabilidades en un tiempo más corto. Por otro lado, la reducción de falsos negativos reduce el número de vulnerabilidades no detectadas en este análisis predictivo.

La extracción y procesamiento de la información a partir de los pro-

gramas y sus ejecuciones también posee características propias, que lo hacen particularmente útil en el momento de utilizarse en situaciones reales. Estas características son:

1. **No se requiere del código fuente**: La información para lograr una buena predicción es extraída a partir del análisis estático y dinámico de los programas ejecutables, permitiendo el uso de esta técnica incluso en sistemas operativos propietarios.
2. **Automatización**: Algunas técnicas de aprendizaje automatizado dependen fuertemente de variables seleccionadas de manera manual para obtener buenos resultados. Típicamente esto requiere de un experto humano que revise una lista de variables candidatas antes o durante la etapa de entrenamiento. En este trabajo, nos enfocamos exclusivamente en variables que pueden ser extraídas y seleccionadas de manera totalmente automática, dado un conjunto de datos lo suficientemente grande.
3. **Escalabilidad**: Debido a que nuestro trabajo se centra en técnicas que se aplican a un gran conjunto de datos, utilizaremos solamente información estática y dinámica que pueda extraerse de manera rápida. Las operaciones costosas como la ejecución instrucción por instrucción fueron específicamente excluidas para mantener la practicidad del enfoque en aplicaciones reales.

5.2. Metodología

Para poder exhibir resultados experimentales utilizando *VDiscover* en la predicción de vulnerabilidades son necesarios dos elementos: (1) un procedimiento de detección de vulnerabilidades concreto y (2) un conjunto de datos sobre los cuales realizar entrenamientos y pruebas. En particular, evaluamos nuestra técnica utilizando un *fuzzer* de manera de detectar fallos de corrupción de memoria fácilmente explotable en un gran conjunto de programas. A pesar de que nuestra evaluación se limita al uso de un *fuzzer* concreto, es importante destacar que este componente

de *VDiscover* es reemplazable por otro procedimiento. De hecho, el uso de criterios adaptables por parte de las técnicas de aprendizaje automatizado provee una conveniente maquinaria para lograr la predicción de vulnerabilidades utilizando otros procedimientos. Para los experimentos a gran escala, utilizaremos unos mil programas distintos provenientes de los repositorios del sistema operativo *Debian* Linux.

5.2.1. Detectando Corrupción de Memoria

Nuestro procedimiento de detección de vulnerabilidades se compone de dos módulos: un *fuzzer* que se encarga de mutar los casos de prueba provistos y un depurador que detecta cuando la memoria del programa se corrompió y resulta fácilmente explotable.

Usamos un *fuzzer* sencillo para mutar los casos de pruebas originales y explorar una gran variedad de fallos. El mismo está inspirado en el *fuzzer* utilizado en *XCraft* y sólo produce dos tipos de mutaciones cada vez que recibe una entrada:

- Remplazo de un byte por otro seleccionado aleatoriamente.
- Repetición de un byte un gran número de veces (entre 1 y 10.000).

Luego de realizar una de estas mutaciones, se ejecuta el programa analizado pero utilizando los datos corruptos. Con este procedimiento, lanzamos una serie de experimentos a gran escala para inducir fallos en los casos de prueba disponibles para entrenamiento. Para cada uno de ellos, se generan 10.000 mutaciones y ejecuciones.

También es necesario detallar un criterio para decidir si un programa resulta potencialmente vulnerable a una corrupción de memoria fácil de explotar. Dicho criterio deberá ser completamente automatizable para poder aplicarlo a un gran número de casos de prueba. Detectar este tipo de vulnerabilidad no es tan simple como puede parecer, si no se cuenta con información interna de los programas como su código fuente o sus símbolos de depuración.

Definimos dos enfoques para detectar potenciales vulnerabilidades de corrupción de memoria: a través de evidencias explícitas e implícitas. Los

indicios explícitos de corrupción de memoria utilizados son los siguientes:

1. Corrupción de memoria de pila: algunos programas ejecutables de Debian fueron compilados con protección de memoria de pila provista por la librería estándar de C de GNU. En estos casos, la corrupción de memoria es detectada por el sistema y el programa aborta. En los casos que no se haya compilado con dicha protección, se inspecciona la secuencia de llamados a función de la pila cuando el programa falló mediante un depurador. Si se encuentra al menos un puntero de retorno inválido, se identifica el fallo como corrupción de memoria de pila.
2. Corrupción de memoria de *heap*: la librería estándar de C de GNU provista en *Debian* incluye una verificación de consistencia de memoria de *heap* para detectar si se ha producido un fallo que afectó esta región de memoria. Utilizamos dicho enfoque para identificar fallos de corrupción de *heap*.

Los indicios implícitos de corrupción de memoria son:

1. Argumentos inválidos en llamados a función: ciertas funciones claves de manejo de memoria tales como `strcpy`, `memcpy`, `fread`, `fwrite` entre otras son inspeccionadas cada vez que se ejecutan. Si los argumentos que se utilizan son inválidos o poco frecuentes, por ejemplo copiar un enorme cantidad de memoria (más de 2^{24} bytes), entonces se identifica el fallo como posible corrupción de memoria.
2. Corrupción del puntero a instrucción: cuando ocurre un fallo, inspeccionamos si el registro que contiene el puntero a función apunta a una página inválida o sin permisos de ejecución.

5.3. Conjunto de datos

Para poder utilizar un enfoque de aprendizaje supervisado en la predicción de vulnerabilidades es necesario aprender de un numeroso conjunto de casos de prueba etiquetados. Desafortunadamente, antes de este

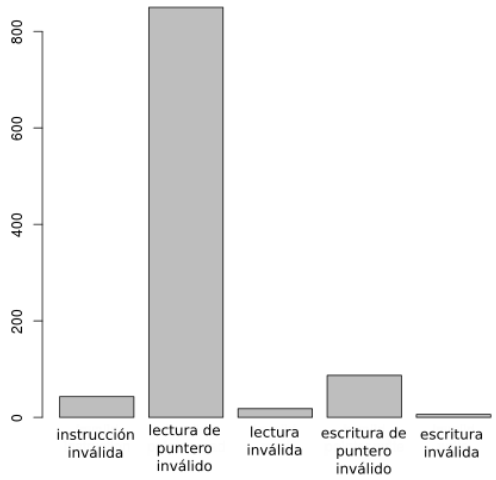


Figura 5.2: Sumario de las causas de los errores en programas de *VDiscovery*.

trabajo, no existía ningún conjunto de datos utilizable para dicha tarea. Esta necesidad fue la principal motivación para definir y recolectar nuestro propio conjunto de datos: ***VDiscovery.***

Este conjunto de datos fue creado analizando 1039 casos de prueba tomados directamente del *bug tracker* del sistema operativo *Debian GNU/Linux*. Cada caso de prueba utiliza un programa ejecutable distinto proveniente de 496 paquetes de *Debian* distintos. Estos casos de prueba fueron recolectados previamente con la herramienta de ejecución simbólica *Mayhem* [13], que fue utilizada para detectar fallos en programas de *Debian* de manera masiva por investigadores de CMU [87]. Los programas incluidos en *VDiscovery* son de diversa naturaleza: se incluyen, por ejemplo, herramientas científicas para procesamiento de datos, juegos simples, programas de escritorio e incluso un reconocedor óptico de caracteres (OCR).

5.3.1. Análisis de Causa Directa

Antes de comenzar a utilizar este conjunto de datos, se realizó un análisis de causa directa de cada fallo, similar al descripto en la Sección

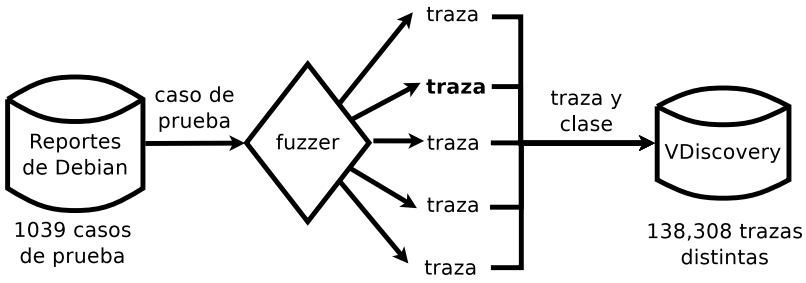


Figura 5.3: Extracción y etiquetado de trazas en *VDiscovery*.

4.3.2. Un resumen de los errores que componen nuestro conjunto de datos se encuentra en la Figura 5.2. La mayoría de los errores son causados por lecturas de memoria utilizando punteros inválidos (aquellos que apuntan a páginas de memoria no mapeada). Este caso incluye errores comunes como dereferencia de punteros nulos.

Es interesante notar que un gran número de errores son causados por intentar ejecutar una instrucción inválida. Es tentador pensar que el ejecutador simbólico encontró una corrupción de memoria que de alguna manera causó dicho error. Desgraciadamente este fenómeno es una falla en la recolección de datos en el origen. Una inspección manual de estos casos de prueba revela que los programas ejecutables que fallan en realidad son librerías dinámicas que poseen el bit de ejecución activo. Si se intenta ejecutar una librería dinámica, el resultado fallará siempre ya que este tipo de archivos no define un procedimiento principal. Estos casos fueron identificados y descartados antes de realizar los análisis planteados en las secciones que siguen, ya que no proporcionan información útil.

5.3.2. Clases

A primera vista, el resumen de las fallas de la Figura 5.2 muestra que los errores recolectados en nuestro conjunto de datos parecen ser poco interesantes desde el punto de vista del descubrimiento de vulnerabilidades. La primera causa de errores resulta ser la lectura de memoria inválida. Este tipo de errores generalmente es catalogado de baja prioridad en cuestión de seguridad del sistema debido a lo difícil que puede ser para

un atacante utilizar estos errores a su favor. Sin embargo, utilizando el procedimiento de descubrimiento de vulnerabilidades definido en la Sección 5.2.1 procedimos a dividir los casos de prueba en "identificados como vulnerables" y "no identificados como vulnerable". En este sentido, un caso de prueba se dice "identificado como vulnerable" si existe al menos una traza que exhiba un patrón de corrupción de memoria como se detalló en la sección anterior. El proceso se encuentra esquematizado en la Figura 5.3.

En *VDiscovery*, sólo el 8 % de los casos de prueba fueron identificados como vulnerables. Debido a esto, es esperable que los datos estén desbalanceados. Este problema deberá abordarse antes de entrenar un predictor utilizando este conjunto de datos.

5.3.3. Variables

En el presente trabajo, dos conjuntos de características se definieron y evaluaron: aquellas extraídas de información dinámica de los casos de prueba y aquellas extraídas directamente de programas mediante análisis estático. Ambos conjuntos intentan capturar el uso de patrones de la librería estándar de C por parte del programa analizado. Esta información se estructura en secuencias de longitud variable y puede estar correlacionada con la existencia de vulnerabilidades de corrupción de memoria como las planteadas en la Sección 5.2.1. Será tarea de los métodos de aprendizaje automatizado descubrir esa sutil y compleja relación para lograr la predicción de los casos de prueba que deberán ser analizados posteriormente.

Adicionalmente, para cada conjunto de variables definidas en esta sección, se incluye una descripción del costo computacional de manera de justificar su uso a gran escala.

Ejemplo: Para explicar mejor la extracción de variables dinámicas y estáticas utilizaremos un fragmento de código ensamblador de uno de los programas que componen nuestro conjunto de datos. En la Figura 5.4, se

```
1  call getenv
2  test %eax,%eax
3  je @15
4  lea -0x100c(%ebp),%ebx
5  mov %eax,0x4(%esp)
6  mov %ebx,(%esp)
7  call strcpy
8  mov $0x123,0x4(%esp)
9  mov %ebx,(%esp)
10 call strtok
11 ...
12 ...
13 ...
14 ...
15 ret
```

Figura 5.4: Fragmento de código ensamblador proveniente de `xa`.

muestra parte del código de `xa`, un sencillo ensamblador cruzado para los procesadores de 8-bits conocidos como 65xx (por ejemplo, el utilizado por la *Commodore 64*). Seleccionamos parte de su código para ejemplificar porque realiza algunos llamados a funciones estándar de C.

5.3.3.1. Variables Estáticas

Las variables estáticas capturan la información relacionada con el código del programa en su totalidad y se obtienen utilizando análisis sin ejecutar el mismo. Las técnicas de análisis estáticos clásicas utilizan representaciones basadas en grafos para expresar la estructura del código. Entre los ejemplos de estas estructuras se incluyen los grafos de llamadas a función, de control y de flujo de datos. Sin embargo, estas técnicas suelen ser computacionalmente costosas y muchas veces resulta imposible extraerlas precisamente del código ejecutable sin utilizar símbolos.

El enfoque que proponemos en esta sección *aproxima* la estructura del código mediante una secuencia finita de llamadas a función de la librería estándar de C. Dichas secuencias pueden verse como una abstracción del

grafo de llamadas a función donde se utilizan sólo algunas funciones y su estructura se aplana.

Estas variables estáticas se extraen directamente de los programas ejecutables usando técnicas de análisis estático liviano. Primero, el binario completo se desensambla utilizando un algoritmo de barrido lineal. El conjunto I de llamados directos a función de la librería estándar de C se extrae utilizando el código desensamblado. Los elementos del conjunto I se utilizarán como punto de partida para una exploración aleatoria del grafo de flujo de control del programa analizado. Se construye el conjunto S de secuencias de llamadas a función utilizando repetidamente el algoritmo descripto a continuación:

1. Se selecciona un elemento c perteneciente a I y se lo inserta en una secuencia vacía σ;
2. Se sigue la secuencia de instrucciones siguientes a c en el código desensamblado:
   - si se encuentra una llamada a una función de la librería estándar de C, se agrega dicha llamada a la secuencia σ y se continúa con la siguiente instrucción:
   - si se encuentra un salto incondicional a la dirección x, se continúa con la instrucción en la dirección x;
   - si se encuentra un salto condicional, se selecciona aleatoriamente una de las ramas y se continúa con las instrucciones en dicha rama;
   - finalmente, si ninguna de las anteriores condiciones se cumple, se continúa con la próxima instrucción;
3. Si en el paso anterior se localiza una instrucción de retorno o un salto indirecto, la secuencia σ se considera terminada y se la agrega al conjunto S

Este simple y eficiente procedimiento extrae subsecuencias de potenciales llamadas a función de la librería estándar de C utilizando un camino aleatorio del grafo de flujo de control del programa analizado.

Ejemplo. En el código presentado en la Figura 5.4, si el proceso de extracción de variables estáticas comienza en el llamado a función `getenv` en (1), dos subsecuencias pueden localizarse, de acuerdo al salto condicional en (3). Entonces, el conjunto S resultante en este caso sería:

$$\{[\texttt{getenv}; \texttt{strcpy}; \texttt{strtok}; \ldots], [\texttt{getenv}]\}$$

Costo Computacional. La extracción de este tipo de variables requiere el desensamblado completo del programa analizado. Luego, un proceso de análisis estático eficiente se realiza. Este proceso no requiere la computación de un estado interno, con lo que sus requerimientos de memoria son mínimos. Además la exploración aleatoria de las secuencias sólo necesita recolectar una muestra de las llamadas a función de la librería estándar de C, por lo que sus requerimientos de procesamiento son moderados.

5.3.3.2. Variables Dinámicas

Las variables dinámicas capturan la información relacionada con muestras del comportamiento del programa en función de secuencias de eventos. Definimos **eventos de programa** a cada llamada a función de la librería estándar de C que el programa realiza y su estado final. Formalmente notamos los eventos de programa de la siguiente manera:

$$fc_i(arg_1, .., arg_n) \,|\, \texttt{EstadoFinal}$$

donde fc_i abstrae los nombres de las funciones de la librería estándar de C y `EstadoFinal` representa una de las posibilidades del estado final del programa de acuerdo a los desarrollado en la Sección 3.1.2:

$$\texttt{Salida} \,|\, \texttt{Aborto} \,|\, \texttt{Fallo}$$

A diferencia de las variables estáticas, las dinámicas requieren de un

caso de prueba: un programa y su correspondiente entrada. La extracción de estas variables se realiza mediante la ejecución de los casos de prueba del programa utilizando un mecanismo que permita detectar y almacenar cada llamada a función relevante en una secuencia. Adicionalmente, se incluye información sobre el estado final del programa obtenido mediante el uso de un depurador.

Otra importante diferencia entre las variables estáticas y las dinámicas es la cantidad de información que puede ser extraída de un caso de prueba debido a que la ejecución pude generar una larga secuencia de eventos. Incluso programas pequeños con un bucle puede generar un número arbitrariamente grande de secuencias.

El uso de variables dinámicas en forma de trazas trae aparejados algunos desafíos. El más importante de ellos es que puede resulta difícil o imposible utilizar técnicas de aprendizaje automatizado sobre los eventos tal como se los definieron aquí, sin utilizar una representación adecuada. Esto es debido a que no está especificado como representar valores en los argumentos de los llamados a función. Por ejemplo, en el caso de que se haya capturado una llamada a la función `memcpy` durante una ejecución, el evento puede verse así:

```
strcpy(0xbfffc0fc, 0x7f1b89a0, 128)
```

Es notable observar que los valores de los argumentos son *de bajo nivel:* por ejemplo números enteros o punteros. Existen dos problemáticas importantes relacionadas con las representaciones de valores de *bajo nivel* para su utilización en aprendizaje automatizado. En primer lugar, estas representaciones necesitan rangos de valores extremadamente grandes (por ejemplo, de tamaño 2^{32} en 32-bits). Esto significa que si queremos representar los eventos como secuencias discretas, es esencial reducir el rango de esos tamaños. En segundo lugar, los valores como números enteros transmiten muy poca información por si mismos al método de aprendizaje automatizado: es necesario agregarles información semántica para poder aprender de los mismos.

Afortunadamente, podemos abordar estas problemáticas utilizando la

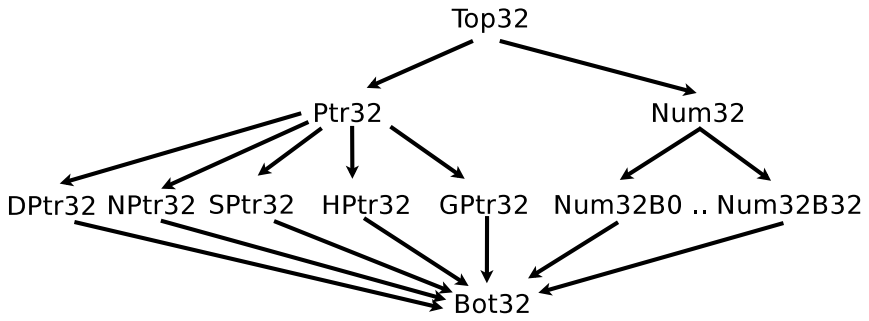


Figura 5.5: Relación de subtipado utilizada para procesar los valores de los argumentos.

especificación de cada función de la librería de C. En el caso de memcpy:

```
void *memcpy(void *dest, const void *src, unsigned int n)
```

Por esta razón, decidimos *marcar* cada valor de los argumentos con una etiqueta que indica su *tipo*, utilizando una relación adecuada de subtipos expuesta en la Figura 5.5. Esta idea está inspirada por las técnicas utilizadas por herramientas para ingeniería inversa como *TIE* [48] y *PointerScope* [100].

En el caso de tener punteros (`Ptr32`), es útil saber a qué región están apuntando. Por ejemplo, `HPtr32` indica que se apunta a la memoria de *heap*, mientras que `SPtr32` indica que se apunta a la memoria de pila y `GPtr32` a la memoria global. También es de gran importancia saber si los punteros son inválidos (`DPtr32`) o nulos (`NPtr32`).

En el caso de los números enteros (`Num32`), estos aportan menos información que los punteros. Por eso puede ser útil diferenciar si se trata de un valor nulo, pequeño o muy grande. Para formalizar estos subtipos, dividimos el rango total de los números de 32-bits en segmentos logarítmicos: el subtipo `Num32B`n indica si el número se encuentra entre 2^n y $2^{(n+1)}$. De esta manera, cuando se trate de valores sospechosamente pequeños o grandes, al leer o escribir bytes con `memcpy`, este tipo de representación puede ser útil para que un algoritmo de aprendizaje automatizado utilice este tipo de datos.

Ejemplo. En el código presentado en la Figura 5.4, luego de una ejecución se captura la siguiente traza expuesta en comparación con la misma traza de *ltrace* [12]:

ltrace	***VDiscover***
`getenv('XAINPUT')`	`getenv(GPtr32)`
`strcpy(0xbfffc0fc, 'input')`	`strcpy(SPtr32,HPtr32)`
`strtok('input', ',')`	`strtok(HPtr32,GPtr32)`

Costo Computacional. La extracción de este tipo de variables requiere la ejecución de un caso de prueba. Para poder obtener las trazas, el programa ejecutable y algunas de las librerías que utiliza deben ser instrumentadas de manera eficiente para poder detectar las llamadas a funciones de librería estándar de C. Adicionalmente se utilizaron optimizaciones para detectar y descartar llamadas a función de ciertas librerías del sistema operativo para evitar que estas complejicen las trazas sin agregar información importante sobre el comportamiento del programa analizado.

5.4. Resultados y Discusión

5.4.1. Preprocesamiento del Conjunto de Datos

Antes de comenzar con el entrenamiento y la predicción de casos de prueba vulnerables usando *VDiscover*, es necesario preprocesar el conjunto de datos utilizado. Este proceso es esencial para poder realizar el entrenamiento de una variedad de predictores utilizando aprendizaje automatizado. Adicionalmente, el preprocesamiento reduce la complejidad de los datos secuenciales de *VDiscovery*, permitiendo que el entrenamiento de predictores se realice más fácilmente.

Para procesar las diferentes variables de *VDiscovery*, se utilizaron tres procedimientos de los campos de procesamiento de texto y minería de datos: la bolsa de palabras, *word2vec* y *fastText* ya explicados en la Sección

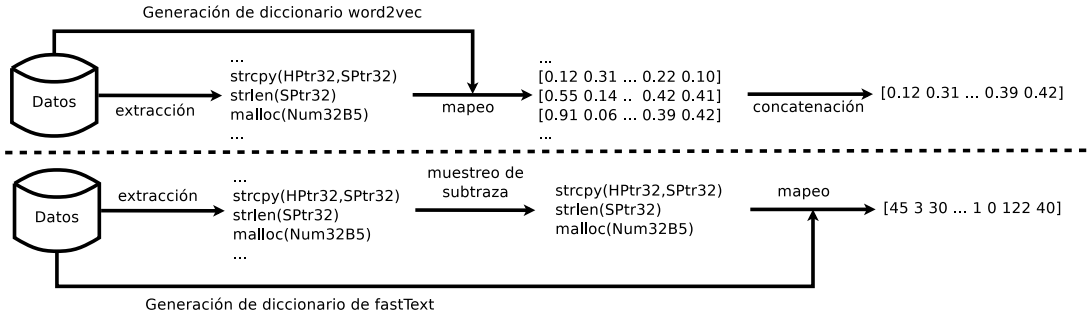


Figura 5.6: Preprocesamiento de las variables dinámicas usando *word2vec* y *fastText*.

2.4. La idea de utilizar técnicas de minería de texto para procesar trazas es similar al enfoque utilizado en otros trabajos que analizan este tipo de datos [72, 95].

El proceso de preprocesamiento depende del tipo de variable que se quiera utilizar y por lo tanto, es necesario separar los procedimientos de preprocesamiento utilizados para las variables estáticas y las variables dinámicas. Por un lado, en el caso de las variables estáticas, se utilizó la **bolsa de palabras** para representar cada traza de nuestro conjunto de datos. A pesar de que este preprocesamiento elimina la información relacionada con el ordenamiento de las subsecuencias de eventos, se utiliza por ser una técnica simple y eficiente: en particular experimentamos utilizando 1-gramas, 2-gramas y 3-gramas de manera de obtener representaciones vectoriales adecuadas para entrenar y evaluar un predictor. Las técnicas más modernas de representación vectorial de texto tales como *word2vec* y *fastText* no se utilizaron con este tipo de variables porque requieren abundantes cantidades de datos.

Por otro lado, en el caso de las variables dinámicas, también utilizamos **bolsa de palabras** para representar cada traza de nuestro conjunto de datos utilizando 1-gramas, 2-gramas y 3-gramas. No obstante, también experimentamos con el uso de *word2vec* como muestra la Figura 5.6: primero, esta técnica se usó para generar una representación vectorial de cada evento posible. Luego, para cada traza, se construyó un vector utilizando los primeros y últimos eventos de la traza experimentando con 20, 50, 100 y 200 eventos concatenados para caracterizar trazas completas.

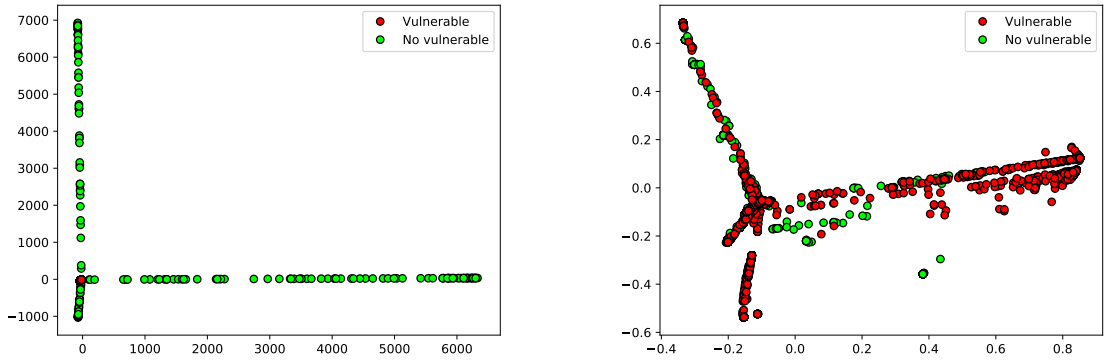
Finalmente utilizamos *fastText* para representar las trazas de manera vectorial. Tal como lo muestra la Figura 5.6, *fastText* muestrea subtrazas y las vectoriza utilizando un diccionario precalculado. Experimentamos extrayendo subtrazas de 5, 15, 50 y 100 eventos.

Estos enfoques para el preprocesamiento fueron utilizados de forma independiente debido a que resultan en vectores completamente distintos para el mismo conjunto de datos. De esta manera podemos determinar cuál es el enfoque más efectivo en la predicción de nuevos datos.

Desbalance de clases. Un serio problema para el aprendizaje de nuestro conjunto es el desbalance de clases. Tal como se mencionó en la Sección 5.3, el número de ejemplos "identificados como vulnerables" constituye solamente el 8 % de nuestros datos. Dicho desbalance es problemático cuando se utiliza un algoritmo de aprendizaje automatizado para intentar aprender patrones de los datos, debido a que el procedimiento de entrenamiento tiende a aprender la información de la clase mayoritaria o sea, los casos "no vulnerables". El resultado de utilizar datos desbalanceados directamente para el aprendizaje en este problema es que ningún caso de prueba será clasificado como vulnerable. Abordamos este problema utilizando sobremuestreo aleatorio [36] para poder facilitar el aprendizaje de datos desbalanceados debido a que la cantidad de datos de entrenamiento es muy acotada para realizar submuestreo.

5.4.2. Exploración de los datos

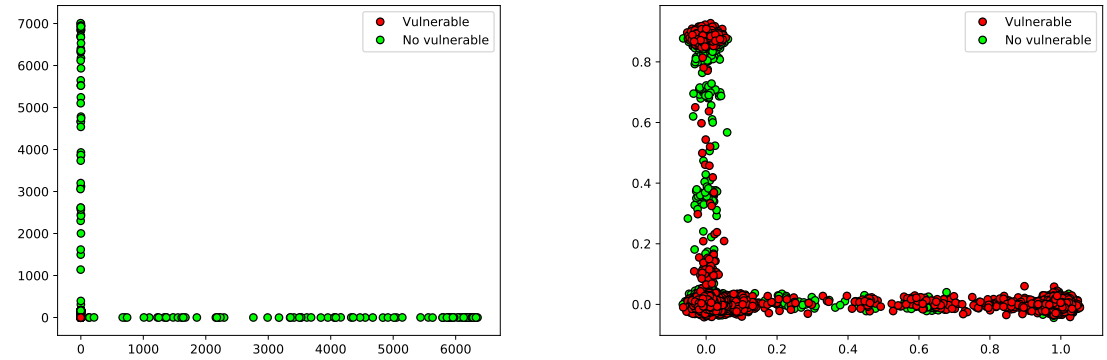
Luego de procesar los datos en crudo extraídos de los programas y las trazas, es interesante detallar algunas propiedades de las representaciones vectoriales. Para esto, se realizó una exploración de los datos utilizando las técnicas de reducción de dimensionalidad y visualización detalladas en la Sección 2.2.1 sobre las representaciones de 1-gramas de *VDiscovery*. Además, para una adecuada visualización de los gráficos, los datos que representan a casos de prueba vulnerables han sido resaltados utilizando un pequeño ruido aditivo para evitar que estos queden ocultos bajo los



(a) Utilizando el conteo directo.

(b) Utilizando la frecuencia inversa.

Figura 5.7: Variables dinámicas visualizadas con PCA.



(a) Utilizando el conteo directo.

(b) Utilizando la frecuencia inversa.

Figura 5.8: Variables dinámicas visualizadas con LSA.

malloc:0=Num32B0	
puts:0=SPtr32	0.45
abort:eip=GPtr32	0.45
StackCorruption	0.45

malloc:0=Num32B8	
memset:0=HPtr32	0.50
strtok:1=GPtr32	0.32
free:0=HPtr32	0.27

malloc:0=Num32B16	
strcat:1=DPtr32	0.32
fwrite:3=NPtr32	0.29
vsnprintf:0=GPtr32	0.28

malloc:0=Num32B32	
signal:0=Num32B8	0.37
getenv:0=HPtr32	0.30
memcpy:2=Num32B32	0.30

Tabla 5.1: Llamadas a la función `malloc` y sus eventos más relacionados de acuerdo a *word2vec*.

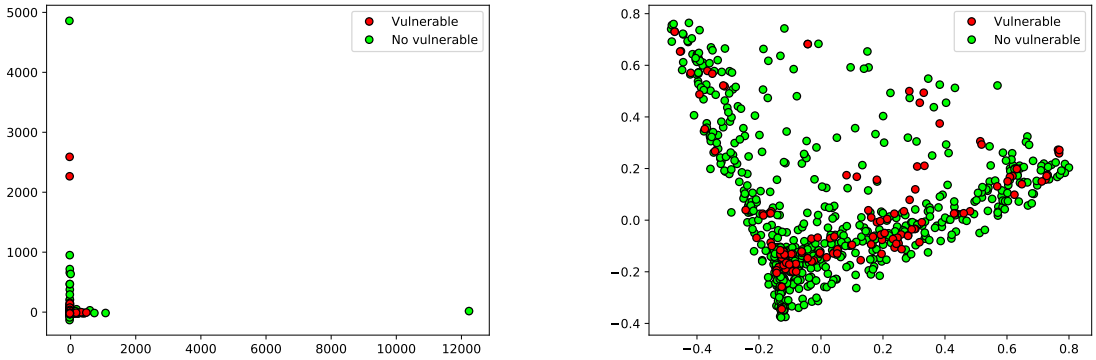
casos no vulnerables debido al gran desbalance del conjunto de datos.

En el caso de las variables dinámicas, las Figuras 5.7a, 5.7b, 5.8a y 5.8b muestran las distribuciones de los casos de prueba vulnerables (en rojo) y no vulnerables (en verde) utilizando conteo directo y frecuencia inversa. En el caso de las variables estáticas, las Figuras 5.9a, 5.9b, 5.10a y 5.10b muestran las distribuciones de los programas vulnerables (en rojo) y no vulnerables (en verde) utilizando conteo directo y frecuencia inversa.

Sin importar la forma en la que se preprocesan y visualizan los datos, se destacan tres características particulares de los mismos: primero, los datos se encuentran superpuestos unos con otros, incluso aquellos de distintas clase. Segundo, no se observa la existencia de agrupaciones claras que permitan separar los casos de prueba vulnerables de los no vulnerables. Tercero, se distingue una variedad de valores atípicos (o *outliers* según su denominación en inglés).

La combinación de estos factores indica que esta tarea de clasificación es particularmente difícil de resolver en función de un determinado número de variables y que la información que se puede extraer de cada caso de prueba tiene un alto nivel de ruido. También es interesante notar que aunque los datos totales parecen suficientes, se aprecia una gran cantidad de redundancia.

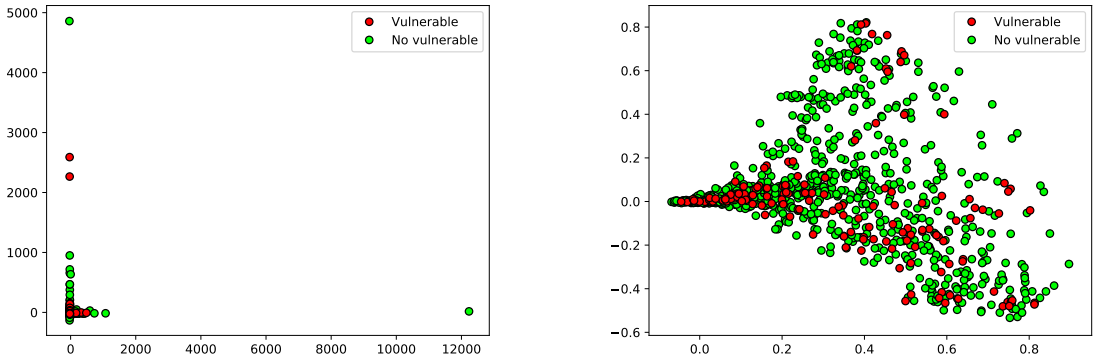
En el caso de las variables dinámicas, una tarea que llevamos a cabo para entender mejor los datos fue explorar las propiedades vectoriales de las representaciones que los métodos como *word2vec* y *fastText* nos brindan. Para esto, se compararon los vectores resultantes de cada evento utilizando una medida de similaridad entre dos vectores. Definimos dicha medida utilizando el coseno del ángulo que dos vectores definen. Entonces, si dos vectores se encuentran a una corta distancia, se dicen que están



(a) Utilizando el conteo directo.

(b) Utilizando la frecuencia inversa.

Figura 5.9: Variables estáticas visualizadas con PCA.



(a) Utilizando el conteo directo.

(b) Utilizando la frecuencia inversa.

Figura 5.10: Variables estáticas visualizadas con LSA.

relacionados.

Por ejemplo, en el caso de las representaciones de cada evento de las variables dinámicas obtenidas con *word2vec* podemos observar como estos se agrupan. En este sencillo análisis, se exploran los eventos relacionados con la reserva de memoria en *heap* a través de la función `malloc` presentes en la Tabla 5.1. Normalmente las llamadas a dicha función se realizan con valores que varían de unas decenas de bytes a centenares de megabytes. En este caso, los eventos *relacionados* parecen normales. En cambio, si una llamada a dicha función reserva un valor nulo (`Num32B0`) o el tamaño máximo en 32-bits (`Num32B32`), los eventos *relacionados* se asocian con comportamientos erróneos. Por ejemplo, una llamada a `memcpy` para copiar una gran cantidad de memoria o un aborto del programa.

5.4.3. Procedimientos de entrenamiento de predictores

Para poder realizar una evaluación independiente de la relevancia de las distintos tipos de variables de nuestro conjunto de datos, se procedió a entrenar distintos tipos de clasificadores utilizando solamente variables estáticas y dinámicas. Para cada uno de estas configuraciones, se realizó un total de 40 experimentos predictivos dividiendo cada muestra del conjunto de datos en tres subconjuntos: entrenamiento, validación y prueba. Es muy importante destacar que estos subconjuntos **no comparten datos de los mismos programas**: de esta manera el entrenamiento del predictor se centra en generalizar los patrones de un programa a otro. Además, los resultados obtenidos resultan más realistas.

En cada experimento, se entrenaron varios tipos de clasificadores basados en aprendizaje automatizado: regresión logística, una red neuronal artificial de una capa oculta y un *random forest*. Se utilizaron paquetes de software de código libre tales como *scikit-learn* [71] y *pylearn2* [32] para el entrenamiento y prueba de los distintos clasificadores.

Entrada	Regresión Logística	Redes Neuronales
200 eventos	38 % ± 1	35 % ± 1
100 eventos	**34** % ± 1	37 % ± 1
50 eventos	35 % ± 1	36 % ± 1
20 eventos	37 % ± 1	35 % ± 1

(a) Variables dinámicas utilizando *word2vec*.

Entrada	Regresión Logística	Redes Neuronales
100 eventos	38 % ± 1	37 % ± 1
50 eventos	36 % ± 1	37 % ± 1
15 eventos	**35** % ± 1	37 % ± 1
5 eventos	37 % ± 1	39 % ± 1

(b) Variables dinámicas utilizando *fastText*.

Entrada	Regresión Logística	*Random Forest*
1-gramas	40 % ± 1	32 % ± 1
1–2-gramas	40 % ± 1	**31** % ± 1
1–3-gramas	40 % ± 1	**31** % ± 1

(c) Variables dinámicas utilizando bolsa de palabras.

Entrada	Regresión Logística	*Random Forest*
1-gramas	38 % ± 1	47 % ± 1
1–2-gramas	**37** % ± 1	46 % ± 1
1–3-gramas	**37** % ± 1	45 % ± 1

(d) Variables estáticas utilizando bolsa de palabras.

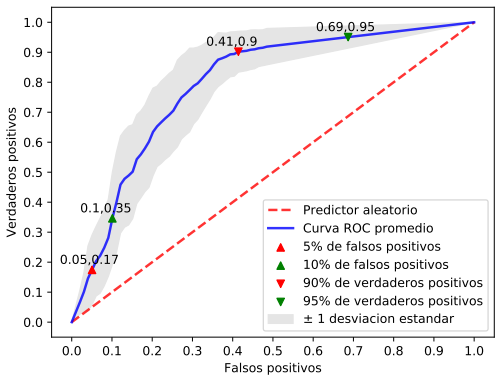
Tabla 5.2: Error de prueba promedio de la predicción de vulnerabilidades usando *VDiscover*.

5.4.4. Resultados experimentales

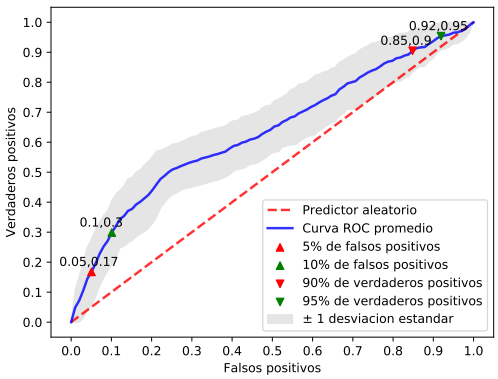
Las Tablas 5.2a, 5.2b, 5.2c y 5.2d muestran los resultados de errores en los subconjuntos de prueba sobre los distintos clasificadores usando variables estáticas y dinámicas. Los experimentos indican que si se utilizan las variables estáticas con la bolsa de palabras, la regresión logística resulta el mejor predictor con 37 % de error de prueba. Por otro lado, utilizando variables dinámicas con *word2vec,* la regresión logística resultó el mejor predictor con 34 % de error de prueba. En el caso de utilizar *fastText*, la regresión logística también fue la mejor opción con 35 % de error de prueba. Finalmente, si se utiliza la bolsa de palabras para transformar los datos, el mejor clasificador resulta un *random forest* con 31 % de error de prueba.

5.4.5. Discusión

Para mostrar en detalle y analizar la efectividad de los mejores predictores utilizando variables estáticas y dinámicas, se presentan las correspondientes matrices de confusión en las Tablas 5.3a y 5.3b en función de los porcentajes de error de los casos que *VDiscover* predice. El clasificador más preciso resulta ser el *random forest* entrenado únicamente con variables dinámicas procesadas con la bolsa de palabras. El error de prueba de este clasificador es de 31 %. Usando este clasificador podemos estimar la reducción del esfuerzo necesario para descubrir nuevas vulnerabilidades.



(a) Utilizando variables dinámicas.



(b) Utilizando variables estáticas.

Figura 5.11: Curvas ROC de los mejores predictores para cada tipo de variables.

	Vulnerable	**No Vulnerable**
Vulnerable	**55** %	17 %
No Vulnerable	45 %	**83** %

(a) Matriz de confusión del mejor predictor utilizando variables dinámicas.

	Vulnerable	**No Vulnerable**
Vulnerable	**52** %	25 %
No Vulnerable	48 %	**75** %

(b) Matriz de confusión del mejor predictor utilizando variables estáticas.

Tabla 5.3: Comparativa de predicción de casos de prueba.

Las Figuras 5.11a y 5.11b muestran las curvas de característica operativa del receptor (o ROC por sus siglas en inglés) de los mejores predictores confeccionadas utilizando el subconjunto de datos de prueba. En lineas generales, se puede apreciar que el clasificador que utiliza variables dinámicas resulta mucho más robusto que el que utiliza variables estáticas. Los valores de áreas bajo la curva confirman esta conclusión, ya que resultan de 0,79 para la Figura 5.11a y 0,63 para la Figura 5.11b.

Para analizar en detalle el comportamiento de los mejores predictores, se definieron dos escenarios en los que los predictores podrían utilizarse. En el primero de ellos, se busca encontrar una vulnerabilidad lo más rápidamente posible, minimizando el esfuerzo. Este escenario podría utilizarse para lograr descubrir una vulnerabilidad sin un gran requerimiento de recursos computaciones en un componente software (por ejemplo, el software frecuentemente actualizado que se utiliza en dispositivos móviles). Esto resulta particularmente útil para un atacante que quiere realizar un mínimo esfuerzo para encontrar la primera vulnerabilidad. Idealmente, se busca reducir los falsos negativos tanto como sea posible, siempre y cuando se descubran vulnerabilidades (o sea, verdaderos positivos). Desgraciadamente ninguna de las curvas ROC muestra que sea posible reducir los falsos negativos totalmente, es por eso que definimos este escenario utilizando un porcentaje falsos negativos de 5 % a 10 %. Entonces, el predictor debe utilizar un umbral de clasificación donde los falsos negativos estén acotados por estos porcentajes, tratando de maximizar los verdaderos positivos lo más posible.

Es interesante notar que ambos predictores tienen comportamientos parecidos en este escenario, con porcentajes de verdaderos positivos entre el 17 % y el 35 %. Utilizando estos datos podemos calcular que tan eficientemente resultan el mejor predictor respecto a seleccionar los casos de prueba para analizar al azar. Si recordamos los porcentajes de programas vulnerables (8 %) y no vulnerables (92 %) especificados en la Sección 5.3, podemos calcular el porcentaje de programas detectados como vulnerables por *VDiscover* mediante un promedio ponderado:

$$8\,\% * \overbrace{0{,}35}^{\substack{\text{verdaderos}\\ \text{positivos}}} + 92\,\% * \overbrace{0{,}10}^{\substack{\text{falsos}\\ \text{positivos}}} = 2{,}8\,\% + 9{,}2\,\% = 12\,\%$$

Intuitivamente, estos números nos indican que si utilizamos el PDV definido en 5.2.1 para analizar el 12 % de nuestro conjunto de prueba señalado por *VDiscover* como potencialmente vulnerable, podemos detectar el 35 % de los casos de prueba vulnerables. Es esperable que sin la ayuda de nuestra herramienta, la elección de casos de prueba al azar requiera analizar el 35 % de los casos de prueba para descubrir el 35 % de las vulnerabilidades. Por lo tanto, utilizando los resultados de nuestros experimentos, podemos cuantificar la mejora en la detección de vulnerabilidades en un 291 % más veloz ($\approx 35\,\%/12\,\%$) respecto a utilizar una selección al azar.

Por otro lado, existe otro escenario donde se busca encontrar el mayor número de vulnerabilidades con el mínimo esfuerzo. Existen una gran variedad de componentes de software crítico donde la sola existencia una fallo puede poner en riesgo a un gran número de usuarios. En estos casos, encontrar todas las vulnerabilidades posibles se vuelve una prioridad. Dicho escenario es particularmente útil para las empresas y gobiernos que quieran defender su infraestructura al mismo tiempo que minimizan los costos computaciones necesarios. Idealmente, se busca maximizar los verdaderos positivos tanto como sea posible, siempre y cuando se evite analizar todos los casos de prueba (o sea, alcanzar un 100 % de falsos negativos). Desgraciadamente ninguna de las curvas ROC muestra que sea posible maximizar los verdaderos positivos totalmente sin analizar todos los casos de prueba, es por eso que definimos este escenario utilizando un porcentaje de verdaderos positivos de 90 % a 95 %. Entonces, el predictor debe utilizar un umbral donde los verdaderos positivos estén acotados por estos porcentajes, tratando de minimizar los falsos positivos lo más posible. En este escenario, la Figura 5.11a muestra un predictor mucho más robusto que el de la Figura 5.11b. De hecho, este último predice

prácticamente al azar. Por lo tanto, el uso de variable estáticas en este escenario no resulta adecuado. Nuestra hipótesis es que las vulnerabilidades más difíciles de prever sólo pueden descubrirse utilizando información dinámica y es por eso que las variables estáticas tienen una baja precisión. Al igual que en el escenario anterior, podemos calcular el porcentaje de programas detectados como vulnerables por *VDiscover* mediante un promedio ponderado:

$$8\,\% * \overbrace{0{,}90}^{\substack{\text{verdaderos}\\ \text{positivos}}} + 92\,\% * \overbrace{0{,}41}^{\substack{\text{falsos}\\ \text{positivos}}} = 7{,}20\,\% + 37{,}72\,\% = 44{,}92\,\%$$

Por lo tanto, utilizando los resultados de nuestros experimentos, podemos cuantificar la mejora en la detección de vulnerabilidades en un 200 % más veloz ($\approx 90\,\%/44{,}92\,\%$).

5.5. Evaluación

Luego de realizar el correcto entrenamiento en base a los datos disponibles, es importante investigar en detalle de que manera los clasificares diferencian y clasifican los casos de prueba. En particular, para analizar qué tan robustos son los clasificadores, es esencial identificar qué variables resultan más importantes en la clasificación y cómo se utilizan.

Para analizar en detalle la importancia de cada variable, primero resulta útil definir un subconjunto especial de las variables dinámicas: las variables directamente relevantes al PDV utilizado. En nuestros experimentos, estas variables corresponden a aquellas directamente correlacionadas con la existencia de vulnerabilidades de corrupción de memoria tal como fue explicado en la Sección 5.2.1. Entre estas variables se incluyeron aquellas que representan llamados a funciones que se asocian directamente con corrupción de memoria, por ejemplo `strcpy` o `memcpy`, como también las que indican si la traza terminó normalmente o con un

fallo.

El análisis de robustez de variables supone todo un desafío: a pesar de que la interpretabilidad es una propiedad que buscamos, es importante notar que no existen técnicas generales para entender por qué los predictores entrenados toman decisiones sobre los datos. Afortunadamente, debido a que *random forest* resultó el mejor predictor identificado en la Sección 5.4.4, podemos extraer fácilmente la información sobre la relevancia de cada variable dinámica utilizando una sencilla técnica descripta por Breiman [8].

Sin pérdida de generalidad, basamos nuestro análisis en uno de los predictores más simples que fueron entrenados: un *random forest* que utiliza los datos procesados con bolsa de palabras (1-gramas). Se seleccionó dicho predictor debido a que este alcanzó una efectividad razonable (32 %). Las variables más significativas para la predicción de este clasificador se muestran en la Tabla 5.4a. Aquellas que son relevantes al PDV utilizado se encuentran en negrita. En particular, podemos observar que estas variables son ampliamente utilizadas en la predicción. A pesar de esto, el clasificador distribuye la importancia entre las variables analizadas, en vez de concentrarlas en unas pocas. Es muy importante averiguar si las variables relevantes al PDV son las responsables directas de la precisión en la predicción. De ser así, el clasificador sólo se limita a utilizar información obvia y su uso no revela información importante.

Para esto utilizamos un sencillo procedimiento basado en el reentrenamiento del clasificador elegido. Para evitar que el clasificador utilice las variables relevantes a la vulnerabilidad buscada, quitamos del conjunto de datos dichas variables. Los resultados son interesantes: el clasificador reentrenado resulta sólo marginalmente más impreciso (35 % de error). Las variables más significativas para la predicción de este clasificador se muestran en la Tabla 5.4b.

Utilizando este simple procedimiento mostramos que el clasificador resulta robusto en el sentido que no depende de un conjunto definido de variables para mantener su precisión. Hipotetizamos que el clasificador utiliza a su favor la generalidad de las variables para identificar patro-

nes en el comportamiento de los programas. Utilizando estos patrones, el clasificador logra predecir razonablemente bien qué casos de prueba resultarían vulnerables en vez de utilizar solamente las variables directamente relacionadas con la vulnerabilidad buscada.

5.6. Limitaciones

La extracción de información por medio de técnicas livianas de análisis de programas posee varias limitaciones: un error en la predicción del 31 % al usar *VDiscover* indica que existen aspectos para mejorar. Para empezar, las matrices de confusión de las Tablas 5.3a y 5.3b presentan visibles desbalances entre la precisión en la clasificación de casos de prueba vulnerables y no vulnerables. Creemos que ese efecto se produce debido al relativamente pequeño número de casos vulnerables (~ 100) que posee la herramienta para su entrenamiento.

El uso de los distintos tipos de variables tiene sus limitaciones propias. Por ejemplo, las variables estáticas no pueden ser usadas para analizar casos de prueba del mismo programa, ya que se extraen directamente del ejecutable sin considerar qué entrada utilice. Esta limitación no afectó nuestros experimentos ya que todos los casos de prueba utilizaban programas únicos, pero ciertamente sería un problema si *VDiscover* se utiliza para buscan vulnerabilidades en programas con varios módulos. Algunos de estos módulos podrían exhibir patrones asociados con vulnerabilidades y otros no, dificultando notablemente el entrenamiento y uso de la herramienta. Es por eso que las variables estáticas tal como se definieron pueden ser consideradas naturalmente más imprecisas que las dinámicas debido a que todos los programas no triviales poseen un gran número de comportamientos distintos y este tipo de variables no captura esa información.

El uso de variables dinámicas también posee limitaciones: en general es difícil aprender de la información secuencial de las trazas debido a que poseen longitudes variables y pueden contener distintas cantidades de información útil. Por ejemplo, los programas complejos utilizan librerías de

código para realizar tareas específicas: estas poseen distintos patrones de eventos intrínsecos y las trazas pueden presentar dichos patrones intercalados, haciendo que la predicción de casos de prueba vulnerables sea una tarea muy compleja.

Finalmente, otro aspecto importante que limita el uso de los datos es la gran dificultad para combinar variables estáticas y dinámicas para mejorar la precisión de la herramienta: es lógico pensar de que *VDiscover* podría mejorar así su efectividad. No obstante, los resultados de esta estrategia no son los esperados. Los errores de predicción se acercan más a los obtenidos usando sólo variables estáticas. Este fenómeno puede explicarse debido a que las variables estáticas se extraen por cada programa ejecutable, mientras que las dinámicas se extraen por cada ejecución distinta. La combinación de ambas requiere la replicación de las información estática en cada traza. Es posible que durante la etapa de entrenamiento, la baja diversidad de las variables estáticas esté afectando negativamente a los predictores debido a que los modelos de aprendizaje automatizado usualmente asumen independencia entre los datos que reciben y ante la presencia de esta replicación artificial de las variables estáticas, las utilizan con más prioridad durante su entrenamiento. Por ende, la predicción resulta poco efectiva al utilizar las variables combinadas. No encontramos ninguna forma efectiva de combinar distintos tipos de variables para mejorar la precisión en la predicción.

5.7. Comparativa

Al momento de la elaboración de este trabajo, no se ha podido encontrar ninguna técnica para descubrir vulnerabilidades a gran escala sin utilizar código fuente que sea adecuada para la comparación con *VDiscover*. No obstante, sí pudimos identificar una herramienta preexistente que produce una rápida evaluación de un fallo de programa para identificar si oculta una vulnerabilidad: *!Exploitable*. Esta herramienta originalmente desarrollada por *Microsoft* [56] y luego adaptada para funcionar en *Linux* por el CERT [42], realiza un análisis general y rápido del fallo para

estimar si puede ser aprovechado por un atacante.

A diferencia de nuestro enfoque, *!Exploitable requiere* un fallo para analizar el estado final junto con las instrucciones que provocan el fallo. Esta herramienta produce una categoría de explotabilidad de acuerdo a heurísticas internas que posee codificadas por reglas elaboradas por expertos. Las posibles categorías son: *explotable, probablemente explotable, probablemente no explotable* y *desconocido*. Por ejemplo, la regla `isFloatingPointException` que se muestra a continuación:

$$\text{if signal} == \texttt{SIGFPE} \Rightarrow \texttt{PROBABLY\_NOT\_EXPLOITABLE}$$

identifica la señal `SIGFPE` con un fallo *probablemente no explotable*. Esto es debido a que no se conocen técnicas para crear *exploits* a partir de fallos inducidos por excepciones aritméticas de punto flotante.

Luego de ejecutar todos los casos de prueba de nuestro conjunto de datos utilizando *!Exploitable*, el resultado se puede observar en la Tabla 5.5. Para empezar, es importante destacar que esta herramienta posee ventajas frente a nuestro enfoque ya que la misma analiza fallos sin extraer ninguna traza, por lo que se ejecutan a velocidad nativa y no requiere una etapa de entrenamiento.

Para poder realizar una comparación entre *!Exploitable* y *VDiscover* es necesario uniformizar los resultados que ambas herramientas producen. Para esto consideramos que *!Exploitable* identifica casos de prueba vulnerables si estos resultan *explotables* o *probablemente explotables.* Con este criterio, podemos calcular el error de predicción de *!Exploitable* al clasificar los casos de prueba. El error en la clasificación de esta herramienta resulta del 44 %, mientras que el error de prueba de *VDiscover* resulta 31 %. La Tabla 5.3 recopila todos los resultados. Estos experimentos utilizando nuestro conjunto de datos muestran que la efectividad de *!Exploitable* en la predicción es cercana a una elección al azar, sin tener en cuenta el desbalanceo (o sea, un error de clasificación del 50 %) y por lo tanto no resulta útil en la búsqueda de vulnerabilidades. Es posible que para obtener mejores resultados usando *!Exploitable*, sea necesario

Variable	Importancia
fflush:0=Ptr32	6 %
StackCorruption	6 %
MemoryCorruption	4 %
malloc:0=Num32B24	4 %
fread:1=Num32B8	3 %
memset:0=GPtr32	3 %
memset:1=Num32B0	2 %
strcat:1=SPtr32	2 %
strcat:1=GPtr32	2 %
exit:0=Num32B32	2 %
strncpy:0=SPtr32	2 %
strrchr:0=SPtr32	2 %

(a) Utilizando variables relevantes.

Variable	Importancia
strrchr:1=Num32B8	11 %
printf:0=GPtr32	9 %
_IO_getc:0=Ptr32	4 %
malloc:0=Num32B32	3 %
getenv:0=GPtr32	3 %
strcasecmp:1=GPtr32	3 %
open:1=Num32B8	3 %
fprintf:0=Ptr32	3 %
Timeout	2 %
strcasecmp:0=SPtr32	2 %
fopen:0=SPtr32	1 %
malloc:0=Num32B16	1 %

(b) Sin utilizar variables relevantes.

Tabla 5.4: Comparativa de la importancia de variables con respecto al uso de variables relevantes a las vulnerabilidades de corrupción de memoria.

	Vulnerables	**No Vulnerables**
Explotable	**14** %	5 %
Probablemente Explotable	**21** %	18 %
Probablemente No Explotable	43 %	**59** %
Desconocido	22 %	**18** %

Tabla 5.5: Clasificación de los casos de prueba de *VDiscovery* usando *!Exploitable*.

revisar y adaptar las reglas internas que posee para tener en cuenta la amplia variedad de casos de pruebas y fallos posibles. Este procedimiento es manual y requiere un considerable tiempo, incluso para un experto en seguridad informática.

De todas maneras, es importante aclarar que esta comparativa está limitada al descubrimiento de vulnerabilidades de corrupción de memoria, con la muestra de los casos de prueba de *VDiscovery*, por lo que de ninguna manera debe descartarse el uso de *!Exploitable* para otros tipos de vulnerabilidades o casos de prueba específicos.

5.8. Implementación

VDiscover se implementó en *Python* utilizando el paquete `python-ptrace` [84] y GNU *Binutils*. Por un lado, la extracción de variables estáticas se realizó desensamblando el binario con GNU *Binutils*. Por otro lado, la extracción de variables dinámicas resulta un procedimiento un poco más complejo. Se realiza usando puntos de interrupción para engancharse dinámicamente a las funciones del programa analizado mediante `ptrace`. En este caso, existían tres posibles localizaciones para localizar los puntos de interrupción de manera de detener la ejecución cuando una función f se ejecute:

- Al iniciar la llamada a f en el código del programa.
- En la tabla de enlace de procedimientos de f.
- Al iniciar el código de la función f.

Decidimos implementar *VDiscover* utilizando la tabla de enlaces de procedimientos para detectar llamadas a función de la librería estándar de C. Este enfoque apuntaba a ser eficiente y flexible pues no requería escudriñar todo el código ensamblador del programa analizado y proveía la posibilidad de utilizarse en ciertos módulos e ignorar otros.

Es importante destacar que *VDiscover* está implementado para programas ejecutables ELF (*Linux*) que se ejecutan en las plataformas `x86` y `x86_64`. A pesar de que la implementación actual está restringida a dichas plataformas, puede ser ampliado para funcionar en otros sistemas

operativos sin soporte para librerías como `ptrace`, por ejemplo *Windows* o *OSX*, si existe la posibilidad de insertar puntos de interrupción e inspeccionar la memoria de un proceso en ejecución.

Finalmente, se destaca que *VDiscover* es software libre (GPL3) y está disponible gratuitamente en su sitio web oficial.

Eficiencia en la extracción de datos. Nuestra herramienta fue diseñada para evitar el uso de operaciones extremadamente lentas tales como la ejecución instrucción por instrucción. Incluso programas pequeños como `/bin/echo` ejecutan millones de instrucciones por segundo en las computadoras actuales. Es por eso que dicho análisis causa un tiempo de ejecución de 50.000 veces más lento si se intenta desensamblar cada una de las instrucciones ejecutadas.

En el caso de las variables dinámicas, tal como se mencionó en la Sección 5.3.3.2, su eficiencia se define en función del tiempo de ejecución del caso de prueba. Debido a que la extracción de variables dinámicas se realiza utilizando `ptrace`, las ejecuciones instrumentadas resultan 7 veces más lentas. Esta penalidad resulta aceptable para la ejecución de programas complejos.

En el caso de las variables estáticas, tal como se mencionó en la Sección 5.3.3.1, su eficiencia se define en función del tamaño del código desensamblado. El proceso de extracción de variables estáticas no requiere ningún cálculo de estado interno, debido a que el grafo de flujo de control es recorrido aleatoriamente para recolectar secuencias de llamadas a función. No es realmente necesario reconstruir el grafo de flujo de control, sino que simplemente es posible extraer las variables de las instrucciones desensambladas linealmente. Para realizar esta tarea utilizamos GNU *Binutils*, un paquete de software libre para el análisis de programas ejecutables que está muy optimizado para el desensamblado de programas. El proceso completo no toma más de medio minuto en una computadora de escritorio moderna.

Capítulo 6

Conclusiones y Trabajos Futuros

En esta tesis, se plantearon nuevos enfoques para la detección de fallos y vulnerabilidades de software utilizando una variedad de técnicas. En general, podemos concluir que, a pesar del gran avance ocurrido en los últimos años para mejorar la ejecución simbólica o el análisis estático, los enfoques más sencillos, como los *fuzzers*, todavía resultan efectivos en el descubrimiento de fallos y vulnerabilidades en software complejo. Además, podemos afirmar que la detección de vulnerabilidades y fallos es todavía un campo en amplio desarrollo. Este punto se refuerza debido a que las mejoras y extensiones hacia técnicas tradicionales de *fuzzing* planteadas en el Capítulo 4 todavía resultan útiles. También resulta ciertamente posible utilizar aprendizaje automatizado para mejorar la efectividad de las herramientas existentes con un enfoque predictivo, tal como se muestra en el Capítulo 5.

En particular, para cada una de las herramientas presentadas que funcionan para la detección de fallos y vulnerabilidades, se exponen conclusiones específicas y posibles trabajos futuros:

QuickFuzz. Este *fuzzer* generacional utiliza código de terceros para generar y mutar un amplio espectro de formatos de archivos. *QuickFuzz* utiliza los tipos de datos y el código de paquetes de terceros como especificaciones incompletas de distintos formatos de archivos complejos. Tal como se presentó en la Sección 4.2.5, esta herramienta ha resultado efectiva en la búsqueda de nuevos fallos y vulnerabilidades. Adicionalmente, los experimentos realizados en la Sección 4.2.4 indican que los métodos de *fuzzing* que utiliza están altamente optimizados para la generación y mutación

de distintos tipos de archivos.

Los trabajos futuros apuntan a extraer aún más información útil sobre los tipos de datos que indican cómo construir archivos de distintos formatos. Una prometedora línea de investigación busca introducir diferentes mutaciones estructurales utilizando la información de alto nivel provista por los tipos de *Haskell*. Actualmente, las mutaciones sólo son realizadas por los *fuzzers* en los bits o bytes del archivo ya generado. Sin embargo, sería interesante utilizar la información provista por los tipos correspondientes para la definición de una nueva clase, que podríamos denominar `Mutation`. Dicha clase se podría utilizar para realizar mutaciones especializadas en valores previamente obtenidos. El objetivo propondría combinar más efectivamente los enfoques de los *fuzzers* mutacionales y generacionales.

Otra mejora importante que podría investigarse consiste en la posibilidad de incorporar la minimización automática del tamaño de los archivos que producen fallos en los programas. Esto sería posible gracias a la variada información que brindan los tipos y a alguna noción adecuada de tamaño.

Finalmente, se podría desarrollar una extensión para producir tipos de archivos que requieran valores monádicos. Las mónadas, en lenguajes fuertemente tipados y funcionales como *Haskell*, nos permiten encapsular efectos, de manera que sea posible mantener la transparencia referencial. Utilizando tipos de datos monádicos, los programadores pueden especificar tipos de datos que construyan computaciones con estados internos, tales como sesiones de protocolos o conexiones de red. Utilizando una forma de generación de valores arbitrarios adecuada, *QuickFuzz* podría extenderse para ser capaz de realizar pruebas de protocolos y, de esta manera, descubrir errores en software de redes.

XCraft. Esta herramienta para la generación automática de *exploits* aplica un enfoque caja negra para lograr la construcción de los mismos de una manera rápida, incluso sin tener información precisa sobre el comportamiento interno de los programas. Dicho enfoque, se basa en suponer lo que los programas parecen estar haciendo para identificar vulnerabilidades y construir *exploits* para demostrar su gravedad. Tal como se presentó en la Sección 4.3.5, esta herramienta ha resultado efectiva en la búsqueda de nuevos fallos y vulnerabilidades en programas del sistema operativo *Debian*.

Respecto a los trabajos futuros, se plantea la posibilidad de extender la integración de *XCraft* con otros tipos de *fuzzers* de manera que sea posible descubrir nuevos tipos de fallos. Debido a que la herramienta fue desarrollada para ser modular y extensible, éste sería un objetivo a corto plazo.

Además, es interesante considerar extender el soporte para la generación automática de nuevos tipos de *exploits*. Esto podría realizarse utilizando técnicas adicionales para generar *exploits*, por ejemplo: los ataques de formatos de texto [89] o los fallos causados por *use-after-free* [62].

Un objetivo a más largo plazo consistiría en integrar herramientas que usen enfoques de caja blanca, como el análisis simbólico, para poder utilizarse en los casos que no se pueda generar el *exploit*. De esta manera, *XCraft* podría alternar entre el análisis de caja negra que actualmente emplea (rápido, pero menos preciso) y la ejecución simbólica (lenta y costosa, pero muy precisa), de manera tal que genere una gran variedad de *exploits*, aún en programas complejos.

VDiscover. *VDiscover* es una de las herramientas centrales presentadas en esta tesis para la predicción de vulnerabilidades en casos de prueba a gran escala. La misma utiliza técnicas de aprendizaje automatizado básicas, tales como fueron definidas en el Capítulo 2. Los experimentos realizados indican que es posible llevar a cabo predicciones basadas en un gran número de casos de prueba, utilizando información rápidamente extraída del código ensamblador de los programas a analizar (en forma de variables estáticas) y sus ejecuciones (en forma de variables dinámicas). A pesar de que los datos recolectados resultan ruidosos y desbalanceados, el mejor predictor encontrado alcanzó el error de prueba del 31 %. El análisis de los resultados en profundidad indica que el uso del mejor predictor podría ser factible en dos escenarios específicos: (1) la detección rápida de una vulnerabilidad realizando un esfuerzo mínimo y (2) la detección de casi todas las vulnerabilidades presentes en un gran conjunto de casos de prueba, minimizando los costos computacionales. En dichos escenarios, se cuantificaron las mejoras de su uso. El predictor más preciso resultó de 2 a 3 veces más eficiente, respecto a la selección aleatoria. Si bien estas mediciones apuntan a que la técnica tiene potencial en la detección de patrones asociados con vulnerabilidades de software, también indica que las líneas de investigación se deben enfocar en mejorar su precisión para permitir su uso en la industria.

Además, existen varias direcciones que nos interesarían explorar en el futuro. Para empezar, debido a que la disponibilidad de los datos resulta fundamental en las aplicaciones del aprendizaje automatizado, la recolección de los mismos son siempre un punto a mejorar. En el caso de nuestra herramienta, la información recolectada de los programas y los casos de prueba tuvo una efectividad diversa. Las variables dinámicas asociadas con las trazas de las ejecuciones de los casos de prueba resultaron mucho más informativas para la predicción que la información obtenida de los ejecutables desensamblados. Es por este motivo que algunos trabajos futuros se centrarán en la mejora de las técnicas para extraer información que mejore la clasificación. Adicionalmente, los resultados obtenidos nos impulsan a trabajar en desarrollos futuros para recopilar datos más variados y abundantes para el aprendizaje de comportamientos asociados con distintos tipos de vulnerabilidades. El acceso a grandes bases de datos con programas y casos de prueba es crítico para que la herramienta pueda realizar mejores predicciones.

Otra posible línea de investigación para desarrollar consta de la búsqueda de una metodología para combinar efectivamente las variables estáticas y dinámicas, de manera que mejoren la predicción de casos de prueba vulnerables. Hasta ahora, este enfoque ha probado ser elusivo debido a que los clasificadores utilizados se confunden al recibir información tan distinta y esto resulta en predicciones con un porcentaje más alto de error.

Respecto a las técnicas de aprendizaje automatizado utilizadas, esperamos que la inclusión de redes neuronales artificiales más modernas mejoren significativamente nuestros resultados. Por un lado, esperamos poder mejorar la detección de patrones utilizando redes convolucionales (o CNN por sus siglas en inglés) [4]. Estas redes pueden reconocer patrones espaciales en secuencias, reduciendo dramáticamente el número de parámetros necesarios, y han sido utilizadas con éxito en el reconocimiento de objetos en imágenes a gran escala [47] o la detección de tumores en imágenes médicas [23].

Por otro lado, las redes *long short-term memory* (usualmente llamadas LSTM por sus siglas en inglés) [29] poseen una sorprendente capacidad para detectar y relacionar patrones presentes en secuencias de datos. Estas redes han sido utilizadas en tareas diversas, como la generación de texto a partir de imágenes [44] y el reconocimiento de piezas musicales [6]. Las redes LSTM podrían ser adecuadas para la

detección focalizada de eventos de programas que resulten sospechosos. Por lo tanto, permitirían detectar indicios de vulnerabilidades potenciales en programas antes de que ocurran.

Bibliografía

[1] *Combined Static and Dynamic Analysis*. Electronic Notes in Theoretical Computer Science, 131:3–14, 2005. Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL-05).

[2] Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson, Gianluca Bontempi: *Credit Card Fraud Detection*. `https://www.kaggle.com/dalpozz/creditcardfraud`, 2015.

[3] Avgerinos, Thanassis, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo y David Brumley: *Automatic Exploit Generation*. Communications of the ACM, 2014.

[4] Bishop, Christopher M *y cols.*: *Pattern recognition and machine learning*, volumen 1. Springer, 2006.

[5] Böhme, Marcel, Van Thuan Pham y Abhik Roychoudhury: *Coverage-based Greybox Fuzzing as Markov Chain*. En *Proceedings of the Conference on Computer and Communications Security*, páginas 1032–1043. ACM, 2016.

[6] Boulanger-Lewandowski, Nicolas, Yoshua Bengio y Pascal Vincent: *Audio Chord Recognition with Recurrent Neural Networks*. En *Proceedings of ISMIR 14*, 2013.

[7] Braverman, Matthew: *Win32/Blaster: a case study from Microsoft's perspective*. 2003.

[8] Breiman, Leo: *Random forests*. Journal of Machine Learning, 2001.

[9] CACA Labs: *zzuf - multi-purpose fuzzer*. `http://caca.zoy.org/wiki/zzuf`, 2010.

[10] Carnegie Mellon University: *Binary Analysis Platform*. `https://github.com/BinaryAnalysisPlatform/bap`, 2015.

[11] Caruana, Rich, Nikos Karampatziakis y Ainur Yessenalina: *An Empirical Evaluation of Supervised Learning in High Dimensions*. En *Proceedings of the 25th International Conference on Machine Learning*, ICML 08, páginas 96–103. ACM, 2008.

[12] Céspedes, Juan: *ltrace*. `http://www.ltrace.org`, 2014.

[13] Cha, Sang Kil, Thanassis Avgerinos, Alexandre Rebert y David Brumley: *Unleashing Mayhem on Binary Code*. En *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, 2012.

[14] Claessen, Koen y John Hughes: *QuickCheck: a lightweight tool for random testing of Haskell programs*. SIGPLAN Notices, 46(4):53–64, 2011.

[15] Computer Security Lab at UC Santa Barbara: *Rex: Shellphish's automated exploitation engine, originally created for the Cyber Grand Challenge*. `https://github.com/shellphish/rex`, 2016.

[16] Cousot, Patrick, Radhia Cousot, Jérôme Feret, Laurent Mauborgne *y cols.*: *The ASTREÉ Analyzer*. Lecture Notes in Computer Science. Springer, 2005.

[17] Cuoq, Pascal, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto *y cols.*: *Frama-C - A Software Analysis Perspective.* Lecture Notes in Computer Science. Springer, 2012.

[18] DARPA: *Cyber Grand Challenge*. `https://www.cybergrandchallenge.com/`, 2016.

[19] Deerwester, Scott, Susan T. Dumais, George W. Furnas, Thomas K. Landauer y Richard Harshman: *Indexing by latent semantic analysis*. Journal of the American Society for Information Science, 41(6), 1990.

[20] Deja vu Security: *Peach: a smartfuzzer capable of performing both generation and mutation based fuzzing*. `http://peachfuzzer.com/`, 2007.

[21] Deja vu Security: *Peach Pits and Pit Packs*. `http://www.peachfuzzer.com/products/peach-pits/`, 2016.

[22] Drucker, Harris, S Wu y Vladimir N Vapnik: *Support vector machines for spam categorization*. IEEE Transactions on Neural Networks, 10(5), 1999.

[23] Esteva, Andre, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau y Sebastian Thrun: *Dermatologist-level classification of skin cancer with deep neural networks*. Nature, 542(7639):115–118, 2017.

[24] Evans, David y David Larochelle: *Improving Security Using Extensible Lightweight Static Analysis*. Journal on IEEE Software, 2002.

[25] FastText: *FastText: library for fast text representation and classification*. `https://github.com/facebookresearch/fastText`, 2016.

[26] Franco Contanstini: *language-python bug report: some stuff missing in Pretty instances*. `https://github.com/bjpop/language-python/issues/30`, 2010.

[27] Ganesh, Vijay, Tim Leek y Martin Rinard: *Taint-based Directed Whitebox Fuzzing*. En *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09. IEEE Computer Society, 2009.

[28] Genkin, Alexander, David D Lewis y David Madigan: *Large-scale Bayesian logistic regression for text categorization*. Technometrics, 49(3), 2007.

[29] Gers, Felix A., JÃŒrgen Schmidhuber y Fred Cummins: *Learning to Forget: Continual Prediction with LSTM*. Neural Computation, 12:2451–2471, 1999.

[30] giflib: *The GIFLIB project*. `http://giflib.sourceforge.net/`, 1989.

[31] Godefroid, Patrice, Michael Y. Levin y David A. Molnar: *SAGE: whitebox fuzzing for security testing*. Communications of the ACM, 2012.

[32] Goodfellow, Ian J., David Warde-Farley, Pascal Lamblin, Vincent Dumoulin *y cols.*: *Pylearn2: a machine learning research library*. 2013.

[33] Google: *honggfuzz: a general-purpose, easy-to-use fuzzer with interesting analysis options*. `https://github.com/aoh/radamsa`, 2010.

[34] Hackage: *The Haskell community's central package archive of open source software*. `http://hackage.haskell.org/`, 2010.

[35] Harrison, D. y D. L. Rubinfeld: *Hedonic prices and the demand for clean air*. J Environ Economics & Management, 5:81–102, 1978.

[36] He, Haibo y Edwardo A Garcia: *Learning from imbalanced data*. IEEE Transactions on Knowledge and Data Engineering, 21(9), 2009.

[37] Hinton, Geoffrey E y Ruslan R Salakhutdinov: *Reducing the dimensionality of data with neural networks*. Science, 313(5786), 2006.

[38] Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever y Ruslan R Salakhutdinov: *Improving neural networks by preventing co-adaptation of feature detectors*. 2012.

[39] Huang, Shih Kun, Min Hsiang Huang, Po Yen Huang, Han Lin Lu y Chung Wei Lai: *Software Crash Analysis for Automatic Exploit Generation on Binary Programs*. IEEE Transactions on Reliability, March 2014.

[40] Intel: *Intel 64 and IA-32 Architectures Software Developerś Manual*. `https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf`, 2012.

[41] Japkowicz, Nathalie: *The Class Imbalance Problem: Significance and Strategies*. En *Proceedings of the International Conference on Artificial Intelligence (ICAI)*, páginas 111–117, 2000.

[42] Jonathan Foote: *CERT Triage Tools*. `http://www.cert.org/vulnerability-analysis/tools/triage.cfm`, 2013.

[43] Joulin, Armand, Edouard Grave, Piotr Bojanowski y Tomas Mikolov: *Bag of tricks for efficient text classification*. arXiv preprint arXiv:1607.01759, 2016.

[44] Karpathy, Andrej y Fei-Fei Li: *Deep Visual-Semantic Alignments for Generating Image Descriptions.* CoRR, abs/1412.2306, 2014.

[45] Ken Lang: *The 20 Newsgroups data set.* `http://qwone.com/~jason/20Newsgroups/`, 1995.

[46] King, James: *Symbolic execution and program testing.* Communications of the ACM, 19:386–394, 1976.

[47] Krizhevsky, Alex, Ilya Sutskever y Geoffrey E Hinton: *Imagenet classification with deep convolutional neural networks.* En *Advances in neural information processing systems*, 2012.

[48] Lee, JongHyup, Thanassis Avgerinos y David Brumley: *TIE: Principled Reverse Engineering of Types in Binary Programs.* En *NDSS*, 2011.

[49] Lee, Wenke, Salvatore J. Stolfo y Kui W. Mok: *Mining audit data to build intrusion detection models.* En *Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM Publisher, 1998.

[50] Marlow, Simon: *Haskell 2010 Language Report*, 2010.

[51] Martin, Guy, James Kinross y Chris Hankin: *Effective cybersecurity is fundamental to patient safety.* BMJ, 2017, ISSN 0959-8138.

[52] Martín Ceresa and Pablo Buiras: *MEga DErivation with Template Haskell.* `https://github.com/CIFASIS/megadeth`, 2016.

[53] Michal Zalewski: *American Fuzzy Lop: a security-oriented fuzzer.* `http://lcamtuf.coredump.cx/afl/`, 2010.

[54] Michal Zalewski: *Technical "whitepaper" for afl-fuzz.* `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2010.

[55] Microsoft Corporation: *Microsoft Security Development Lifecycle.* `MicrosoftSecurityDevelopmentLifecycle`, 2012.

[56] Microsoft Security Engineering Center (MSEC) Security Science Team: *!Exploitable*. `http://msecdbg.codeplex.com`, 2013.

[57] Mikolov, Tomas, Kai Chen, Greg Corrado y Jeffrey Dean: *Efficient estimation of word representations in vector space*. 2013.

[58] Mikolov, Tomas, Kai Chen, Greg Corrado y Jeffrey Dean: *Efficient Estimation of Word Representations in Vector Space*. CoRR, abs/1301.3781, 2013.

[59] Miller, Barton P., Louis Fredriksen y Bryan So: *An Empirical Study of the Reliability of UNIX Utilities*. Communications of the ACM, 33(12):32–44, 1990, ISSN 0001-0782.

[60] Miller, Charlie y Zachary NJ Peterson: *Analysis of mutation and generation-based fuzzing*. Independent Security Evaluators, Tech. Rep, 2007.

[61] Mitchell, Thomas M.: *Machine Learning*. McGraw-Hill, Inc., 1ª edición, 1997, ISBN 0070428077, 9780070428072.

[62] MITRE: *CWE-416: Use After Free*. `https://cwe.mitre.org/data/definitions/416.html`, 2000.

[63] MITRE: *Common Weakness Enumeration*. `https://cwe.mitre.org/`, 2006.

[64] Mozilla: *Dharma: a generation-based, context-free grammar fuzzer*. `https://github.com/MozillaSecurity/dharma`, 2015.

[65] Nair, Vinod y Geoffrey E. Hinton: *Rectified Linear Units Improve Restricted Boltzmann Machines*. En *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, páginas 807–814, 2010.

[66] Nethercote, Nicholas y Julian Seward: *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. SIGPLAN Notices, 42(6):89–100, 2007.

[67] One, Aleph: *Smashing the stack for fun and profit*. Phrack Magazine, 49(14), 1998.

[68] Oulu University Secure Programming Group: *A Crash Course to Radamsa*. `https://code.google.com/p/ouspg/wiki/Radamsa`, 2010.

[69] OUSPG: *rad/mutations.scm*. `https://github.com/aoh/radamsa/blob/master/rad/mutations.scm`, 2016.

[70] Pedram Amini and Aaron Portnoy: *sulley: a pure-python fully automated and unattended fuzzing framework*. `https://github.com/OpenRCE/sulley`, 2012.

[71] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel *y cols.*: *Scikit-learn: Machine Learning in Python*. Journal of Machine Learning Research, 12, 2011.

[72] Pirzadeh, H., A. Hamou-Lhadj y M. Shah: *Exploiting text mining techniques in the analysis of execution traces*. En *IEEE International Conference on Software Maintenance (ICSM)*, 2011.

[73] Pyysalo, S., F. Ginter, H. Moen, T. Salakoski y S. Ananiadou: *Distributional Semantics Resources for Biomedical Text Processing*. En *Proceedings of LBM 2013*, páginas 39–44, 2013.

[74] Rawat, Sanjay y Laurent Mounier: *Finding Buffer Overflow Inducing Loops in Binary Executables*. En *Proceedings of Sixth International Conference on Software Security and Reliability (SERE)*. IEEE, 2012.

[75] Robiah, Y., S. S. Rahayu, S. Sahib, M. M. Zaki, M. A. Faizal y R. Marliza: *An improved traditional worm attack pattern*. En *International Symposium on Information Technology*, volumen 2, páginas 1067–1072, 2010.

[76] Roemer, Ryan, Erik Buchanan, Hovav Shacham y Stefan Savage: *Return-Oriented Programming*. ACM Transactions on Information and System Security, (1):1–34, 2012.

[77] Sang Kil Cha: *ppthtml: CVE-2013-4565: heap-based buffer overflow*. `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=729279`, 2013.

[78] Sanguansat, Parinya: *Principal Component Analysis - Multidisciplinary Applications*. InTech, 2012, ISBN 9789535101291.

[79] Santos, Igor, Jaime Devesa, Felix Brezo, Javier Nieves y Pablo Garcia Bringas: *OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware*

Detection. En *International Joint Conference CISIS-ICEUTE-SOCO*, volumen 189 de *Advances in Intelligent Systems and Computing*. 2013.

[80] Serebryany, Konstantin, Derek Bruening, Alexander Potapenko y Dmitry Vyukov: *AddressSanitizer: A Fast Address Sanity Checker*. USENIX ATC'12, páginas 28–28, 2012.

[81] Sheard, Tim y Simon Peyton Jones: *Template Meta-programming for Haskell*. SIGPLAN Notices, 37(12):60–75, 2002, ISSN 0362-1340.

[82] Shlens, Jonathon: *A Tutorial on Principal Component Analysis*. CoRR, 2014.

[83] Shoshitaishvili, Yan, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel y Giovanni Vigna: *SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis*. En *IEEE Symposium on Security and Privacy*, 2016.

[84] Stinner, Victor: *python-ptrace*. `http://python-ptrace.readthedocs.org`, 2014.

[85] Symantec: *Stuxnet 0.5: The Missing Link*. 2013.

[86] Taigman, Yaniv, Ming Yang, Marc'Aurelio Ranzato y Lior Wolf: *Deepface: Closing the gap to human-level performance in face verification*. En *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[87] Team, Mayhem: *Reporting 1.2K crashes*. `https://lists.debian.org/debian-devel/2013/06/msg00720.html`, 2013.

[88] Theo de Raadt: *i386 W^X*. `http://marc.info/?l=openbsd-misc&m=105056000801065`, 2003.

[89] Tim Newsham: *Format string attacks*. `http://forum.ouah.org/FormatString.PDF`, 2000.

[90] Trail of Bits: *Manticore: a prototyping tool for dynamic binary analysis, with support for symbolic execution, taint analysis, and binary instrumentation*. `https://github.com/trailofbits/manticore`, 2017.

[91] Vincent Berthoux: *svg-tree: SVG loader/serializer for Haskell.* `https://github.com/Twinside/svg-tree`, 2007.

[92] Vincent Berthoux: *Juicy.Pixels: Haskell library to load & save pictures.* `https://hackage.haskell.org/package/JuicyPixels`, 2012.

[93] W3C: *Scalable Vector Graphics (SVG) 1.1 (Second Edition).* `https://www.w3.org/TR/SVG11/`, 2011.

[94] Wolf, Lior, Yair Hanani, Kfir Bar y Nachum Dershowitz: *Joint word2vec networks for bilingual semantic representations.* International Journal of Computational Linguistics and Applications, 5(1), 2014.

[95] Xu, Wei, Ling Huang, Armando Fox, David Patterson y Michael I. Jordan: *Detecting Large-scale System Problems by Mining Console Logs.* En *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[96] Yamaguchi, Fabian, Nico Golde, Daniel Arp y Konrad Rieck: *Modeling and Discovering Vulnerabilities with Code Property Graphs.* En *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14. IEEE Computer Society, 2014.

[97] Yann LeCun, Corinna Cortes y Christopher J.C. Burges: *The MNIST database of handwritten digits.* `http://yann.lecun.com/exdb/mnist/`, 1998.

[98] Zeller, Andreas: *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., 2005.

[99] Zeller, Andreas y Ralf Hildebrandt: *Simplifying and Isolating Failure-Inducing Input.* IEEE Transactions on Software Engineering, 28(2):183–200, 2002.

[100] Zhang, Mingwei, Aravind Prakash, Xiaolei Li, Zhenkai Liang y Heng Yin: *Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis.* 2012.