Detección de vulnerabilidades de código en C y C++ mediante Redes Neuronales Recurrentes

Code vulnerability detection for C & C++ by Recurrent Neural Networks

TRABAJO FIN DE GRADO

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y ADMINISTRACIÓN Y DIRECCIÓN DE EMPRESAS

CURSO 2022–2023

José María García Herranz
Sergio Muñoz Martín

Directores

Luis Javier García Villalba
Luis Alberto Martínez Hernández

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Madrid, Septiembre de 2023

Agradecimientos

Queremos expresar nuestro más sincero agradecimiento a todas las personas que contribuyeron en el trabajo de fin de grado. En primer lugar, nos gustaría agradecer a nuestros directores de TFG Luis Javier García Villalba y Luis Alberto Martínez Hernández, por su guía y apoyo incondicional a lo largo de todo el proceso de investigación.

También, no queremos olvidarnos de nuestros, profesores, compañeros y familiares que con su apoyo, han sido muy importantes todo este tiempo, ayudándonos en los momentos mas difíciles del trabajo y permitiendo que podamos conseguir los objetivos propuestos.

Muchas gracias a todas estas personas que han hecho posible llevar a cabo este proyecto de fin de grado.

Índice General

Índice de Figuras

Índice de Tablas

Lista de Códigos

Lista de Acrónimos

AA	*Aprendizaje Automático*
AP	*Aprendizaje Profundo*
AST	*Abstract Syntax Tree*
BLSTM	*Bidirectional Long Short-Term Memory*
CFG	*Control Flow graph*
CPG	*Code Property Graph*
DevOps	*Desarrollo y operaciones*
DevSecOps	*Desarrollo, Seguridad y Operaciones*
DL	*Deep Learning*
FN	*False Negatives*
FP	*False Positives*
GAN	*Redes Generativas Adversiales*
GNN	*Graph Neural Network*
IA	*Inteligencia Artificial*

LSTM	*Long Short Term Memory*
ML	*Machine Learning*
PDG	*Program dependence graph*
RNC	*Redes Neuronales Convolucionales*
RNN	*Redes Neuronales Recurrentes*
TN	*True Negatives*
TP	*True Positives*

Abstract

Nowadays, vulnerability detection has become a critical issue for any organization, due to the fact that in an increasingly connected world, new applications that meet the needs of users are demanded. Static code analysis is a fundamental test that organizations must perform to verify the robustness of their code prior to putting a new product into production, allowing them to minimize security holes and thus protect an organization's data. However, code analysis is currently performed semi-automatically with the help of commercial tools that are highly accurate but at a stage where the code is almost fully developed, which could lead to human error or overlook a vulnerable piece of software.

Artificial intelligence could enable the development of increasingly secure code thanks to its ability to process large amounts of information. Recurrent Neural Networks are a type of deep learning model that can capture sequences of data, making them suitable for analyzing source code, which is structured in sequences of instructions and symbols. In the context of vulnerability detection, RNNs can analyze the context, identify complex patterns, adapt to variations in the programming style of each developer and learn from large data sets, making correct predictions about files with similar vulnerabilities.

This final degree work focuses on the design of a model through a Recurrent Neural Network, specifically through one of LSTM, which will seek to detect vulnerabilities within the code fragments in C/C++.

Keywords: Deep Learning, RNN, LSTM, Vulnerabilities, C/C++, Python, Big-Vul, Vudenc

Resumen

En la actualidad, la detección de vulnerabilidades se ha vuelto un tema crítico para cualquier organización, esto debido a que en un mundo cada vez más conectado se exigen nuevas aplicaciones que satisfagan las necesidades de los usuarios. El análisis de código estático es una prueba fundamental que deben realizar las organizaciones para verificar la robustez de su código previo a poner en producción un nuevo producto, permitiendo minimizar los huecos de seguridad y con esto proteger los datos de una organización. Sin embargo, en análisis de código en la actualidad se realiza de manera semi-automática con la ayuda de herramientas comerciales que destacan por su alta precisión pero en una etapa donde el código se encuentra desarrollado casi en su totalidad, lo que podría ocasionar errores humanos o pasar por alto una pieza de software vulnerable.

La inteligencia artificial podría permitir desarrollar códigos cada vez más seguros gracias a su capacidad de procesar grandes cantidades de información. Las Redes Neuronales Recurrentes son un tipo de modelo de aprendizaje profundo que pueden capturar secuencias de datos, lo que las hace adecuadas para analizar código fuente, que se estructura en secuencias de instrucciones y símbolos. En el contexto de la detección de vulnerabilidades, las RNN pueden, analizar el contexto, identificar patrones complejos, adaptarse a variaciones propias de la forma de programación de cada desarrollador y aprender de grandes conjuntos de datos, consiguiendo realizar predicciones correctamente sobre ficheros con vulnerabilidades similares.

Este trabajo de fin de grado se centra en el diseño de un modelo a través de una Red Neuronal Recurrente, concretamente a través de una de LSTM, con el que se buscará detectar vulnerabilidades dentro de los fragmentos de código en C/C++.

Palabras clave: Aprendizaje Profundo, RNN, LSTM, Vulnerabilidades, C/C++, Python, Big-Vul, Vudenc

Capítulo 1

Introducción

1.1. Motivación

Durante las últimas décadas, gracias a numerosos avances y desarrollos en el mundo de la tecnología actual, la seguridad de los sistemas informáticos se ha vuelto un tema especialmente primordial, sobre todo la seguridad en el código de software.

Estos cambios han provocado que la sociedad sea cada vez más dependiente de las plataformas digitales a medida que éstas siguen en constante evolución, incrementando las posibilidades de que se puedan llevar a cabo tanto ataques maliciosos como propagaciones de nuevas amenazas informáticas. Como consecuencia de estas acciones, se ha vuelto fundamental la protección de la integridad y la confidencialidad de los datos, así como la disponibilidad de los sistemas.

Además, en el área de la ingeniería de software, uno de los temas principales para poder garantizar la seguridad de los sistemas reside en la detección y prevención de vulnerabilidades dentro del código. Actualmente, debido a la complejidad de los sistemas y a la incorporación de nuevas tecnologías, el análisis y detección manual de vulnerabilidades se ha vuelto una tarea cada vez más difícil y propensa a generar errores humanos.

En este sentido, tecnologías como el aprendizaje automático surge como una posible solución para mejorar la eficacia y la eficiencia de las pruebas de seguridad en el desarrollo del software.

Por lo anterior, en el presente trabajo, se realizará un estudio de diferentes técnicas de Inteligencia Artificial que puedan ser útiles para el análisis estático de código en el lenguaje de programación C y C++.

1.2. Contexto

El presente Trabajo Fin de Grado se enmarca dentro de un proyecto de investigación titulado Platform for Analysis of Resilient and Secure Software – LAZARUS, aprobado por la Comisión Europea dentro del Programa Marco Horizonte (convocatoria HORIZON-CL3-2021-CS-01) en virtud del acuerdo de subvención número 101070303 y en el que participa el Grupo GASS de la Universidad Complutense de Madrid (Grupo de Análisis, Seguridad y Sistemas, https://gass.ucm.es, grupo 910623 del catálogo de grupos de investigación reconocidos por la UCM).

Además de la Universidad Complutense de Madrid participan en LAZARUS las siguientes entidades: Athena Research Center – ARC (Grecia), The University of Padua (Italia), Infotrend Innovations Company Limited (Chipre), Data Centric Services SRL (Rumanía), Luxembourg Institute of Science and Technology (Luxemburgo), Motivian EOOD (Bulgaria), Binare Oy (Finlandia), Fundación APWG European Union Foundation (España), Maggioli Spa (Italia).

Tienen más información en:

https://cordis.europa.eu/project/id/101070303

https://lazarus-he.eu

1.3. Objeto de la Investigación

El código, en el contexto de la programación, está formado por un conjunto de instrucciones que el desarrollador ordena ejecutar al computador. Sin embargo, pueden producirse errores, a la hora de programar, que no sean visibles a simple vista, generando lo que se conoce como vulnerabilidades. Éstas se pueden deber a dos motivos: un error en el código de la aplicación desarrollada o un error de codificación en las bibliotecas usadas. Por esta razón, se empezaron a desarrollar modelos con el fin de reducir este tipo de problemas.

Actualmente, existen diferentes modelos que son capaces de encontrar vulnerabilidades dentro del código. Sin embargo, una de sus desventajas es que solamente están focalizados en un tipo de lenguaje, y no solo eso, sino que tampoco son muy precisos, ya que a veces no indican las líneas que son problemáticas o el porcentaje de fiabilidad.

Con todo esto, la funcionalidad de este Trabajo de Fin de Grado consistirá en diseñar e implementar un modelo de *Machine Learning* que sea capaz de realizar la identificación de vulnerabilidades en trozos de código dados en el lenguaje de programación C y C++, con el fin de mejorar la identificación de *bugs* y reducir el tiempo de análisis. Además, este modelo permitirá realizar la revisión de vulnerabilidades en código en un corto periodo de tiempo con el fin de ayudar a los desarrolladores a generar piezas de software que sean cada vez más seguras y a prueba de errores humanos en materia de seguridad.

Para poder llevar esto a cabo, se hará un uso de un marco de pruebas estándar para encontrar los mejores parámetros y métodos de entrenamiento para poder moldear el modelo. Asimismo, este proyecto pretende comprender el funcionamiento de las *Recurrent Neural Networks* (**RNN**, por sus siglas en inglés), específicamente de las *Long Short-Term Memories* (**LSTM**, por sus siglas en inglés), y así conocer su estructura y entrenamiento, además de las dificultades que propone diseñar una desde cero.

1.4. Plan de Trabajo

El trabajo se ha desarrollado en tres fases principales:

1. **Investigación**: Para empezar, se llevó a cabo un periodo de adaptación durante los primeros cuatro meses con el fin de entender el contexto de trabajo y adquirir un nivel de conocimientos necesarios para comenzar con el posterior desarrollo. Lo primero que se hizo en esta fase fue realizar una reunión general en la que se plantearon diferentes puntos, como fueron los objetivos a lograr, una pequeña guía de cómo comenzar el trabajo y el proceso de investigación y cuáles iban a ser los conocimientos necesarios para poder completar el trabajo. Igualmente, se acordó tener reuniones semanales para poder ver el seguimiento del proceso y resolver las dudas que se planteaban. Además, los tutores se dedicaron a explicar diferentes conceptos sobre los campos que concierne este trabajo, de los cuales se hablará en los siguientes apartados, ya que ambos integrantes no teníamos conocimientos sobre el Aprendizaje Automático o sobre la Inteligencia Artificial y sus usos, recomendando distintas herramientas para facilitar la búsqueda de información. Entre ellas cabe destacar *Google Scholar*, ya que sirvió para encontrar todo tipo de artículos científicos centrados en el mismo campo de estudio y así poder utilizarlos para poder entender este tipo de tecnologías. Por último, antes de empezar con el desarrollo de este proyecto se tomaron múltiples conclusiones y opiniones para el primer moldeado y sus requerimientos.

2. **Desarrollo**: Una vez adquiridos estos conocimientos básicos y necesarios para la realización de este trabajo, empezó esta segunda fase donde se dedicó menos a tiempo a investigar y se comenzó a trabajar con esa información recolectada para la codificación de la propuesta. De esta forma, la búsqueda de información solamente se basaba en conceptos surgidos durante esta fase, como pueden ser librerías como *TensorFlow* o *Keras* o conceptos de programación de Python. Igualmente, estas fuentes de información que fueron proporcionadas por otros trabajos científicos han sido la base y guía para poder modelar los conjuntos de entrenamiento.

3. **Experimentación**: En esta última fase se empezaron a desarrollar los prototipos de la idea inicial a través de la aplicación de diferentes herramientas incluidas en las librerías explicadas anteriormente. Además, en esta fase se compararon los resultados obtenidos con otros trabajos y se ajustaron los parámetros para poder configurar el modelo, con el fin de afinarlo para encontrar mejores resultados. Paralelamente se siguió trabajando en el desarrollo del proyecto con la finalidad de buscar la optimización en el modelo.

1.5. Estructura del Trabajo

El resto de la memoria se va organizar en diferentes capítulos. El Capítulo 2 introducirá algunos conceptos claves para poder entender el contexto de la investigación. Entre estos elementos se hablará sobre la seguridad de la información y sobre los tipos de ataques, se explicará la práctica de *DevOps y DevSecOps* en el trabajo y los accesos de caja negra y caja blanca.

El Capítulo 3 estará centrado en el estado del arte, el cual estará enfocado en los lenguajes de programación de C/C++ y Python. Primero se hará una pequeña introducción, seguido de una explicación sobre los conjuntos de datos de entrenamiento más utilizados. Después, se presentará la representación del código, ya bien sea a través de grafos o textos. Por último, se realizará el entrenamiento del modelo que detecte vulnerabilidades en los lenguajes indicados anteriormente, comentando todos los pasos seguidos.

Dentro de las propuestas estudiadas en el estado del arte, el Capítulo 4 servirá para enfocarse en una de ellas, que en este caso será el modelo Vudenc. Además, se explicará cuál es su funcionamiento y cómo se generan y entrenan los dos modelos que lo componen. Igualmente, se realizarán análisis de código utilizando estas herramientas para ver qué resultados producen.

El Capítulo 5 describirá los experimentos realizados para evaluar la efectividad de los algoritmos utilizados en la fase anterior, presentando el conjunto de datos que se ha seleccionado. Asimismo, los resultados que se obtengan serán analizados y comparados con los obtenidos en otros modelos.

El Capítulo 6 servirá para reflejar las conclusiones que se lleguen a desarrollar. Igualmente, se presentarán estudios futuros que puedan ayudar a complementar este trabajo.

La contribución de los miembros del equipo estará reflejada en el Capítulo 7, indicando qué partes y qué función ha realizado cada uno.

Por último, los Capítulos 8 y 9 constituirán la traducción al inglés de la introducción (Capítulo 1) y de las conclusiones (Capítulo 6).

1.6. Diagrama de Gantt

Figura 1.1: *Diagrama de Gantt*

Tareas	Dur	Fecha ini	Fecha fin	Sep	Oct	Nov	Dic	En	Feb	Mar	Abr	May	Jun	Jul	Ago
Cursos Python, ML	4 sem	05/09/22	30/09/22												
DevSecOps	2 sem	01/10/22	15/10/22												
GANS	4 sem	16/10/22	15/11/22												
Aplicaciones ataques y defensas	2 sem	1/11/22	16/11/22												
Estudiar primer modelo	4 sem	16/11/22	15/12/22												
Estado del arte	8 sem	01/12/22	31/01/23												
Busqueda Conjunto de datos	4 sem	01/01/23	31/01/23												
Ejecucion código modelos:															
Njsscan	8 sem	01/02/23	31/03/23												
Vuden	10 sem	16/03/23	31/05/23												
Adaptar Vuden a C/C++	2 sem	01/06/23	15/06/23												
Datos de entrenamiento	4 sem	16/06/23	15/07/23												
Experimentos	6 sem	16/07/23	31/08/23												
Redacción memoria	24 sem	01/03/23	31/08/23												

Capítulo 2

Contexto de la Investigación

2.1. Historia de la Inteligencia Artificial

El campo de la *Inteligencia Artificial* (IA) siempre ha estado en completa evolución. Desde sus inicios por la década de los 60s, con la primera arquitectura de red neuronal, este sector ha comenzado a crecer a velocidades extremas, sobre todo a nivel de hardware, favoreciendo la aparición de nuevas máquinas más potentes capaces de construir y soportar arquitecturas más complejas y eficientes.

Según [Rou18] y [RVPO21], se podría definir la IA como la capacidad que tienen las máquinas para poder utilizar los algoritmos, para aprender de información y datos dados y para poder aplicar el conocimiento aprendido en la toma de decisiones, siguiendo un comportamiento parecido al de los humanos. Igualmente, estas máquinas cuentan con la ventaja de que son dispositivos que no necesitan descansar, lo que les permite analizar simultáneamente grandes volúmenes de información, realizando un menor número de errores que los humanos.

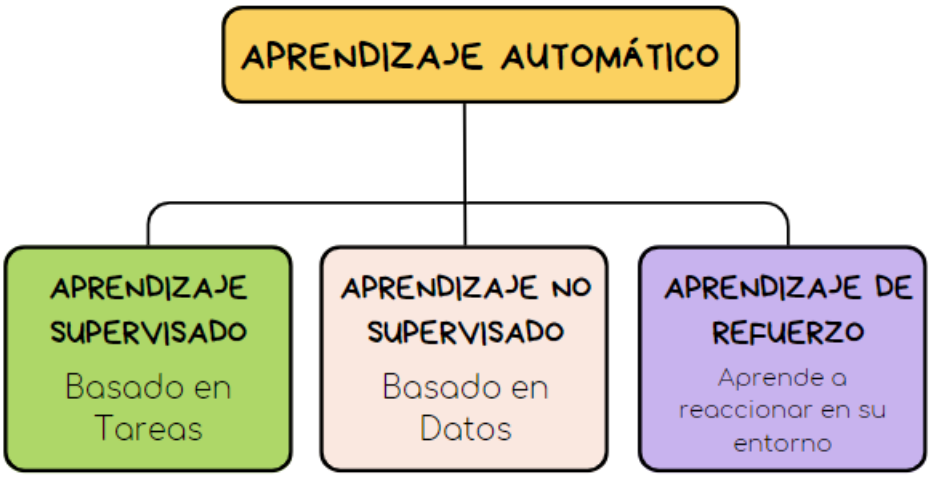
Estas técnicas son tan importantes para detectar vulnerabilidades porque por medio de la IA las máquinas son capaces de identificar relaciones de datos que un ser humano no es capaz, haciendo que durante los últimos años, la gran mayoría de las propuestas de análisis de código centraran su metodología en el uso de la IA.

2.2. Aprendizaje Automático

A raíz de estas evoluciones tecnológicas, empezaron a surgir nuevas aplicaciones con la IA. Uno de los principales avances y que tiene gran peso actualmente en el mundo informático consiste en el *Aprendizaje Automático* (AA) (también conocido como *Machine Learning* (ML) por su traducción al inglés), tratándose de un aspecto que tienen las máquinas que las permite tener la capacidad de aprender y analizar información sin estar programadas para ello, tal y como se puede ver en el motor de búsqueda que utiliza Google.

En cuanto al AA, se puede decir que usa distintos algoritmos para poder aprender de los patrones de datos, existiendo una pequeña clasificación de tres subconjuntos según el tipo de aprendizaje, tal y como se puede apreciar en la Figura 2.1 que aparece a continuación, destacando:

Figura 2.1: Tipos de Aprendizaje Automático



- **Aprendizaje supervisado:** Se utilizan datos que están etiquetados para indicar cómo tendría que ser categorizada esa nueva información.
- **Aprendizaje no supervisado:** Al contrario que el anterior, los datos no son etiquetados, por lo que son ellos los que tienen que encontrar la forma de clasificarse a sí mismos.
- **Aprendizaje de refuerzo:** Para este tipo de algoritmo hay que reforzar positivamente cada vez que el programa tiene un acierto, haciendo que aprenda a lo largo del tiempo en base a la experiencia anterior.

2.3. Aprendizaje Profundo

Por otro lado, también cabe mencionar otro de los avances surgidos de la IA, como puede ser el *Aprendizaje Profundo* (AP) (también conocido como *Deep Learning* (DL) por su traducción al inglés).

Si bien se ha encontrado alguna propuesta más antigua que implementaba modelos de AA [NZHZ07], en el estado del arte todas las propuestas utilizan AP, ya que aunque ambas son técnicas de inteligencia artificial muy similares, las redes de AP son más complejas que las de AA, destacando el papel fundamental que tiene la máquina, que es mayor en el análisis y entreno de los datos.

Además, como ya se ha mencionado anteriormente, permite extraer aquellas relaciones que un ser humano no es capaz de detectar en la gran mayor parte de las ocasiones, mientras que en el AA, durante la fase de entrenamiento, se centra mucho más en el etiquetado de los datos y es la persona la que le indica al modelo que características son relevantes.

Dentro del AP, también se puede hacer una clasificación sobre los tipos de redes, destacando:

- ***Redes Neuronales Convolucionales* (RNC)** [LLY+22]: Son especialmente adecuadas para el procesamiento de imágenes o audio y suelen ser aplicadas para la clasificación de imágenes o sonidos y para el reconocimiento facial. Se basan en la idea de convoluciones para aprender características visuales y detectar patrones como bordes, texturas y formas en imágenes.
- ***Redes Neuronales Recurrentes* (RNN)** [YSHZ19]: Las RNN son apropiadas para datos secuenciales y temporales, como el procesamiento natural del lenguaje y la predicción de series temporales.
- ***Redes Generativas Adversiales* (GAN)** [DGC+20]: Las GAN están formadas por dos redes, un generador y un discriminador. Mediante esta arquitectura las dos redes compiten entre sí para retro alimentarse. Por un lado, el generador genera nuevos datos tratando que sean indistinguibles de los reales, mientras que el discriminador es un clasificador que tiene que distinguir los datos reales de los datos del generador. Las GAN han sido muy populares recientemente en la generación de imágenes y arte, pero también pueden tener un impacto importante en la seguridad generando nuevos ataques que no han sido descubiertos todavía y preparándose para ellos.
- ***Graph Neural Network* (GNN)** [WPC+21]: Las GNN son redes neuronales diseñadas para realizar el análisis de datos sobre datos en forma de grafos y proporcionan una manera fácil de realizar tareas de predicción a nivel de nodo, nivel de borde y nivel de gráfico. Son un gran complemento para las RNC y las RNN, ya que mediante grafos se recogen información de los datos de un modo totalmente diferente .

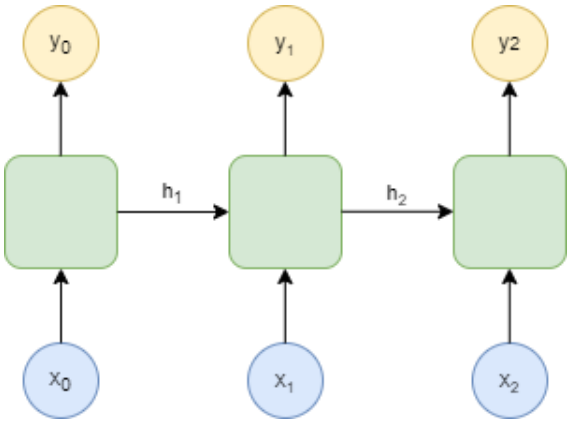
2.3.1. RNN

Las RNN son un tipo de red neuronal enfocada en el procesamiento de datos secuenciales o temporales. Se caracterizan por procesar datos de entrada de manera independiente y pueden mantener una memoria interna, la cual procesa información en función de lo que han visto anteriormente en la secuencia. En [Gra12] se explica como las RNN se centran en entrenamiento de datos en bucle mediante secuencias como se observa en la Figura 2.2. El entrenamiento de los datos se realiza en forma de secuencia mediante varias unidades neuronales que se conectan entre sí, y en las que para calcular la salida y_t de cada dato las neuronas utilizan el dato x_t que toca en el momento de tiempo t y otra variable h_{-t} en la que representa la información o estado de los datos anteriores.

$$h_t = f(h_{t-1}, x_t)$$

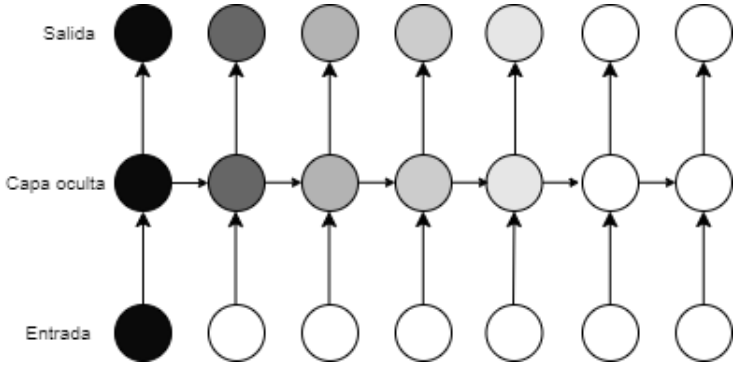
En este proceso de entrenamiento que tenían las RNN más básicas se encuentra un problema con el desvanecimiento del gradiente.

Figura 2.2: *Secuencia RNN*



Mediante este mecanismo las RNN consiguen que según vaya avanzando la secuencia los resultados tengan en cuanta la información de los datos anteriores. Sin embargo, debido a lo que se conoce como el problema del desvanecimiento del gradiente [WZ95], en las secuencias de gran tamaño, como se observa en la Figura 2.3 al calcular el error del gradiente la influencia de los datos iniciales se reduce de manera exponencial, por lo que estas redes solo tienen memoria a corto plazo.

Figura 2.3: *Problema desaparición del gradiente*



2.3.2. LSTM

Las *Long Short Term Memory* (LSTM) [LST20] son un tipo de red neuronal recurrente que solucionan el problema con la memoria a largo plazo que tienen las RNN más básicas.

En la Figura 2.4 se muestra la arquitectura de una sola celda, que conectándose con otras formaría una red similar a la de la Figura 2.2. Esta celda es la principal diferencia de las LSTM y el resto de RNN. Está basada en el control de la información que se puede añadir y eliminar entre tres puertas: olvido, entrada y salida. Este mecanismo de puertas permite que la LSTM regule el flujo de información y maneje mejor secuencias con escalas de tiempo variables y dependencias a largo plazo. Para llevar el control, cada puerta tiene una capa sigma que indica cuánta información debe dejar pasar de cada dato al estado. De esta manera cada puerta tiene la siguiente función y realiza los siguientes cálculos:

- **Puerta de olvido.** Recibe el estado anterior y decide qué información hay que descartar en el siguiente estado de la celda.

$$f_t = \sigma(W_f[h_{t-1}, x_t + b_f] \tag{2.1}$$

- **Puerta de entrada.** Decide cuál es la información que hay que añadir al nuevo estado C_t. En i_t se calcula qué información es la que hay que añadir al estado mediante la función σ. En c'_t mediante la función tangente hiperbólica ($tanh$) se genera un vector de nuevos valores candidatos al nuevo estado. C_t es el nuevo estado resultado de combinar i_t y c'_t.

$$i_t = \sigma(W_i[h_{t-1}, x_t + b_i] \tag{2.2}$$

$$c'_t = tanh(W_c[h_{t-1}, x_t + b_c] \tag{2.3}$$

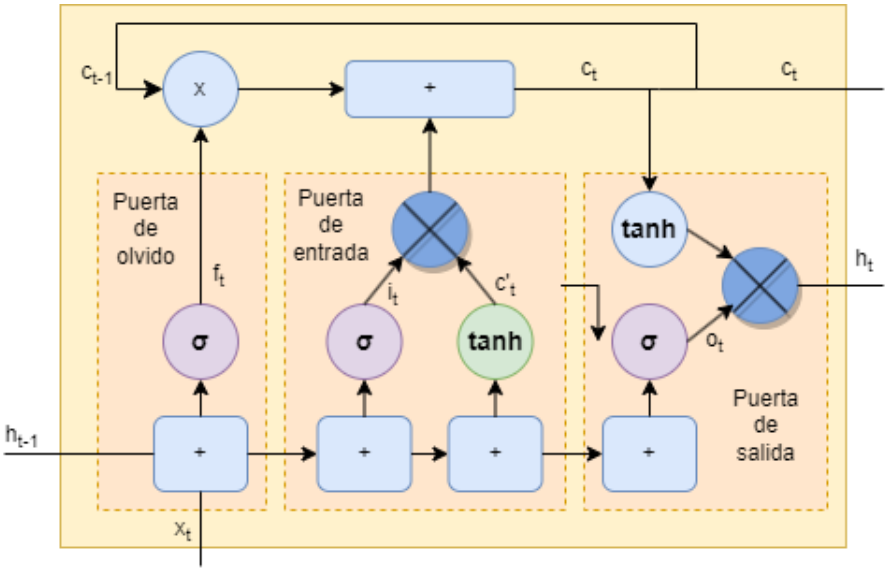
$$C_t = \sigma(f_f[C_{t-1}, i_t + c'_t] \tag{2.4}$$

- **Puerta de salida.** Finalmente calcula la salida h_t de manera similar a la que se realiza en la puerta de entrada.

$$o_t = \sigma(W_o[h_{t-1}, x_t + b_o] \tag{2.5}$$

$$h_t = o_{t-1} * tanh(C_t] \tag{2.6}$$

Figura 2.4: *Arquitectura celda LSTM*



2.4. Seguridad Informática

La ciberseguridad se ha convertido en un área clave de estudio en la era de conectividad digital en la que nos encontramos. Aunque el constante avance tecnológico ha reportado numerosos beneficios, también ha generado un panorama de ciberamenazas cada vez más complejo. La seguridad informática se refiere a una variedad de procedimientos, técnicas y herramientas utilizadas para garantizar la disponibilidad, confidencialidad e integridad de la información digital, así como para defender los sistemas y redes de intrusiones dañinas.

Por ello, este trabajo pretende ofrecer una visión global de la seguridad informática, empezando por una compresión profunda de las ideas centrales del tema antes de pasar a un subcapítulo concreto que se centra en la seguridad del código, la cual es esencial para mantener sistemas fiables y resistentes en un mundo en el que la digitalización se basa en ella.

A diferencia de las repeticiones habituales, la primera sección de este subcapítulo tratará los fundamentos de la seguridad informática y su relación con los procesos sociales, tecnológicos y económicos en evolución. Un conocimiento profundo de las medidas de seguridad es esencial para combatir los riesgos cambiantes a medida que la tecnología se desarrolla a un ritmo vertiginoso.

Por otro lado, la segunda parte se centrará en la seguridad del código, crucial en un momento en que muchos sistemas y aplicaciones dependen en gran medida del software. Se examinará los defectos inherentes al código y se ofrecerá medidas de seguridad creativas para proteger su integridad y confidencialidad. Se discutirán los métodos que sirvan para prevenir y abordar las vulnerabilidades de seguridad desde las primera fase del desarrollo, desde herramientas de análisis estático y dinámico hasta las mejores prácticas de desarrollo seguro.

Los estudios de casos relevantes destacarán la importancia de la seguridad del código en muchas empresas, y se examinarán nuevos desarrollos como DevSecOps y la integración de IA para la detección eficaz de amenazas.

Además, este trabajo pretende transmitir la importancia de proteger los datos y sistemas en una sociedad interconectada abordando tanto la seguridad informática en general como la seguridad del código en particular. Así mismo, se pretende contribuir a la creación de soluciones eficaces y duraderas en este entorno digital en rápida evolución mediante un enfoque holístico.

2.4.1. Principios Fundamentales de la Seguridad Informática

Debido a la creciente dependencia de la tecnología digital, en la actualidad existen multitud de ciberamenazas que podrían poner en peligro la estabilidad de los sistemas, así como la integridad, la confidencialidad y la disponibilidad de la información. Por ello, el establecer unos principios fundamentales sólidos en materia de ciberseguridad es crucial para comprender y abordar estas dificultades de manera eficaz, los cuales son:

1. **Confidencialidad:** Consisten en la protección de datos personales, garantizando que la información solamente sea accesible a las personas autorizadas, lo que se conoce como mantenimiento de la confidencialidad. Para evitar un acceso no autorizado a datos sensibles, hay que establecer medidas de seguridad como la encriptación y los controles de acceso. Por ende, para salvaguardar tanto la propiedad intelectual como la privacidad de las personas, la confidencialidad es esencial.

2. **Integridad:** Se basa en mantener la veracidad y la autenticidad de los datos. La integridad hace hincapié en evitar la alteración ilícita o involuntaria de la información. Para evitar actualizaciones de datos no autorizadas, se utilizan medidas como el control de versiones y las firmas digitales para garantizar la integridad. Además, para mantener la corrección y validez de la información se requiere de integridad.

3. **Acceso continuo:** Consiste en garantizar la disponibilidad, la cual se conoce como la capacidad de acceder a datos y recursos en el momento oportuno. Para ello, implica poner en marcha estrategias de redundancia, copias de seguridad y recuperación de desastres para garantizar que los sistemas sean accesibles, incluso en circunstancias difíciles. Para evitar la interrupción de servicios y operaciones esenciales, la disponibilidad será un factor fundamental.

4. **Autenticación:** Se basa en los objetivos de verificar la identidad de los usuarios y de garantizar que sólo los usuarios autorizados tienen acceso a los sistemas y datos. Para lo cual se utilizan contraseñas, biometría, tarjetas inteligentes y otros tipos de autenticación.

5. **Autorización:** Consiste en el control de los privilegios de acceso de los usuarios, una vez verificada su identidad, seguido de la concesión de los permisos. Se trata de garantizar que los usuarios sólo tengan el acceso necesario para desempeñar sus funciones y evitar el acceso a información confidencial o sensible. Para reducir la posibilidad de abuso de privilegios, la autorización es crucial.

2.4.2. Seguridad en Código

La integridad, la confidencialidad y la disponibilidad de datos y procesos cruciales pueden verse comprometidas por el código, que es la base de las aplicaciones y sistemas informáticos. Para garantizar la resistencia de las soluciones tecnológicas en un entorno en constante cambio, es crucial comprender las ideas fundamentales de la seguridad del código.

- **Vulnerabilidades en el código:** Encontrar debilidades.

  Las vulnerabilidades en el código son defectos o debilidades que un atacante podría utilizar para obtener acceso o dañar un sistema. Inyecciones SQL, desbordamientos de búfer y problemas con la validación de entradas son algunos ejemplos. Para evitar brechas de seguridad, es imperativo identificar y abordar estas vulnerabilidades.

- **Pruebas de penetración:** Calificación de la solidez del código.

  Las pruebas de penetración simulan ataques reales al código con el fin de encontrar fallos. Con la ayuda de las pruebas de penetración se puede evaluar la resistencia de un sistema a los ataques y encontrar fallos ocultos. Antes de que los atacantes se aprovechen de los puntos débiles, es esencial identificarlos y corregirlos mediante estas pruebas.

- **Desarrollar de forma segura:** Incorporar la seguridad desde el principio.

  Integrar la seguridad en todos los niveles del ciclo de vida del software se conoce como desarrollo seguro. Las mejores prácticas, como la validación de entradas, los privilegios mínimos y el uso de bibliotecas seguras, forman parte de este proceso. Las vulnerabilidades pueden prevenirse desde el principio con la ayuda de un desarrollo seguro.

- **Parches y actualizaciones:** Mantener un código robusto.

  Para hacer frente a las nuevas amenazas y vulnerabilidades, el código debe actualizarse y parchearse con regularidad. Dado que los fallos conocidos pueden ser explotados por los atacantes, mantener el código es esencial para preservar la seguridad.

2.5. DevOps

Desarrollo y operaciones (DevOps) es una práctica de trabajo en el desarrollo del software en la que se unifican o coordinan los procesos de desarrollo y operaciones. Surge debido a que las empresas se encuentran en mercados muy competitivos y en continuo desarrollo donde es vital adaptarse rápidamente a las necesidades de los clientes y generar un producto de calidad en el menor tiempo posible. Como tradicionalmente las tareas de desarrollo y operaciones se encuentran separadamente, es muy común que se produzcan muchas ineficiencias y retrasos a lo largo del desarrollo del software. Por lo tanto, con el objetivo de solucionar estas ineficiencias, se crean las DevOps, donde se coordinan los procesos de desarrollo y operaciones mediante diversas prácticas de comunicación, coordinación e integración.

2.5.1. DevSecOps

Desarrollo, Seguridad y Operaciones (DevSecOps) es una práctica que surge por el mismo problema. Las empresas necesitan desarrollar software en muy poco tiempo, y la seguridad es un proceso fundamental que normalmente se hace al final del desarrollo y que requiere mucho tiempo y energía. Así, con las DevSecOps se integra la seguridad a lo largo de todo el ciclo de vida de desarrollo.

Sin embargo, muchas empresas son reacias a implementar el modelo de DevSecOps porque al tratar de comprobar la seguridad del código a la misma velocidad en la que se desarrolla, surgen muchos problemas y todavía es necesario encontrar la manera de automatizar estas prácticas (que tradicionalmente se han hecho de manera manual) de manera eficiente.

Actualmente las prácticas que ayudan a implementar el modelo DevSecOps son: tratar la seguridad desde el inicio del proceso, realizar prácticas de evaluación continua, establecer unas políticas de trabajo así como unos estándares para facilitar la revisión, aumentar la colaboración entre los equipos de desarrollo y seguridad o tener un experto de seguridad en el equipo de desarrollo.

Aunque todas estas prácticas ayudan a comprobar la seguridad durante el desarrollo del software, todavía son necesarias herramientas de soporte que permitan automatizar las prácticas de seguridad en el desarrollo del código.

2.5.2. Técnicas Black-Box y White-Box

Dentro de la seguridad de la información informática, se utilizan los conceptos de White-Box (Caja Blanca) y Black-Box (Caja Negra) para definir el nivel de acceso que tiene una persona, ya sea para realizar un ataque, que por ejemplo le permita acceder al sistema, como para realizar diferentes pruebas de rendimiento o detección de vulnerabilidades.

- **Método de Caja Negra:** Exploración desde fuera.

  Las pruebas de Caja Negra, también conocidas como Técnica de Caja Negra, simulan un enfoque externo en el que la persona que realiza las pruebas carece de un conocimiento profundo del código fuente y el diseño internos del software. En su lugar, se centra en inspeccionar el software a la manera de un atacante externo y encontrar puntos débiles a través de una exploración exhaustiva y pruebas de penetración. Este método identifica vulnerabilidades como inyecciones SQL, ataques de fuerza bruta y problemas de autenticación que podrían explotarse desde fuera del sistema.

- **Método de Caja Blanca:** Cerrar las lagunas.

  Las pruebas de Caja Blanca se centran en el examen en profundidad de la arquitectura central y el código fuente del software. Con este método, los probadores pueden comprobar cómo se procesan los datos, se llevan a cabo las validaciones y se gestionan las interacciones con pleno acceso al código. Esto permite una detección más precisas de las vulnerabilidades y la identificación de problemas que podrían no ser visibles a primera vista. Las pruebas de Caja Blanca son especialmente útiles para detectar problemas como inyecciones de código y desbordamiento de búfer.

Tanto en el método de Caja Negra como el de Caja Blanca tienen sus propias ventajas. La Técnica de Caja Negra simula una estrategia práctica de asalto externo y es particularmente buena para detectar agujeros externos sin conocimiento interno del sistema. La Técnica de Caja Blanca, por su parte, ofrece un conocimiento profundo del código y es capar de identificar fallos que no serían evidentes desde fuera del sistema. Una estrategia híbrida que combine ambas técnicas puede ofrecer una evaluación de la seguridad más exhaustiva y hacer frente a diversas amenazas potenciales.

2.5.3. Análisis de Código Dinámico y Estático

Debido al gran volumen de software nuevo que se desarrolla cada día con la situación del mercado actual y las DevSecOps, el número de nuevas vulnerabilidades y ataques que se producen también es muy elevado, siendo uno de los focos principales para la seguridad.

El mecanismo más común ante este problema son las herramientas que realizan análisis de código estático, dinámico e híbrido.

El análisis de código estático y dinámico son dos maneras diferentes de abordar el análisis de código para detectar vulnerabilidades. El análisis estático consiste en el estudio del código fuente antes de ejecutar el programa, a través de diversas técnicas como la semántica del código o la búsqueda de patrones; para encontrar vulnerabilidades, siempre sin llegar a ejecutar el código. Al no tener que ejecutar el código, este tipo de análisis es vital en la comprobación de seguridad durante el desarrollo del código, pero como punto negativo estos análisis suelen tener una alta tasa de falsos positivos.

Por otra parte, el análisis dinámico es el estudio del código mediante pruebas de rendimiento realizadas ejecutando el código. Este tipo de prácticas suelen consistir en realizar ejecuciones del programa con diferentes parámetros correctos, incorrectos o aleatorios, como por ejemplo cadenas de caracteres especiales o demasiado largas para comprobar su funcionamiento en situaciones diferentes. Dentro del análisis dinámico una las pruebas más utilizadas actualmente es la prueba *Fuzz* [KRC+18] Para concluir, el análisis de código estático y dinámico ofrecen cosas diferentes y ambos son necesarios para poder llevar a cabo el modelo DevSecOps de manera eficiente, ya que ninguno de los dos es lo suficientemente completo y siguen estando en desarrollo, por lo que en la práctica conviene usar ambos tipos de análisis para asegurarse evitar el mayor número posible de vulnerabilidades.

Capítulo 3

Estado del Arte

Al inicio del proyecto se ha realizado una investigación sobre el Estado del Arte. En esta investigación, se han estudiado las herramientas más recientes que realizan análisis de código estático y que utilizan el AA para aprender las características o patrones del lenguaje de código y detectar las vulnerabilidades.

3.1. Conjunto de datos de entrenamiento

Para poder entrenar una red neuronal, primero es necesario conseguir un gran conjunto de datos con ficheros vulnerables y no vulnerables. Los conjuntos de datos de entrenamiento más utilizados y disponibles son los siguientes:

- **SARD**

  SARD es un proyecto que consiste en la colección de múltiples conjuntos de datos de entrenamiento con vulnerabilidades documentadas [NIS17]. Los conjuntos de datos de SARD están compuestos por una amplia variedad de vulnerabilidades de seguridad de software, incluyendo errores de desbordamiento de búfer, problemas de puntero nulo y vulnerabilidades de inyección de código. Los conjuntos de datos están organizados por lenguaje de programación y versión, y cada conjunto de datos contiene varios archivos de código fuente que ilustran una vulnerabilidad específica. Los programas están escrito en los lenguajes de C, C++, Java y PHP, y cubren más de 150 vulnerabilidades diferentes.

- **Big-vul**

  Big vul [FLWN20] es un conjunto de datos de código C/C++ de proyectos de código abierto de GitHub. Está formado por 3.754 funciones vulnerables y 253.096 funciones no vulnerables, con lo que contamos con una gran muestra para entrenar el modelo y probarlo en funciones de los dos tipos. Sin embargo, a pesar de tener 3.754 funciones vulnerables, estas son de 91 tipos diferentes.

- **Juliet**

  Juliet es uno de los conjunto de datos que podemos encontrar en SARD, es de gran tamaño con muestras de programas enteros. Recoge 64.099 vulnerabilidades y 118 CWEs.

- **mVulPreter**

  La mayoría de propuestas como mVulPreter [ZHL+22] optan por generar su conjunto de datos a partir de varias fuentes entre las mencionadas anteriormente.

- **Funded**

  Funded [WYT+21] es una de las propuestas recogidas en el Estado del Arte que realiza su propio conjunto de datos mediante un entorno de trabajo automático que obtiene muestras del mundo real de páginas como NVD, SARD o GitHub. Principalmente recolectó código de C/C++ y JAVA, aunque también tiene código de PHP y Swift.

- **Vudenc**

  Sin embargo, Vudenc [WNV+22] y Funded, descargan en el momento los datos de entrenamiento a partir de repositorios de código abierto como GitHub. Al hacer una aplicación que descargue automáticamente muestras de repositorios tienen la ventaja que el código se puede ir actualizando con el paso del tiempo. Sin embargo, también hemos encontrado el inconveniente de que muchos de repositorios dejan de ser accesibles, produciendo un error en la ejecución del código, por lo que es necesario actualizar el código o buscar nuevos repositorios.

3.2. Representación de Código

El proceso para la detección de vulnerabilidades está compuesto principalmente de dos fases, la fase de preprocesamiento de los datos y su entrenamiento. El preprocesamiento es un proceso muy importante en este campo, ya que en esta etapa se modifica el conjunto de datos, recogiendo la semántica del grupo, lo que hace que el entrenamiento de la red neuronal pueda ser más efectivo y aumenten los resultados.

Dentro del preprocesamiento se incluyen tareas como la limpieza, la normalización o la transformación de los datos.

- **Limpieza**: Consiste en eliminar aquellos datos atípicos que generan ruido en el modelo y empeoran los resultados.

- **Normalización**: Hace que el código siga unos estándares. Entre ellos puede ser el de eliminar comentarios que no alteren la semántica del programa o asignar nombres genéricos a las variables y a las funciones de los programas (Var1, Var2, Func1,Func2...)

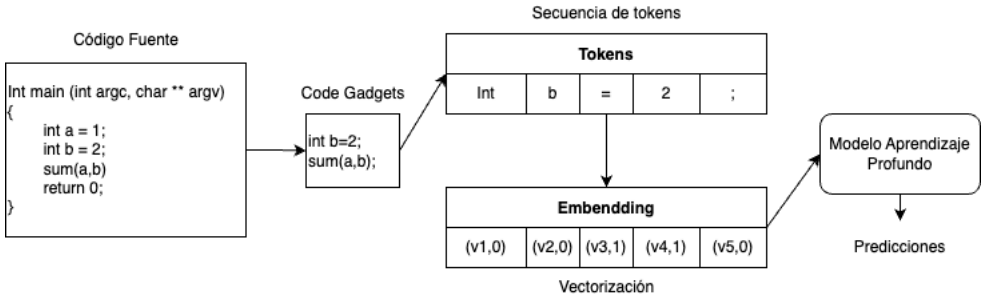
- **Transformación de los datos**: Se conforma por las técnicas que modifican el código de diferentes formas en las que se recogen la semántica del código en un formato más óptimo para la red neuronal. También se incluyen técnicas como la conversión del código a *embeddings*.

Durante esta etapa, las propuestas se diferencian al realizar la transformación de los datos y elegir de qué manera van a recoger la información del lenguaje de código. Las dos metodologías básicas son las representaciones secuenciales y las representaciones mediante grafos.

3.2.1. Representaciones Secuenciales

Como se puede observar en la Figura 3.1, este tipo de metodología busca recoger la semántica del código mediante secuencias de tokens, unidades que forman el lenguaje de programación, como identificadores, operadores, funciones... Es una técnica bastante utilizada en otros ámbitos como la clasificación, la generación o recomendación de código, que permite a los modelos de aprendizaje automático recoger la semántica del código, repeticiones, patrones y dependencias.

Figura 3.1: Arquitectura Modelo Secuencial



Vudenc [WNV+22] y [RKH+18a] utilizan todo el código fuente para poder generar las secuencias de tokens, utilizando una ventana de m caracteres que contiene el contexto al analizar. Como punto negativo de esta implementación, encontramos que al utilizar todo el código fuente se recogen secuencias de código que pueden ser irrelevantes.

[ZWX+21] y [LZX+18] genera agrupaciones de código con las zonas que se consideran que afectan a la vulnerabilidad o puntos de atención (por lo que pueden contener datos de líneas de texto diferentes). Por lo que, como punto negativo, al no utilizar todo el texto, se pueden llegar a omitir zonas que afecten a la vulnerabilidad, y que la generación de las agrupaciones de código requeridas de un mayor uso de parámetros y características diseñadas por humanos, podrían no ser capaces de capturar la semántica oculta en el código fuente, mientras que [RKH+18b] y [WNV+22], dejan que sea el modelo el que realice el análisis de la semántica y determine qué puntos son relevantes, ya que según justifican el código tiene muchas similitudes con el texto de lenguaje natural: repetición de ciertas estructuras y patrones comunes, localidad (las repeticiones ocurren en un contexto local) y dependencias a largo plazo. Además, el código está escrito por humanos, que tienen una tendencia a gravitar hacia patrones convencionales y la repetición de estructuras típicas.

Para realizar el entrenamiento y las predicciones estas propuestas utilizan como redes neuronales diferentes estructuras en las que implementan las LSTM o *Bidirectional Long Short-Term Memory* (BLSTM)

Como se ha explicado previamente en el Capítulo 2, las LSTM son un tipo de RNN diseñado para abordar el problema de la desaparición del gradiente que ocurre con las RNN tradicionales. Por este motivo, las LSTM son actualmente muy populares para tareas como el procesamiento del lenguaje natural, el reconocimiento de voz y la generación de subtítulos de imágenes. La diferencia clave entre un LSTM y una RNN tradicional es que el LSTM tiene una çelda de memoria"que puede mantener información durante un período de tiempo más largo, lo que le permite capturar mejor las dependencias a largo plazo en secuencias, por lo que son ideales en este campo para recoger el mayor número de vulnerabilidades posibles.

En segundo lugar,las BLSTM son una extensión de la arquitectura LSTM que captura información de la secuencia de entrada tanto hacia adelante como hacia atrás en el tiempo. En una red LSTM bidireccional, la secuencia de entrada se alimenta de dos conjuntos de celdas LSTM: uno procesa la secuencia en el orden original y el otro procesa la secuencia en orden inverso. Luego, las salidas de ambas direcciones se combinan para formar la representación final de la secuencia. Esto permite que la red capture tanto las dependencias pasadas como las futuras en la secuencia, lo que puede ser muy útil para tareas de predicción y clasificación.

3.2.2. Representación mediante Grafos

Propuestas del Estado del Arte como [WYT+21], [ZHL+22] o [WZD+22] recogen la semántica del código mediante representaciones de grafos utilizando los *Code Property Graph* (CPG). Representaciones de código en las que se utilizan nodos para representar los objetos y variables del código y ramas, y plasmar las relaciones que tienen los distintos códigos.

Según aparece en [GWXW20], dependiendo del tipo de relaciones que se quieran recoger en el CPG, hay 3 capas principales. Estas capas normalmente se añaden de manera progresiva y en función de las que se escojan se pueden obtener unas características u otras.

1. *Abstract Syntax Tree* (AST). En las AST se encuentran las relaciones entre las declaraciones y expresiones necesarias para ejecutar un programa. Los nodos internos representan los operadores, los nodos de hoja los operadores y los bordes especifican las relaciones de contenedor y contenido.

2. *Control Flow graph* (CFG). Representa el orden que sigue el código y las condiciones de cada ruta. Las declaraciones son representadas por nodos y las condiciones (por ejemplo condicional if True, if False) son las ramas.

3. *Program dependence graph* (PDG). Contiene las dependencias de los datos y de control. Las dependencias de los datos están marcadas con una D y se refieren a las dependencias que modifican un valor, por ejemplo si tenemos:

$$Y = 2 * X, siendo X = calc() \tag{3.1}$$

   El valor de Y depende del resultado de la función calc() por lo que tendrá una dependencia de datos. Por otra parte, las dependencias de control están marcadas con un C y pueden hacer referencia a un condicional o a un bucle.

A la hora de realizar la representación del código y como se explica en [WZY22], se ha visto que estas capas no son incompatibles, se puede utilizar solo una de ellas o todas a la vez. Esta técnica tiene ventajas al aumentar la precisión de la semántica del código, donde el modelo suele tener mayor precisión, también tiene algunas limitaciones como son aumento del tiempo de entrenamiento y el esfuerzo extra necesario para realizar esta técnica. Uno de los inconvenientes de esta técnica es que todas las propuestas revisadas en el estado del arte que utilizaban algún CPG, es que realizaban esta técnicas mediante Joern, una aplicación externa de Python.

En cuanto al AA las propuestas que usas grafos necesitan utilizar GNN, otro tipo de red neuronal diseñada para trabajar con datos estructurados en forma de grafo. Los grafos son una representación matemática de datos donde los puntos de datos (o nodos) están conectados por bordes que representan las relaciones entre ellos.

En las redes neuronales tradicionales, los datos suelen representarse como vectores o matrices, pero esta representación no es adecuada para datos estructurados en forma de grafo. Por otro lado, las GNN pueden procesar directamente datos estructurados en forma de grafo y aprender de las relaciones entre los nodos.

Las GNN suelen operar en un grafo mediante el intercambio de mensajes entre los nodos vecinos y utilizando esta información para actualizar las características de cada nodo. Este proceso se repite iterativamente, permitiendo que la GNN capture información sobre la estructura global del grafo.

Capítulo 4

Metodología

Para la metodología de este trabajo se ha escogido Vudenc, una de las propuestas estudiadas en el Estado del Arte (Capítulo 3) y se ha probado su funcionamiento. Durante el estudio, se ha visto que las bibliotecas que se utilizan en la elaboración de esta propuesta están en continuo desarrollo, haciendo que sea muy común que el código deje de funcionar con las bibliotecas actuales en muy poco tiempo. Debido a este problema, se ha actualizado el código y se ha adaptado la propuesta para probarla con otro conjunto de datos C y C++ y así ver su rendimiento en otros lenguajes de programación.

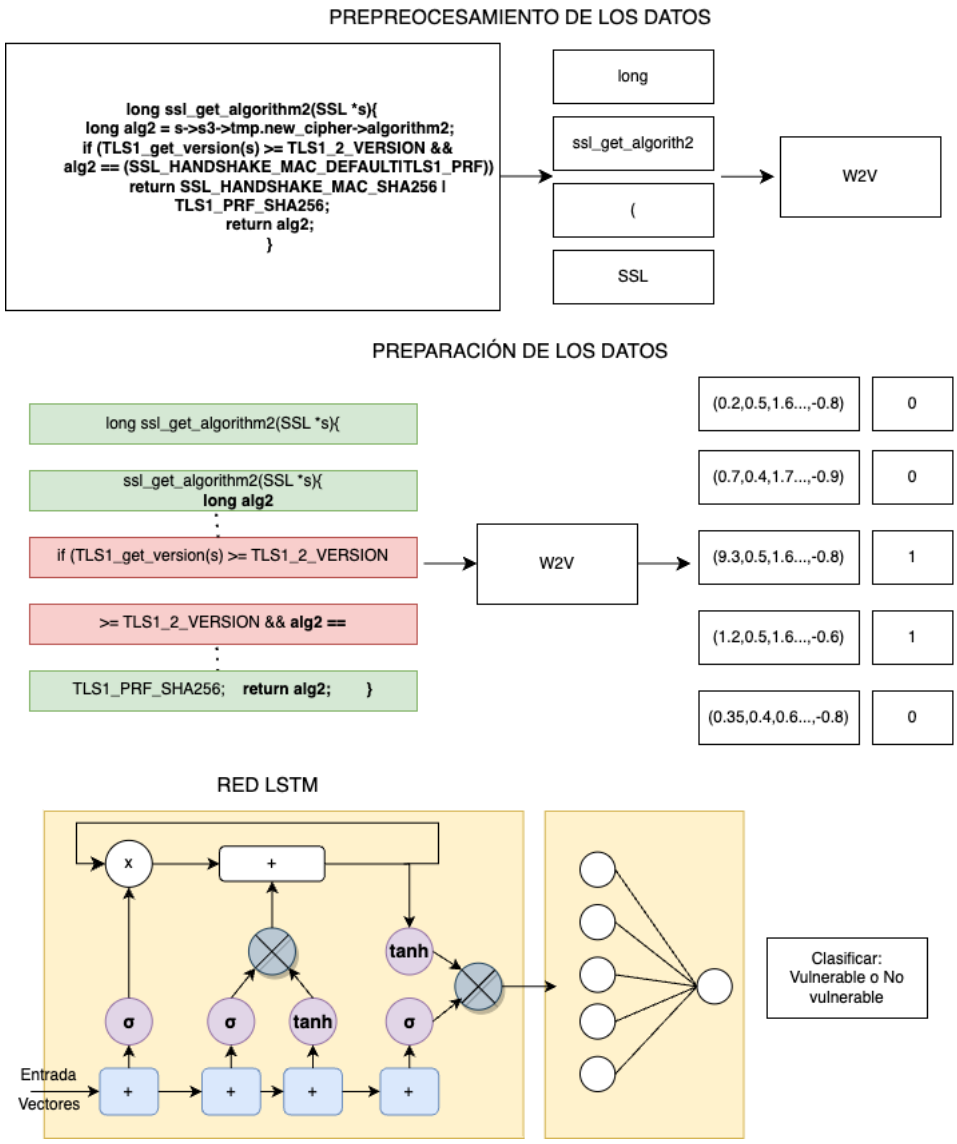
Uno de los motivos para elegir Vudenc como nuestra propuesta, fue el hecho de que toda su propuesta está desarrollada y ejecutada en Python, es decir, no utiliza otras herramientas informáticas como sí lo hacen las propuestas que utilizan representaciones de grafos, las cuales necesitan utilizar Joern para poder generarlos.

Las pruebas y entrenamiento de los datos se realizaron primero en el entorno de Google Colab donde se trabajo con las siguientes especificaciones 25 GB de RAM y una GPU Nvidia V100 o A100 en función de la disponibilidad en ese momento. Posteriormente las pruebas y entrenamientos se realizaron en un ordenador con Windows 10 con CPU Intel Core i7-7920HQ a 3,10 GHz y una memoria RAM de 16,0 GB. Además, se utilizó una GPU con las siguientes especificaciones, GForce RTX 3070 OC Edition con 8 GDDR6.

4.1. Vudenc

Como se ha explicado anteriormente de manera más breve, para aprender los patrones del código fuente con AP, Vudenc sigue una metodología secuencial en la que utiliza un modelo *Word2vec* para identificar tokens de código semánticamente similares y para proporcionar una representación vectorial. Una vez realizado este paso, utiliza la red neuronal LSTM para clasificar las secuencias de tokens de código vulnerables en un nivel detallado, terminando con la generación de una imagen, en la que, para cada área de código, se muestra mediante distintos colores la probabilidad de ser vulnerable o no.

Figura 4.1: Arquitectura Vudenc



La preparación del modelo de Vudenc está dividida en 3 partes como se puede observar en su arquitectura en la Figura 4.1:

1. Preprocesamiento de los datos.
2. Preparación del conjunto de datos.
3. Entrenamiento del modelo LSTM.

4.1.1. Preprocesamiento de los datos

En esta etapa se trabaja con un conjunto de datos para entrenar una red neuronal *Word2vec* que sea capaz de identificar las similitudes del código y realizar la representación del código en vectores numéricos o *embenddings* con la que poder trabajar con el modelo LSTM.

Durante todo el proyecto se ha utilizado el conjunto de datos de Big-vul [FLWN20], un gran conjunto de datos de proyectos en Github con código abierto donde se recogen vulnerabilidades de código C y C++, y el cual está formado por 3.754 funciones vulnerables y 253.096 funciones no vulnerables, con lo que se cuenta con una gran muestra para entrenar el modelo y probarlo en funciones de los dos tipos.

Por otro lado, todo el conjunto de datos se encuentra almacenado en un fichero *CSV* en el que ofrece información detallada para cada función, como el id de la vulnerabilidad, el tipo, proyecto en el que se encuentra o una versión del código vulnerable u otra con la vulnerabilidad corregida. Tal y como queda reflejado en la Tabla 4.1, estas son las columnas necesarias que se han utilizado para este trabajo.

Big-Vul	
Features	Descripcion
Confidentiality Impact	Impacto en la confidencialidad
Integrity Impact	Impacto en la integridad de una vulnerabilidad
Func before	Función vulnerable
Func after	Función corregida
Lines before	Líneas modificadas en la función antes de que se corrija la vulnerabilidad
Lines after	Líneas modificadas en la función después de corregir la vulnerabilidad
Lenguage	C/C++
Vul	1 = Función vulnerable y 0 = Función No vulnerable

Tabla 4.1: Columna de datos de Big-Vul

Como el objetivo del proyecto es entrenar una red neuronal para que luego sea capaz de predecir qué líneas de código son vulnerables, las columnas más interesantes y más útiles para este trabajo son *func before, lines before, func after y lines after.*

En el Código 4.1 se muestra cómo se recogen los datos de los ficheros guardados en *func before* para poder realizar el preprocesamiento.

Como ya se ha mencionado, para realizar el preprocesamiento de los datos se ha utilizado *Word2vec*, una red neuronal enfocada en el procesamiento de lenguaje natural que analiza las relaciones de las palabras del conjunto de datos proporcionado como input de manera automática. Esto es muy útil en el AP siendo utilizado para la búsqueda de sinónimos, la medición de la similitud de palabras, analizar las relaciones o realizar

una categorización. Sin embargo, en este trabajo, *Word2vec* ha sido utilizado para la representación vectorial de las palabras, es decir, convertir las muestras de códigos C en vectores numéricos. Después de entrenar el modelo, este produce vectores numéricos densos para cada palabra en el vocabulario. Estos vectores reflejan la semántica y el significado de las palabras de manera que palabras similares tienen vectores cercanos en el espacio vectorial.

Una vez recogidos los ficheros, para entrenar el modelo *Word2vec* y para procesar el conjunto de datos, es necesario *tokenizar* los ficheros de código extraídos anteriormente (Código 4.2), es decir, dividir el código en unidades más pequeñas como son las funciones, variables, estructuras de control, tipos de datos, operadores...

```
def extraerficheros(pd_filter,output):
  file_cnt = pd_filter.shape[0]
  file_num = 0

  cnt_1 = 0

  for index, row in pd_filter.iterrows():
    file_num += 1
    project_name = row["project"]
    hash_vaule = row['commit_id']
    file_name = project_name + "_" + hash_vaule
    file_name_cnt = 0

    func_before = row['func_before']

    vul_file_name = '1_'+ file_name + '.c'

    with open(output + '/'+ vul_file_name, 'w', encoding='utf-8') as file:
        file.write(func_before)
        cnt_1 += 1
```

Código 4.1: *Raw_Data.py*

```
def tokenizar(folder_path,output_path):

    for filename in os.listdir(folder_path):
        texto = ""
        # Get the full path of the file
        file_path = os.path.join(folder_path, filename)
        base, extension = os.path.splitext(filename)
        # Check if the current file is a file (not a folder)
        count +=1
        print(count)

        if os.path.isfile(file_path):
            # Open the file in read mode using a context manager
            with io.open(file_path, 'rb') as file:
                tokens = tokenize.tokenize(file.readline)
                try:
                    for tok in tokens:

                        #3 es el numero para el string
                        #if (tok.type == 3):
                        #   content = "STRING"
                        #else:
                        content = tok.string
                        texto = texto + content + "\n"

```

```
                    with open(output_path +'/'+ base, "w") as file:
                        file.write(texto)

                except IndentationError:
                    ferror.append(count)
                    continue
                except tokenize.TokenError:
                    ferror.append(count)
                    continue
                except UnicodeEncodeError:
                    ferror.append(count)
                    continue
```

Código 4.2: Función *tokenizar*

En el el Código 4.3, se puede observar como en la primera línea se extraen los ficheros tokenizados, luego se utiliza la biblioteca nltk para guardar los tokens y junto con la definición de los parámetros, se crea la red y se genera el diccionario que relaciona las palabras con los vectores.

- **Dimensiones de los vectores:** Dimensiones o tamaño del vector para cada palabra.
- **Min count:** Parámetro para recortar el vocabulario interno y eliminar aquellas palabras que no aparecen lo suficiente llevando a un contexto y significado muy limitado.
- **Workers**: Opción para realizar paralelización.
- **Iteraciones o *epochs*:** Número de veces que el modelo recorre los datos de entrenamiento.

```
data = ext_files('ficherostoken')

  print('data cargada')

  nltk.download('punkt')

  print('hecho el download')

  if (os.path.isfile('data/pythontraining_processed')):
      with open ('data/pythontraining_processed', 'rb') as fp:
          all_words = pickle.load(fp)
      print("loaded processed model.")
  else:
      print("now processing...")
      processed = data
      all_sentences = nltk.sent_tokenize(processed)
      all_words = [nltk.word_tokenize(sent) for sent in all_sentences]
      print("saving")
      with open('data/pythontraining_processed', 'wb') as fp:
          pickle.dump(all_words, fp)


  #vector_size
  vS = 10
  #min_count
  mC = 10
  #epochs
```

```
    ep = 100
    #workers
    work = 4

    print("processed.\n")

    fname = "../modelosW2V/w2vmodel_" + str(vS)+ "_10.model"
    model = Word2Vec(all_words, vector_size=vS, min_count=mC,
                     epochs=ep, workers = work)
    vocabulary = model.wv.key_to_index
    model.save(fname)
```

Código 4.3: *trainW2V.py*

4.2. Preparación del conjunto de datos

El método propuesto consta de dos redes neuronales, la primera convierte las funciones de código en lenguaje de alto nivel a vectores numéricos. La segunda red neuronal analiza la semántica del código y detecta las partes del código donde posiblemente se encuentre una vulnerabilidad.

Igual que para entrenar el modelo *Word2vec*, lo primero que se ha realizado es la preparación de los datos de entrenamiento que también han sido recogidos de Big-Vul. En este caso, además de utilizar la columna *Func Before* para extraer los ficheros, se ha utilizado la columna *Lines Before* (Código 4.4), ya que resultan muy útiles para poder realizar el etiquetado de los datos y marcar qué secuencias de código son vulnerables.

El modelo Vudenc realiza el preprocesamiento de manera secuencial, una metodología que consiste en la representación textual y el etiquetado de los bloques. Por lo que para estudiar el contexto de cada elemento del código, utiliza una ventana de m caracteres que recorre el texto para guardar el contexto de todos los elementos del código.

En el Código 4.4 se encuentra el bucle que recorre todos los ficheros y llama a las funciones *finfPositions* (Código 4.5), *getblock* (Código 4.6). Estas funciones son necesarias para desplazar la ventana y para realizar el etiquetado del código para clasificarlo como vulnerable o no vulnerable.

Después de guardar los elementos de cada ventana y su etiqueta en *allblocks* como se ve en la Figura 4.1, se procede a dividir cada ventana en tokens y se pasan los datos a un vector numérico con el modelo *Word2vec* que se había generado.

```
def get_allBlocks(pd_filter,full_length,step,num_files = None):
  badparts=[]
  allbadparts=[]
  allblocks = []
  count =0
  for index ,row in pd_filter.iterrows():
    #print(count)
    allbadparts = []
    sourcecode = row["func_before"]
    bad =row["lines_before"]

    if len(sourcecode) > 5000:
      continue
    else:
      count +=1
```

```
      #contiene las lineas vulnerables del fichero
      if count > num_files:
        print(count)
        break
      badparts = bad.split('\n')

      #para cada linea vulnerable
      for bad in badparts:

        #check if they can be found within the file
        if(bad != ""):
          pos = myutils.findposition(bad,sourcecode)
          if not -1 in pos:
            allbadparts.append(bad)

      if(len(allbadparts) > 0):

        positions = myutils.findpositions(allbadparts, sourcecode)
          #get the file split up in samples
        blocks = myutils.getblocks(sourcecode, positions, step, full_length)

        for b in blocks:#each is a tuple of code and label
          allblocks.append(b)
  return allblocks
```

Código 4.4: Función *get_allBlocks*

Función para realizar el etiquetado de las lineas de texto:

```
def findpositions(badparts,sourcecode):

  positions = []
  for bad in badparts:

    if "#" in bad:
      find = bad.find("#")
      bad = bad[:find]

    place = findposition(bad,sourcecode)
    if place != [-1,-1]:
      positions.append(place)
  return positions

return model
```

Código 4.5: Función *findpositions*

```
def getblocks(sourcecode, badpositions, step, fulllength):
    blocks = []
    focus = 0
    lastfocus = 0
    while (True):
        if focus > len(sourcecode):
            break

        focusarea = sourcecode[lastfocus:focus]

        if not (focusarea == "\n"):

            middle = lastfocus+round(0.5*(focus-lastfocus))
            context = getcontextPos(sourcecode,middle,fulllength)
            #print([lastfocus,focus,len(sourcecode)])
```

```
16
17              #context contiene posiciones de un sample
18              if context is not None:
19
20                  vulnerablePos = False
21                  for bad in badpositions:
22
23                      if (context[0] > bad[0] and context[0] <= bad[1]) or
24                      (context[1] > bad[0] and context[1] <= bad[1]) or
25                      (context[0] <= bad[0] and context[1] >= bad[1]):
26                          vulnerablePos = True
27
28                  q = -1
29                  if vulnerablePos:
30                      q = 0
31                  else:
32                      q = 1
33
34                  singleblock = []
35                  singleblock.append(sourcecode[context[0]:context[1]])
36                  singleblock.append(q)
37
38                  already = False
39                  for b in blocks:
40                      if b[0] == singleblock[0]:
41                      #  print("already.")
42                          already = True
43
44                  if not already:
45                      blocks.append(singleblock)
46
47          if ("\n" in sourcecode[focus+1:focus+7]):
48              lastfocus = focus
49              focus = focus + sourcecode[focus+1:focus+7].find("\n")+1
50          else:
51              if nextsplit(sourcecode,focus+step) > -1:
52                  lastfocus = focus
53                  focus = nextsplit(sourcecode,focus+step)
54              else:
55                  if focus < len(sourcecode):
56                      lastfocus = focus
57                      focus = len(sourcecode)
58                  else:
59                      break
60      return blocks
```

Código 4.6: Función *getblocks*

4.3. Entrenamiento del modelo LSTM

En la imagen (Código 5.3) se muestran las líneas de código necesarias para crear la red neuronal LSTM con Keras y como se compila el modelo. Además, se pone a entrenar el modelo en la línea 21 con los datos resultantes del punto 4.2.

Las funciones principales de este código son *model.add*, donde se define que se va a trabajar con una red neuronal LSTM, *model.compile*, donde se añaden la función de pérdida BinaryCrossentropy, el optimizador *ADAM* [KB17] y las métricas *BinaryAccuracy* y *FalseNegatives*.

- **BinaryCrossentropy:**

  Calcula la pérdida de entropía cruzada entre etiquetas verdaderas y etiquetas predichas. Utiliza esta pérdida de entropía cruzada para aplicaciones de clasificación binaria (0 o 1). La función de pérdida requiere las siguientes entradas:

  - **y_true** (Etiqueta verdadera): es 0 o 1.
  - **y_pred** (Valor predicho): Esta es la predicción del modelo, es decir, un único valor de punto flotante que representa un logit o una probabilidad.

- **ADAM:**

  Es un algoritmo que realiza la optimización de funciones objetivo estocásticas basado en gradientes de primer orden, y que se fundamenta en estimaciones adaptativas de momentos de orden inferior. Otros posibles algoritmos con los que se podrían probar son Adagrad, RMSProp y AdaMax.

- **BinaryAccuracy:**

  Es la métrica elegida para realizar el seguimiento de la clasificación del modelo. Representa la proporción de predicciones correctas en relación con el total de muestras.

```
1
2  dropout = 0.2
3  neurons = 100
4  optimizer = "adam"
5  epochs = 100
6  batchsize = 32
7
8
9  model = keras.Sequential()
10
11 model.add(keras.layers.LSTM(neurons, dropout = dropout,
12            recurrent_dropout = dropout))
13 model.add(keras.layers.Dense(1, activation='sigmoid'))
14 model.compile(loss=tf.keras.losses.BinaryCrossentropy(),
15            optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
16            metrics=[tf.keras.metrics.BinaryAccuracy(),
17            tf.keras.metrics.FalseNegatives()])
18
19 class_weights=class_weight.compute_class_weight(class_weight='balanced',
20            classes=numpy.unique(y_Train), y=y_Train)
21 cw = {0:class_weights[0],1:class_weights[1]}
22
23 #training the model
24 history = model.fit(X_train_pad, y_Train, epochs=epochs,
25            batch_size=batchsize, class_weight=cw)
26
27 model.save(argumentos[0]+'/LSTM_model_100_2.h5')
```

Código 4.7: Compilar modelo

4.4. Evaluación del modelo

Para poder llegar a una conclusión sobre el funcionamiento del modelo es necesario compararlo con otros modelos utilizando unas métricas estandarizadas, las cuales son: verdaderos positivos, verdaderos negativos, falsos positivos, falsos negativos, precisión, Recall, exactitud y la puntuación F1.

- **Verdadero positivo (*True Positives* (TP)):** Número de muestras que se han clasificado como vulnerables correctamente.
- **Verdadero negativo (*True Negatives* (TN)):** Número de muestras que se han clasificado como no vulnerables correctamente.
- **Falso positivo (*False Positives* (FP)):** Número de muestras que se han clasificado como vulnerables erróneamente. El número de muestras incorrectamente clasificadas como vulnerables.
- **Falso negativo (*False Negatives* (FN)):** Número de muestras que se han clasificado como no vulnerables erróneamente.
- **Precisión** = Resultado de dividir los verdaderos positivos entre todos los positivos. Indica cómo de preciso es el modelo y cuánto se puede confiar cuando este indica que un bloque de código en vulnerable.

$$\text{Precisión} = \text{TP}/(\text{TP}+\text{FP})$$

- **Recall** = Proporción de positivos que fueron identificados correctamente en comparación con el número total de positivos reales.

$$\text{Recall} = \text{TP}/(\text{TP}+\text{FN})$$

- **F1** = Media armónica de la precisión y el Recall. Es una métrica equilibrada, para evitar posibles resultados engañosos donde un modelo precise siempre un clase muy común, pero es inútil prediciendo otras clases.

$$\text{F1} = 2 * \text{Precisión} * \text{Recall}/(\text{Precisión}+\text{Recall})$$

4.5. Realizar análisis de Código

Finalmente tras entrenar el modelo LSTM, éste se puede utilizar para que realice predicciones en el código e indique cuál es la probabilidad de que ciertas zonas del código sean vulnerables.

Para poder realizar esta tarea se va a necesitar una serie de elementos, los cuales son:

- Un fichero de código C y C++.
- Los modelos W2V y LSTM.
- El Código 4.8, en el que se guardan, en un array, las posiciones de inicio y final de los comentarios que se encuentran en el código.
- El Código 4.9, en el que se comienza generando una imagen RGBA vacía y se realiza un bucle, en el que, para cada iteración, hay un centro de atención que sera la zona que se va a pintar en función del resultado de la perdición del modelo. Luego de manera muy similar a como se hacia en 4.6 hay una ventana de tamaño m que siempre tiene incluida el centro de atención.

  Cada ventana generada en cada iteración se transforma en un vector numérico y se utiliza la red LSTM para que realice su clasificación y en función del valor de la probabilidad obtenida, se pinta el centro de atención de un color u otro.

```
def findComments(sourcecode):
    commentareas = []
    inacomment = False
    star=""
    commentstart = -1
    commentend = -1
    for pos in range(len(sourcecode)):

        if sourcecode[pos] == "/" and sourcecode[pos +1]=="/":
            if not inacomment:
                commentstart = pos
                inacomment = True
                star= "//"
        if sourcecode[pos] == "/" and sourcecode[pos+1]=="*":
            if not inacomment:
                commentstart = pos
                inacomment = True
                star= "/*"
        if sourcecode[pos] == "\n":
            if inacomment and star =="//":
                commentend = pos
                inacomment = False
                star=""
        if sourcecode[pos] == "/" and sourcecode[pos-1]=="*":
            if inacomment and star =="/*":
                commentend = pos
                inacomment = False
                star = ""
        if commentstart >= 0 and commentend >= 0:
            t = [commentstart, commentend+1]
            commentareas.append(t)
            commentstart = -1
            commentend = -1
    return commentareas
```

Código 4.8: Función *findComments*

```
def getblocksVisual(mode,sourcecode, badpositions,commentareas, fulllength,
step, nr,w2v_model, model, threshold,name):

      word_vectors = w2v_model.wv
      ypos = 0
      xpos = 0

      lines = (sourcecode.count("\n"))
      img = Image.new('RGBA', (750, 11*(lines+1)),"WHITE")
      color = "white"
      blocks = []
      focus = 0
      lastfocus = 0
      string = ""

      trueP = False
      falseP = False

      while (True):
        if focus > len(sourcecode):
          break

        comment = False
        for com in commentareas:
          if (focus >= com[0] and focus <= com[1] and lastfocus >= com[0]
          and lastfocus < com[1]):
                focus = com[1]
                comment = True
          if (focus > com[0] and focus <= com[1] and  lastfocus < com[0]):
              focus = com[0]
              comment = False
          elif (lastfocus >= com[0] and lastfocus < com[1] and focus > com[1]):
              focus = com[1]
              comment = True

        focusarea = sourcecode[lastfocus:focus]

        if(focusarea == "\n"):
          string = string + "\n"
        else:
          if comment:
              color = "grey"
              string = string + colored(focusarea,'grey')
          else:

              middle = lastfocus+round(0.5*(focus-lastfocus))
              context = getcontextPos(sourcecode,middle,fulllength)

              if context is not None:

                vulnerablePos = False
                predictionWasMade = False
                text = sourcecode[context[0]:context[1]].replace("\n", " ")
                token = getTokens(text)
                if (len(token) > 1):
                  vectorlist = []
                  for t in token:

                    if t in word_vectors.key_to_index and t != " ":
                      vector = word_vectors[t]
                      vectorlist.append(vector.tolist())
```

```
62
63                  if len(vectorlist) > 0:
64                      p = predict(vectorlist,model)
65                      #print(vectorlist)
66                      if p >= 0:
67                        predictionWasMade = True
68                        if p > threshold[3]:
69                            color = "DarkGreen"
70                        elif p > threshold[6]:
71                            color = "orange"
72                        else:
73                            color = "darkred"
74
75        try:
76          if len(focusarea) > 0:
77            d = ImageDraw.Draw(img)
78
79            for i in range (len(focusarea)):
80                if focusarea[i] == "\n":
81                  ypos = ypos + 11
82                  xpos = 0
83                  d.text((xpos, ypos), focusarea[i], fill=color)
84                  xpos = xpos + d.textsize(focusarea[i])[0]
85                else:
86                  d.text((xpos, ypos), focusarea[i], fill=color)
87                  xpos = xpos + d.textsize(focusarea[i])[0]
88
89        except Exception as e:
90          print(e)
91
92        if ("\n" in sourcecode[focus+1:focus+7]):
93          lastfocus = focus
94          focus = focus + sourcecode[focus+1:focus+7].find("\n")+1
95        else:
96          if nextsplit(sourcecode,focus+step) > -1:
97            lastfocus = focus
98            focus = nextsplit(sourcecode,focus+step)
99          else:
100           if focus < len(sourcecode):
101             lastfocus = focus
102             focus = len(sourcecode)
103           else:
104             break
105     for i in range(1,100):
106       if not os.path.isfile('file' + "_" + str(i) + "_FL_" +
107           str(fulllength) + '.png'):
108               img.save('ile' + "_" + str(i) + "_" + "_FL_" +
109                   str(fulllength) + '.png')
110               print("saved png.")
111               break
```

Código 4.9: Función getblocksVisual

Capítulo 5

Experimentos y Resultados

5.1. Introducción

Para realizar la fase de experimentación se han generado varios modelos en los que se han ido modificando algunos parámetros y se han evaluado los resultados obtenidos en las métricas de precisión, recall y F1. También, se han realizado pruebas con la predicción de código para un único fichero observando cuales han sido los parámetros que han ofrecido mejores resultados en cada situación.

Al comienzo de la ejecución se eligieron los parámetros que Vudenc establece como óptimos [WNV+22]. Sin embargo, al ser un conjunto de datos de entrenamiento diferente, algunos de estos parámetros han sido modificados para optimizar los resultados y reducir la memoria utilizada que sobrepasaba la capacidad del ordenador, y finalmente, se realizaron las pruebas recogidas en la tabla 5.1.

- **W2V**

  Dimensiones de los vectores: Dimensiones o tamaño del vector para cada palabra. Se ha comenzado con un tamaño de 10 y se ha incrementado hasta 50. También se hicieron pruebas con un tamaño de 100 y de 200, pero estas aumentaban demasiado la carga y el tiempo de entrenamiento del modelo.

  Min count: Parámetro para recortar el vocabulario interno y eliminar aquellas palabras que no aparecen lo suficiente llevando a un contexto y significado muy limitado. Se ha confirmado que en el modelo solo se eliminan aquellos token que aparecen de forma esporádica y su valor se ha fijado en 10 para todos los experimentos que se muestran.

  Tamaño de la ventana: Para el tamaño de las secuencias de entrenamiento se han probado ventanas desde 200 hasta 50 m.

  Workers: Opción para realizar paralelización. Todas las pruebas se realizaron con un valor de 4.

- **LSTM**

  Dropout: Parámetro para evitar el sobreajuste o *overfiting*, con un valor de 20 %.

  Neurons: 100

Optimizado: Se ha escogido el optimizador Adam para nuestro modelo.

Iteraciones o *epochs*: Número de veces que el modelo recorre los datos de entrenamiento. El modelo W2V también tiene este parámetro y en ambos el número de iteraciones han sido 100.

Batch size: Este parámetro corresponde al número de datos que tiene cada iteración. Para los ficheros grandes del conjunto de datos se ha establecido en 32 debido a que medidas más grandes como 100 excedían la memoria. Para comprobar el efecto de este parámetro en los experimentos realizados con muestras que solo contenían funciones con un tamaño máximo 1000 caracteres, se han realizado pruebas con Batch size de 32 y de 100. Durante estas pruebas, las métricas no varían demasiado pero la duración del entrenamiento se redujo considerablemente, por lo que conviene aumentarla si el equipo y los datos de entrenamiento lo permiten.

5.2. Experimentos

Experimentos realizados				
Parámetros	Exp.1	Exp.2	Exp.3	Exp.4
Vector Size	10	50	50	50
Epochs LSTM	100	50	100	100
Dropout	0,2	0,2	0.2	0.2
Neurons	100	100	100	100
Batchsize	32	32	100	100
Muestra	5.000	1.000	1.000	1.000
Samples	610.606	104.711	145.358	127.957
Tam. Bloque vulnerable	31,39 %	25,5 %	60,33 %	50,76 %

Tabla 5.1: Experimentos realizados

Para poder determinar el desempeño de todos los experimentos, se han obtenido en la Tabla 5.2 las métricas obtenidas con los datos de entrenamiento, mientras que la Tabla 5.3 muestra las métricas obtenidas con los datos de prueba. Como es lógico, los resultados con los datos que se han utilizado durante el entrenamiento son superiores. Sin embargo, se puede observar como a excepción del experimento 1 los resultados no disminuyen más de 0,5 en ninguna métrica, manteniendo los resultados en datos nuevos que contienen vulnerabilidades similares a las recogidas en los datos de entrenamiento.

En el primer experimento se ha comprobado que el modelo funciona correctamente, sin embargo los resultados obtenidos en las métricas en las Tablas 5.2 y 5.3 no han sido los esperados, ya que han sido inferiores a los obtenidos en otras propuestas del Estado del Arte o a los que se observaban en la versión original de Vudenc para Python.

Por esta razón, se ha decidido modificar algún parámetro para aumentar la precisión del modelo, siendo una de las posibilidades la de aumentar la precisión con la del modelo w2v a la hora de pasar los tokens a vectores numéricos, por lo en el experimento 2 se aumento el *Vector Size* a 50. Sin embargo, este cambio aumentaba el tamaño de los datos, por lo que, para no tener problemas de memoria, se ha optado por disminuir el conjunto de datos y pasar de una muestra de 610.606 a 104.711. Con estas medidas todas las métricas mejoraron, alcanzando mínimo el 90 % y como se puede ver en el Capítulo 5.3, los resultados en la predicción de vulnerabilidades son fructíferos.

Sin embargo, se observó que en aquellos datos de prueba en los que el fichero era muy pequeño, el tamaño de la ventana era demasiado grande y no permitía sacar conclusiones al realizar la predicción. Por este motivo, en los siguientes experimentos se realizó un entrenamiento enfocado a los ficheros del conjunto de datos pequeños, reduciendo la longitud de la ventana. Como resultado de estos últimos dos experimentos, se disminuyó en gran medida el contexto que tiene el modelo para realizar las predicciones, conllevando a que se obtuvieran los peores resultados durante las métricas. Por lo tanto, se ha llegado a la conclusión que esta técnica implementada es mejor para funciones de un cierto tamaño donde se pueda aprovechar el tamaño de la ventana.

Resultados métricas Train				
Métricas	Exp.1	Exp.2	Exp.3	Exp.4
Accuracy	0,74	0,93	0,63	0,77
Precision	0,90	0,98	0,70	0,76
Recall	0,70	0,93	0,66	0,78
F1	0,79	0,96	0,68	0,77

Tabla 5.2: Métricas realizadas con los datos de entrenamiento

Resultados métricas FinalTest				
Métricas	Exp.1	Exp.2	Exp.3	Exp.4
Accuracy	0,79	0,95	0,59	0,72
Precision	0,81	0,90	0,67	0,72
Recall	0,45	0,91	0,64	0,73
F1	0,58	0,91	0,66	0,73

Tabla 5.3: Métricas realizadas con los datos de prueba

5.3. Imágenes

Una vez entrenados varios modelos, se va a elegir cuál va a ser el experimento más óptimo con el que se van a obtener las mejores métricas en los resultados para los datos de *Finaltest*. Para este trabajo, se ha escogido el experimento número 2, ya que se ha considerado que era el que más se adaptaba a la definición del proyecto.

Después de seleccionar el experimento, se han realizado gran cantidad de pruebas para observar los resultados que genera este modelo, prediciendo la salida en un único fichero. Además, de forma complementaria, y con un fin de mayor claridad, se ha generado una imagen donde se refleja el fragmento de código leído, con la diferencia de que las líneas van a estar reflejadas por un color distinto, indicando la probabilidad que tiene esa línea de ser vulnerable o no. Entre estos colores se encuentran:

- **Verde** ($P > 0.6$): Indicará las líneas que no son vulnerables.
- **Naranja** ($0.6 > P > 0.3$): Indicará las líneas que pueden ser o no vulnerables.
- **Rojo** ($0.3 > P$): Indicará las líneas que son muy vulnerables.

Para observar los resultados que se obtienen del modelo elegido, se van a realizar diferentes pruebas sobre varios ficheros, los cuales han sido seleccionados para presentar los diferentes escenarios que han surgido de estos experimentos. En estos ejemplos, se mostrará, por un lado, las líneas de texto que son vulnerables, y por otro lado, la imagen donde se reflejan los colores anteriores sobre ese mismo código, analizando si el modelo ha tenido éxito o no.

5.3.1. Resultado Ejemplo 1

El primer ejemplo que se va a tratar, va a consistir en analizar un fichero que tiene que cumplir las medidas estándar. Esto es un tamaño de código menor a 5.000 caracteres, utilizando una ventana de lectura con una longitud de 200, mismas medidas que se usó para poder configurar el modelo de entrenamiento, al igual que ese fichero no puede corresponder al conjunto de entrenamiento.

Continuando con el análisis de este ejemplo, se puede ver la línea vulnerable (Código 5.1) que existe dentro del fragmento de código a estudiar, y tomando de ayuda la Figura 5.1, se puede ver que el modelo marca de color rojo la zona donde se encuentra esa vulnerabilidad, señalizando de color naranja las zonas de alrededor, y de verde el área más alejada a esa zona vulnerable.

Estos resultados se muestran así, debido a que a la hora de etiquetar las secuencias de texto, se han etiquetado las secuencias que contuvieran parte de la zona vulnerable del código, marcadas por el tamaño de la propia ventana, haciendo que lea más o menos caracteres según su longitud.

```
1
2 maskBuffer = (unsigned char *)gmalloc (row_stride * maskHeight);
3   buffer = (unsigned char *)gmalloc (width * height * 4);
```

Código 5.1: Líneas vulnerables ejemplo 1

Figura 5.1: *Ejemplo 1 con ventana de 200 c*

```
void CairoOutputDev::drawMaskedImage(GfxState *state, Object *ref,
Stream *str, int width, int height,
GfxImageColorMap *colorMap,
Stream *maskStr, int maskWidth,
int maskHeight, GBool maskInvert)
{
  ImageStream *maskImgStr;
  maskImgStr = new ImageStream(maskStr, maskWidth, 1, 1);
  maskImgStr->reset();

   int row_stride = (maskWidth + 3) & ~3;
   unsigned char *maskBuffer;
  maskBuffer = (unsigned char *)gmalloc (row_stride * maskHeight);
   unsigned char *maskDest;
   cairo_surface_t *maskImage;
   cairo_pattern_t *maskPattern;
  Guchar *pix;
  int x, y;

  int invert_bit;

  invert_bit = maskInvert ? 1 : 0;

  for (y = 0; y < maskHeight; y++) {
    pix = maskImgStr->getLine();
    maskDest = maskBuffer + y * row_stride;
    for (x = 0; x < maskWidth; x++) {
      if (pix[x] ^ invert_bit)
*maskDest++ = 0;
      else
*maskDest++ = 255;
    }
  }

  maskImage = cairo_image_surface_create_for_data (maskBuffer, CAIRO_FORMAT_A8,
 maskWidth, maskHeight, row_stride);

  delete maskImgStr;
  maskStr->close();

  unsigned char *buffer;
  unsigned int *dest;
  cairo_surface_t *image;
  cairo_pattern_t *pattern;
  ImageStream *imgStr;
   cairo_matrix_t matrix;
   int is_identity_transform;

  buffer = (unsigned char *)gmalloc (width * height * 4);

   /* TODO: Do we want to cache these? */
   imgStr = new ImageStream(str, width,
   colorMap->getNumPixelComps(),
   colorMap->getBits());
  imgStr->reset();

  /* ICCBased color space doesn't do any color correction
   * so check its underlying color space as well */
  is_identity_transform = colorMap->getColorSpace()->getMode() == csDeviceRGB ||
  (colorMap->getColorSpace()->getMode() == csICCBased &&
   ((GfxICCBasedColorSpace*)colorMap->getColorSpace())->getAlt()->getMode() == csDeviceRGB);

  for (y = 0; y < height; y++) {
    dest = (unsigned int *) (buffer + y * 4 * width);
    pix = imgStr->getLine();
    colorMap->getRGBLine (pix, dest, width);
  }

  image = cairo_image_surface_create_for_data (buffer, CAIRO_FORMAT_RGB24,
 width, height, width * 4);

  if (image == NULL) {
    delete imgStr;
    return;
  }
  pattern = cairo_pattern_create_for_surface (image);
  maskPattern = cairo_pattern_create_for_surface (maskImage);
  if (pattern == NULL) {
    delete imgStr;
    return;
  }

  LOG (printf ("drawMaskedImage %dx%d\n", width, height));

  cairo_matrix_init_translate (&matrix, 0, height);
  cairo_matrix_scale (&matrix, width, -height);

  /* scale the mask to the size of the image unlike softMask */
  cairo_pattern_set_matrix (pattern, &matrix);
  cairo_pattern_set_matrix (maskPattern, &matrix);

  cairo_pattern_set_filter (pattern, CAIRO_FILTER_BILINEAR);
  cairo_set_source (cairo, pattern);
  cairo_mask (cairo, maskPattern);

  if (cairo_shape) {
#if 0
    cairo_rectangle (cairo_shape, 0., 0., width, height);
    cairo_fill (cairo_shape);
#else
    cairo_save (cairo_shape);
    /* this should draw a rectangle the size of the image
     * we use this instead of rect,fill because of the lack
     * of EXTEND_PAD */
    /* NOTE: this will multiply the edges of the image twice */
    cairo_set_source (cairo_shape, pattern);
    cairo_mask (cairo_shape, pattern);
    cairo_restore (cairo_shape);
#endif
  }

  cairo_pattern_destroy (maskPattern);
  cairo_surface_destroy (maskImage);
  cairo_pattern_destroy (pattern);
  cairo_surface_destroy (image);
  free (buffer);
  free (maskBuffer);
  delete imgStr;
}
```

Como ya se ha explicado anteriormente, la longitud de la ventana es de 200 caracteres. Sin embargo, no funciona para todos los casos de igual forma. Por ello, vamos a exponer dos diferentes ejemplos donde se cambia la longitud (tanto reducirla como ampliarla) para poder obtener mejores resultados.

5.3.2. Resultado Ejemplo 2

El primer experimento va a tratar de analizar un fichero más pequeño de lo general (suponemos general a ficheros entre 1.000 y 2.000 caracteres), siendo en este caso de unos 350. Al realizar las pruebas se ha observado que un tamaño de 200 es demasiado grande y no es capaz de mostrar ninguna información útil. Para resolver este problema, se ha reducido el tamaño de la ventana a 50, recogiendo unos resultados mejores y más precisos, ya que con una ventana de longitud 200 cogía casi en su totalidad al fichero, marcándolo entero como vulnerable, tal y como se puede ver en la Figura 5.2.

Por lo tanto para fragmentos más pequeños, lo más recomendable sería realizar un entrenamiento más exclusivo pensando en estas funciones y reduciendo el tamaño de la ventana desde el comienzo del entrenamiento, para obtener resultados parecidos a la Figura 5.3, donde se puede ver con mayor claridad la distinción entre las líneas que son vulnerables y las que no lo son.

A continuación, se encuentran las líneas que son vulnerables en el ejemplo 2.

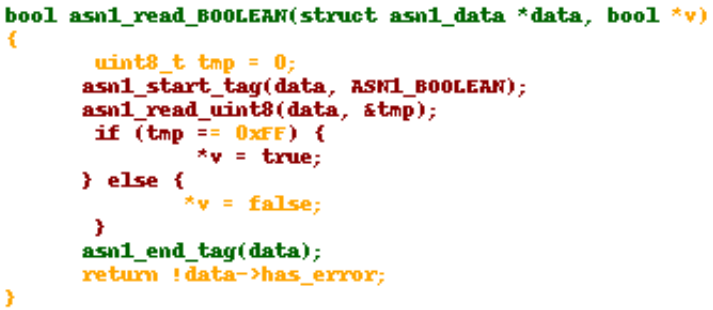
```
1
2 asn1_start_tag(data, ASN1_BOOLEAN);
3 asn1_read_uint8(data, &tmp);
4 } else {
5 *v = false;
6 asn1_end_tag(data);
7 return !data->has_error;
```

Código 5.2: Líneas vulnerables ejemplo 2

Figura 5.2: *Ejemplo 2 con ventana de 200 c*

```
bool asn1_read_BOOLEAN(struct asn1_data *data, bool *v)
{
        uint8_t tmp = 0;
        asn1_start_tag(data, ASN1_BOOLEAN);
        asn1_read_uint8(data, &tmp);
        if (tmp == 0xFF) {
                *v = true;
        } else {
                *v = false;
        }
        asn1_end_tag(data);
        return !data->has_error;
}
```

Figura 5.3: *Ejemplo 2 con ventana de 50 c*



5.3.3. Resultado Ejemplo 3

En el ejemplo anterior se puede ver como el tamaño de la ventana puede ser determinante en los resultados en los ficheros pequeños. Sin embargo, para los ficheros más grandes, pasa todo lo contrario, es decir, hay que aumentar aún más la ventana en vez de disminuirla.

Para mostrar este tipo de ejemplo se han realizado diferentes pruebas para distintos tamaños de ventana. Para el primer caso (Figura 5.4), la longitud consta de 50 caracteres, y como se puede apreciar, el tamaño no es suficiente para recoger todas las vulnerabilidades y por ende, sale más disperso. En el caso de 200 caracteres (Figura 5.5), las líneas vulnerables salen mejor marcadas pero todavía sigue sin ser muy preciso. En el último caso (Figura 5.6), el de 300 caracteres, se puede apreciar que ya el resultado es bastante mejor que las otras dos pruebas, destacando cuáles son las zonas vulnerables dentro del código.

Como ya se viene explicando a lo largo de los ejemplos, para los fragmentos de código más grandes, se va a recomendar a realizar un entrenamiento con un tamaño de ventana mayor, con el fin de precisar y mejorar los resultados cuando se dan este tipo de casos.

A continuación, se encuentran las líneas que son vulnerables dentro del ejemplo 3.

```
1
2 if (timestr->length < 13) {
3
4 php_error_docref(NULL TSRMLS_CC, E_WARNING,
5     "extension author too lazy to parse %s correctly", timestr->data);
6
7 strbuf = estrdup((char *)timestr->data);
8 thestr = strbuf + timestr->length - 3;
```

Código 5.3: Líneas vulnerables ejemplo 3

Figura 5.4: *Ejemplo 3 con ventana de 50 c*

```
static time_t asn1_time_to_time_t(ASN1_UTCTIME * timestr ISRMLS_DC) /* {{{ */
{
/*
This is how the time string is formatted:

   snprintf(p, sizeof(p), "%02d%02d%02d%02d%02d%02dZ",ts->tm_year%100,
      ts->tm_mon+1,ts->tm_mday,ts->tm_hour,ts->tm_min,ts->tm_sec);
*/

time_t ret;
struct tm thetime;
char * strbuf;
        char * thestr;
        long gmadjust = 0;

       if (timestr->length < 13) {
               php_error_docref(NULL ISRMLS_CC, E_WARNING, "extension author too lazy to parse %s correctly", timestr->data);
                return (time_t)-1;
        }

       strbuf = estrdup((char *)timestr->data);

        memset(&thetime, 0, sizeof(thetime));

        /* we work backwards so that we can use atoi more easily */

       thestr = strbuf + timestr->length - 3;

        thetime.tm_sec = atoi(thestr);
        *thestr = '\0';
thetime.tm_mon = atoi(thestr)-1;
*thestr = '\0';
thestr -= 2;
thetime.tm_year = atoi(thestr);

if (thetime.tm_year < 68) {
thetime.tm_year += 100;
}

thetime.tm_isdst = -1;
ret = mktime(&thetime);

#if HAVE_TM_GMTOFF
gmadjust = thetime.tm_gmtoff;
#else
/*
** If correcting for daylight savings time, we set the adjustment to
** the value of timezone - 3600 seconds. Otherwise, we need to overcorrect and
** set the adjustment to the main timezone + 3600 seconds.
*/
gmadjust = -(thetime.tm_isdst ? (long)timezone - 3600 : (long)timezone + 3600);
#endif
ret += gmadjust;

efree(strbuf);

return ret;
}
/* }}} */
```

Figura 5.5: *Ejemplo 3 con ventana de 200 c*

```
static time_t asn1_time_to_time_t(ASN1_UTCTIME * timestr ISRMLS_DC) /* {{{ */
{
/*
This is how the time string is formatted:

   snprintf(p, sizeof(p), "%02d%02d%02d%02d%02d%02dZ",ts->tm_year%100,
      ts->tm_mon+1,ts->tm_mday,ts->tm_hour,ts->tm_min,ts->tm_sec);
*/

time_t ret;
struct tm thetime;
char * strbuf;
        char * thestr;
        long gmadjust = 0;

       if (timestr->length < 13) {
               php_error_docref(NULL ISRMLS_CC, E_WARNING, "extension author too lazy to parse %s correctly", timestr->data);
                return (time_t)-1;
        }

       strbuf = estrdup((char *)timestr->data);

        memset(&thetime, 0, sizeof(thetime));

        /* we work backwards so that we can use atoi more easily */

       thestr = strbuf + timestr->length - 3;

        thetime.tm_sec = atoi(thestr);
        *thestr = '\0';
thetime.tm_mon = atoi(thestr)-1;
*thestr = '\0';
thestr -= 2;
thetime.tm_year = atoi(thestr);

if (thetime.tm_year < 68) {
thetime.tm_year += 100;
}

thetime.tm_isdst = -1;
ret = mktime(&thetime);

#if HAVE_TM_GMTOFF
gmadjust = thetime.tm_gmtoff;
#else
/*
** If correcting for daylight savings time, we set the adjustment to
** the value of timezone - 3600 seconds. Otherwise, we need to overcorrect and
** set the adjustment to the main timezone + 3600 seconds.
*/
gmadjust = -(thetime.tm_isdst ? (long)timezone - 3600 : (long)timezone + 3600);
#endif
ret += gmadjust;

efree(strbuf);

return ret;
}
/* }}} */
```

Figura 5.6: *Ejemplo 3 con ventana de 300 c*

```
static time_t asn1_time_to_time_t(ASN1_UTCTIME * timestr ISRMLS_DC) /* {{{ */
{
/*
This is how the time string is formatted:

   snprintf(p, sizeof(p), "%02d%02d%02d%02d%02d%02dZ",ts->tm_year%100,
      ts->tm_mon+1,ts->tm_mday,ts->tm_hour,ts->tm_min,ts->tm_sec);
*/

time_t ret;
struct tm thetime;
char * strbuf;
        char * thestr;
        long gmadjust = 0;

       if (timestr->length < 13) {
               php_error_docref(NULL ISRMLS_CC, E_WARNING, "extension author too lazy to parse %s correctly", timestr->data);
                return (time_t)-1;
        }

       strbuf = estrdup((char *)timestr->data);

        memset(&thetime, 0, sizeof(thetime));

        /* we work backwards so that we can use atoi more easily */

       thestr = strbuf + timestr->length - 3;

        thetime.tm_sec = atoi(thestr);
        *thestr = '\0';
thetime.tm_mon = atoi(thestr)-1;
*thestr = '\0';
thestr -= 2;
thetime.tm_year = atoi(thestr);

if (thetime.tm_year < 68) {
thetime.tm_year += 100;
}

thetime.tm_isdst = -1;
ret = mktime(&thetime);

#if HAVE_TM_GMTOFF
gmadjust = thetime.tm_gmtoff;
#else
/*
** If correcting for daylight savings time, we set the adjustment to
** the value of timezone - 3600 seconds. Otherwise, we need to overcorrect and
** set the adjustment to the main timezone + 3600 seconds.
*/
gmadjust = -(thetime.tm_isdst ? (long)timezone - 3600 : (long)timezone + 3600);
#endif
ret += gmadjust;

efree(strbuf);

return ret;
}
/* }}} */
```

5.3.4. Resultado Ejemplo 4

En este ejemplo se va a analizar un fragmento de código que formó parte del conjunto de ficheros, el cual se utilizó para configurar el modelo de entrenamiento. Además, gracias al estudio de este fragmento, se podrá ver si existe o no mucha diferencia con los ejemplos presentados anteriormente.

Tal y como se puede ver en la Figura 5.7, la representación gráfica es muy parecida a las ya analizadas, sin embargo, esta última es mucho más clara que el resto, ya que tiene mejor marcada e identificada por los colores correctos, la zona más vulnerable del fichero escogido.

```

*session_data_size = psession.size;
```

Código 5.4: Línea vulnerable en ejemplo 4

Figura 5.7: *Ejemplo 4 con ventana de 200 c*

```
gnutls_session_get_data (gnutls_session_t session,
                         void *session_data, size_t * session_data_size)
{

  gnutls_datum_t psession;
  int ret;

  if (session->internals.resumable == RESUME_FALSE)
    return GNUTLS_E_INVALID_SESSION;

  psession.data = session_data;

  ret = _gnutls_session_pack (session, &psession);
  if (ret < 0)
    {
       gnutls_assert ();
       return ret;
    }
  *session_data_size = psession.size;

   if (psession.size > *session_data_size)
     {
       ret = GNUTLS_E_SHORT_MEMORY_BUFFER;
       goto error;
     }

   if (session_data != NULL)
     memcpy (session_data, psession.data, psession.size);

  ret = 0;

error:
  _gnutls_free_datum (&psession);
  return ret;
}
```

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Una vez realizadas y analizadas todas las pruebas anteriores, y tras haber comparado unas con otras, se ha podido sacar en claro una serie de conclusiones en relación a lo presentado en este estudio.

En primer lugar, y como ya se viene mencionando durante todo el trabajo, las tecnologías están cada vez más en auge, y debido a su rápido desarrollo, está provocando que tanto el análisis como la detección manual de vulnerabilidades sea aún más complejo, conllevando al uso de otras aplicaciones informáticas o nuevas tecnologías, para poder solucionar este tipo de problemas; haciendo que esta idea sea parte principal del motor en que se basa este estudio.

De esta forma, el desarrollo del modelo creado con AP ha resultado muy útil y práctico para poder elaborar este trabajo, ya que, a través del ajuste de diferentes parámetros, se ha podido detectar un número mayor de vulnerabilidades de código que si se hubiese hecho de forma manual, ahorrándose una gran cantidad de errores humanos.

Además, gracias a la implementación de la imagen coloreada según el grado de vulnerabilidad, se puede ver más claramente la simulación de las pruebas y ser una herramienta muy beneficiosa para el programador, ya que puede acceder con más rapidez y así detectar cuáles son las zonas más problemáticas del código, con el fin de corregirlas y mejorar la seguridad de los sistemas.

Por otra parte, como ya se ha visto en el Capítulo 5, tanto la longitud de la ventana de lectura como el tamaño de los fragmentos de código guarda una importante relación, ya que para el correcto funcionamiento de este modelo, es fundamental que ambos parámetros guarden una escala parecida. Esto es, para ficheros pequeños, la ventana tendrá que tener un tamaño más reducido, y viceversa, porque de otra forma, la simulación y la imagen sobre las vulnerabilidades saldrá completamente alterada y muy distinta de la realidad.

Por último, cabe decir que si se cumplen todos los parámetros, el modelo funciona a la perfección. Como ya se ha visto en las pruebas del capítulo anterior, las simulaciones de los ficheros que no formaban parte del conjunto de entrenamiento, eran muy parecidas a las que sí configuraron el modelo. Además, tanto la precisión como la exactitud en los aciertos, han sido valores muy elevados, indicando que esta herramienta ha cumplido con creces los objetivos buscados, y con el uso de alguna aplicación complementaria, puede ser

de gran uso en el mundo tecnológico.

6.2. Trabajo Futuro

Como último punto de esta memoria, se van a presentar diferentes alternativas que se podrían realizar de forma complementaria a este trabajo. Estos trabajos futuros tendrán el objetivo de mejorar las estadísticas que se han conseguido en este para elaborar un proyecto más profundo y de mayor nivel, ya que, aunque se hayan generado buenos resultados, el modelo tiene algunas limitaciones como problemas con exceso de la memoria o el tamaño de la ventana en función del fichero que se analice.

En primer lugar, se podría buscar la manera de incrementar tanto el número de ficheros que pueden formar el conjunto de configuración como el tamaño de estos. Con esta implementación, el modelo sería más completo y tanto las simulaciones como los valores saldrían más ajustados, haciendo que la detección de vulnerabilidades fuera más exacta. De esta forma, las líneas de código vulnerables serían mejor identificadas, y por ende, las imágenes más claras. Sin embargo, para poder llevar a cabo este desarrollo, habría que disponer de un equipo informático muy potente y capaz de soportar gran cantidad de información.

Otra de las alternativas, y relacionada con las pruebas hechas anteriormente, tiene que ver con regular el tamaño de los ficheros y de la ventana de lectura. Con este punto, se hace referencia a que el propio modelo tenga la suficiente inteligencia para poder variar la longitud de la ventana y adaptarla según los caracteres que tiene cada fragmento de código. De esta forma, las simulaciones se realizarían correctamente, pero sobre todo, no habría distinción a la hora del entrenamiento y tampoco habría que modificarla cada vez que el tamaño de fichero no concuerda con el de la ventana, facilitando la labor del programador.

Finalmente, este trabajo tendrá un objetivo final, el cual consiste en desarrollar un plugin para algunas herramientas, como puede ser VisualStudio, con el que se pueda realizar el análisis de código en tiempo real, es decir, un plugin que avise al programador si está escribiendo líneas de código vulnerables, con el fin de corregirlas al momento y así evitar problemas futuros en el programa.

Capítulo 7

Capítulo de Contribución

En este apartado se resumen las tareas realizadas por cada uno de los miembros del proyecto.

7.1. José María García Herranz

Tras la reunión inicial, en la que los tutores nos presentaron las bases del proyecto y de la metodología que se iba a utilizar a lo largo del curso, se preparó un calendario aproximado para organizar todas las tareas que se nos indicaron, con el fin de completar el trabajo en un cierto tiempo.

Durante los primeros meses del proyecto, previo a la fase de investigación, al no tener muchos conocimientos sobre el Aprendizaje Automático, tuve que realizar varios cursos informativos que presentaron los tutores. Estos cursos, llamados 'Supervised Machine Learning: Regression and Classification', 'Advanced Learning Algorithms' y 'Unsupervised Learning, Recommenders, Reinforcement Learning', fueron en inglés y sirvieron para conocer más a fondo estos temas.

Como la mayoría de los modelos de DL y ML se pueden y se suelen programar con Python, también tuve que hacer otro curso de este lenguaje para reafirmar los conocimientos adquiridos en la facultad, llamado 'Curso Maestro de Python', con el que aprendí a programar en este lenguaje, ya que no lo había visto antes, sirviendo además, para desarrollar una aplicación informática sobre proyección de cartera de seguros para el TFG de ADE, para complementar los conocimientos y relacionar ambos grados. Esta primera fase de aprendizaje y familiarización con los diferentes temas, duró alrededor de un par de meses, donde además, se trató de buscar y recoger información de otros proyectos que utilizaban estas técnicas para un mejor aprendizaje y poder llegar a utilizarlos como guías para ayudarme a elaborar este proyecto.

Una vez adquiridos unos conocimientos básicos sobre estos temas, comenzó la fase de investigación, con la que se comenzó a buscar artículos científicos, gracias a la herramienta *Google Schoolar*, centrados en este campo de estudio para poder entender este tipo de tecnologías. Después se siguió con la recolección de información sobre las GANS y su aplicación en los sistemas informáticos, estudiando su funcionamiento en ataques como en defensas e identificando las diferentes técnicas y algoritmos que aplicaban y los conjuntos de datos que usaban, recogido a través de la técnica de *Web Scrapping*.

Al ser varios miembros dentro del grupo de trabajo, junto a mi compañero Sergio, se nos asignó el trabajo de buscar modelos y algoritmos que utilizaran las redes neurales en defensa, enfocándose en la búsqueda de vulnerabilidades dentro del código para cualquier tipo de lenguaje, recogiendo así estas técnicas dentro del Estado del Arte.

Para poder entrar en la fase de desarrollo, primero, entre Sergio y yo, decidimos estudiar un modelo cada uno de entre las propuestas analizadas anteriormente para entender el funcionamiento. Este modelo fue NJsscan [MFBJ21], el cual es una herramienta de prueba de aplicaciones estáticas que puede encontrar patrones de código inseguros en sus aplicaciones *node.js*, utilizando un comparador de patrones simple de *libsast* y la herramienta de búsqueda de patrones de código semántico consciente de la sintaxis *semgrep*, tal y como se puede ver en su GitHub: https://github.com/ajinabraham/njsscan.

Una vez analizado este modelo, fue compararlo con el otro que había estudiado Sergio, sacando diferentes conclusiones sobre ambas opciones y decantándonos por el modelo Vudenc, ya que esta propuesta era más llamativa, por el uso de las RNN y porque era más completo, además de que el lenguaje utilizado nos era más fácil de modificar. También, su configuración y aplicación eran más sencillas. Por ello, nos enfocamos en el mismo modelo y empezamos a desarrollar la propuesta de este trabajo, la cual consistió en adaptar el modelo Vudenc a otro tipo de lenguaje de programación, aplicando librerías como *Tenserflow o Keras*. Además, al no haber analizado esta propuesta, tuve que pasar un tiempo para estudiar y entender su funcionamiento, con la ayuda de mi compañero en todo momento, resolviéndome las dudas en cualquier momento.

A partir de ahí, ambos colaboramos conjuntamente con la adaptación al nuevo lenguaje, buscando primeramente conjuntos de datos que nos sirvieran para poder utilizarlos en nuestro modelo. Para ello, los que encontramos los descargamos y los analizamos para ver cuál encajaba mejor en nuestro proyecto, siendo Big-Vul el elegido, haciendo que adaptáramos Vudenc de Python a C/C++. Una vez terminado, empezamos a generar el modelo de entrenamiento que nos serviría para conseguir los resultados esperados. Durante esta fase de experimentación, se desarrolló el proyecto con diferentes parámetros para buscar la optimización del modelo, generando varios datos de entrenamiento con distintos atributos. De entre estos experimentos, fue escogido el que mejor resultados generó, ayudándonos de la imagen que se creaba por las vulnerabilidades, finalizando así este trabajo de fin de grado.

Paralelamente a estas dos últimas fases, se contribuyó a redactar todos los puntos de este estudio, dedicando gran parte del tiempo a revisar la redacción de la memoria, con el fin de que quedara bien estructurada y sin errores ortográficos.

7.2. Sergio Muñoz Martín

Al comienzo del proyecto, tuvimos unas primeras reuniones donde se presentaron la metodología con la que se iba a trabajar y cuales eran algunos de los temas del proyecto como: la seguridad de la información, las *DevSecOps*, y la Inteligencia Artificial. Conocidos los temas del proyecto comencé a buscar información y realizar cursos que trataran sobre estos temas en los que no había trabajado en profundidad anteriormente. Comencé realizando unos cursos básicos de Inteligencia Artificial a traves de la pagina de Kaggle: 'Intro to Machine Learning' y 'Intermediate Machine Learning'.

Realizados estos cursos, para ampliar los conocimientos de la seguridad y de las DevSecOps, leí y realice resúmenes de varias de los artículos y libros que los tutores nos proporcionaron como referencias. Sobre la Seguridad de la Información estudie varios temas de los libros [Hsu18] y [WM21] en los que vi los principios de las seguridad de las información y cual es el papel de la seguridad dentro de las *DevOps*. Sobre las *DevSecOps* leí el articulo [RZBS22], una review que explica las diferencias entre las *DevOps* y las *DevSecOps* y cuales son las medidas necesarias para implementarlas en las empresas. Para finalizar la introducción a los temas del proyecto leí el articulo ciéntifico [DGC+20] que presenta una review sobre las redes GANS con la que estudie como funcionaban y cuales eran las principales diferencias entre este tipo de redes de aprendizaje automático y el resto de modelos que había visto durante los cursos.

Finalizado estos primeros meses, mis tutores me dieron la tarea de buscar información reciente sobre propuestas que usaran Aprendizaje Profundo para defenderse de ataques o como mecanismo de seguridad. Así comencé buscando información a través de la de herramienta *Google Schoolar* sobre redes GANS, que se utilizaran como defensas pero no encontré ninguna información útil para el proyecto, tan solo encontré ejemplos de ataques.

Por lo que pase a buscar otros tipos de redes de Aprendizaje Profundo que se pudieran utilizar como medidas de defensa en las *DevSecOps*, y termine encontrando el articulo [WYT+21], en el que se presenta *Funded* una propuesta para predecir vulnerabilidades con Aprendizaje Profundo que mas tarde paso a formar el estado del arte. Pase una semana estudiando este artículo, analizando como funcionaba y cuales eran las herramientas que utilizaba y después de presentarlo como una posible medida de defensa en las *DevSecOps* empecé a buscar más artículos similares con el objetivo de formar el estado del arte, busque propuestas que realizaran las predicciones para diferentes lenguajes, que usaran técnicas de aprendizaje diferentes o que llevaran los resultados a otro nivel como podría ser por ejemplo realizar las predicciones en el tiempo real y ofrecer algún consejo. Este proceso duro un par de meses en los que analice las propuestas que se encuentran en el estado del arte y otras que finalmente no entraron. Al mismo tiempo, busque más información sobre los CPG, sus distintos niveles y sobre el funcionamiento de Joern,la herramienta mas popular que se utiliza en todos los artículos que encontré para implementarlos, buscando la manera de implementar los grafos sin utilizar Joern pero con igual o mejor efectividad.

Tras finalizar el estado del arte, para poder entrar en la fase de desarrollo, entre Jose María y yo, decidimos estudiar un modelo cada uno de entre las propuestas analizadas anteriormente para entender el funcionamiento, ejecutarlas en nuestros propios ordenadores y ver los resultados que se obtenían. El modelo que elegí fue Vudenc la propuesta que más me había gustado y que más adelante adaptaríamos.

Durante el proceso para probar los resultados de Vudenc tuve más problemas de los esperados inicialmente. Al no encontrar un fichero de requisitos para saber que versiones de los paquetes tenia que instalar, durante la ejecución del código me enfrente a varios problemas con funciones que en las últimas versiones de sus bibliotecas habían sido eliminadas o modificadas y ya no devolvían el mismo tipo de datos. Por lo que empecé a actualizar el código para que fuera posible ejecutarlo con las versiones actuales de los paquetes, evitando así posibles incompatibilidades buscando la versión correcta de cada paquete.

También tuve problemas de compatibilidad con mi gráfica y la instalación del paquete Tensorflow. Este paquete es fundamental para implementar el modelo de Aprendizaje Profundo, por lo que estuve varias semanas tratando de solucionar este problema y finalmente pase a realizar las pruebas a través del entorno de *Google Scholar* donde tras actualizar el código y pasarlo a formato *notebook* si que pude ejecutar las pruebas de Vudenc. Debido a los problemas y a tener que actualizar el código durante la revisión del código de Tensorflow realice el *Curso de Introducción a Tensorflow* en DataCamp.

Cuando conseguir que el código de Vudenc me mostrara unos resultados, Jose Maria y yo presentamos cada uno la propuesta que habiamos probando cada y elegimos Vudenc como la mejor propuesta en la que seguir trabajando y decidimos adaptar el modelo Vudenc a otro tipo de lenguaje de programación y utilizando un conjunto de datos diferente al que se utilizaba en el modelo original, aplicando librerías como *Tenserflow o Keras.* A partir de ahí, ambos colaboramos conjuntamente con la adaptación al nuevo lenguaje,buscando en primer lugar conjuntos de datos para diferentes lenguajes que nos sirvieran para poder utilizarlos en nuestro modelo. Para ello, los que encontramos los descargamos y los analizamos para ver cuál encajaba mejor en nuestro proyecto, siendo Big-Vul el elegido, haciendo que adaptáramos Vudenc de Python a C/C++. Una vez terminado, empezamos a generar el modelo de entrenamiento que nos serviría para conseguir los resultados esperados. Durante esta fase de experimentación, se desarrolló el proyecto con diferentes parámetros para buscar la optimización del modelo, generando varios datos de entrenamiento con distintos atributos. De entre estos experimentos, fue escogido el que mejor resultados generó, ayudándonos de la imagen que se creaba por las vulnerabilidades, finalizando así este trabajo de fin de grado.

Paralelamente a estas dos últimas fases, se contribuyó a redactar todos los puntos de este estudio, dedicando gran parte del tiempo a revisar la redacción de la memoria, con el fin de que quedara bien estructurada y sin errores ortográficos.

Capítulo 8

Introduction

8.1. Motivation

During the last decades, due to the numerous advances and developments in today's technological world, the security of informatic systems has become an especially essential topic, above all, the security inside software's code.

These changes have caused an ever increasingly dependence on digital platforms as they continue to evolve, increasing the possibilities that malicious attacks such as the spread of new informatic threats could be carried out. Consequently, to these actions, the protection of the integrity and the confidentiality of the data as well as the availability of systems have become fundamental.

Furthermore, in the area of software engineering, one of the main topics to guarantee systems security resides in the detection and prevention of vulnerabilities inside the code. Currently, due to the complexity of the systems and the entrance of new technologies, the analysis and manual detection of vulnerabilities has become a task increasingly difficult and prone to human error.

In this sense, technologies like automatic learning come up as a possible solution to improve the efficiency and effectiveness of the security trials in the development of the software.

Therefore, in the present work, we will conduct a study on the different technics of artificial intelligence that can be useful for the static analysis of the code in the programing language of C and C++.

8.2. Context

The present final degree work is framed inside the investigation project under the title: Platform for Analysis of Resilient and Secure Software – LAZARUS, approved by the European Commission inside the Program Horizon Frame (call HORIZON-CL3-2021-CS-01) in virtue of the agreement of the grant number 101070303 with the participation of Group GASS of the Universidad Complutense de Madrid (Analysis, Security and Systems Group https://gass.ucm.es, group 910623 of the groups of investigation catalogue recognized by the UCM).

Besides the Universidad Complutense de Madrid the following entities participate in LAZARUS: Athena Research Center – ARC (Greece), The University of Padua (Italy), Infotrend Innovations Company Limited (Cyprus), Data Centric Services SRL (Romania), Luxembourg Institute of Science and Technology (Luxembourg), Motivian EOOD (Bulgary), Binare Oy (Finland), Fundación APWG European Union Foundation (Spain), Maggioli Spa (Italy).

More information on:

https://cordis.europa.eu/project/id/101070303

https://lazarus-he.eu

8.3. Object of the Investigation

The code, in the context of the programming, is formed by an ensemble of instructions that the developer commands to execute to the computer. Nevertheless, errors may happen, at the time of programming, which are not visible at first glance, creating what is known as vulnerabilities. These can be due to two different reasons: a mistake in the code of the developed app or a code error in the used libraries. For this reason, they began to develop models with the aim to reduce these types of problems.

Nowadays, there are different models that are capable of finding vulnerabilities inside the code. However, one of their disadvantages is that they are just focused on one type of language, and not only that but they are not very precise, since sometimes they don't point out the lines that are problematic or the percentage of reliability.

With all this intp account, the functionality of this end of degree work is going to consist of designing and implementing the model of Machine Learning that is capable of carring out the identification of vulnerabilities in bits of the code given in the programming language C and C++, with the aim to improve the identification of bugs and reduce the analysis time. Furthermore, this model will allow to make the review of vulnerabilities in the code in a brief time period trying to help developers generate pieces of software that are increasingly more secure and human error- proof in matters of security.

To conduct this idea, we will use the frame of standard trails to find the best parameters and training methods to mold the model. Likewise, this project pretends to understand the operation of the *Recurrent Neural Networks* (**RNN**), specifically the *Long Short-Term Memories* (**LSTM**) and thus know the structure and training, besides the challenges that present with designing one from scratch.

8.4. Workplan

The work has been developed in three separate phases:

1. **Research:** To start, an adaptation period was held during the first four months with the aim of understanding the context of the work and acquiring the level of necessary knowledge to begin the later development. The first thing that was done in this phase was a general meeting in which different points were raised, such as the objectives to achieve, a small guide of how to begin the work, and the process of investigation and which was the necessary knowledge to complete the work. Equally, it was agreed to hold weekly meetings to follow up on the process and resolve the emerging doubts. Furthermore, the tutors dedicated themselves to explain the different concepts over the fields that concern this work, which will be discussed in following chapters, because the members did not have the knowledge on Automatic Learning or Artificial Intelligence and their uses, recommending different tools to facilitate the search of information. Among them *Google Scholar* should be noted, since it was useful to find all kinds of scientific articles centered in the same study field and thus used them to understand this type of technologies better. Lastly, before beginning with the development of this project multiple conclusions and opinions were taken for the first molding and its requirements.

2. **Development**: Once the basic and necessary knowledge for the work had been acquired, the second phase began where less time was dedicated to investigation and the work with the collected information for the coding of the proposal began. In this way, the search for information was just based on emerging concepts during this phase, such as libraries like *Tensorflow* or *Keras* or programming concepts from Python. Similarly, these sources of information that were supplied by other scientific works have been the base and guide to model the training sets.

3. **Results**: In this last phase the prototypes of the initial idea through the implementation of the different tools included in the previously explained libraries began to be developed. Additionally, in this phase the obtained results were compared with other works and the parameters to set up the model were accordingly adjusted. Simultaneously, the work on the development of the project was continued with the goal of searching the optimization in the model.

8.5. Structure of the Work

The remaining study will be organized in different chapters. In Chapter 2 some key concepts will be introduced in order to understand the context of the investigation. Among these elements we will include the security of the information and the kinds of attacks, it will also explain the practice of *DevOps y DevSecOps* and the accesses to the black box and the white box.

Chapter 3 will be focused on the state of the art, in which the main focus will be the programming languages of C/C++ and Python. Firstly, there will be a small introduction, followed by an explanation about the data training sets most used. Afterwards, the representation of the code will be presented, through graphs or texts. Lastly, training on the model to detect the vulnerabilities on the previously mentioned languages will be carried out, discussing all the steps taken.

Within the studied proposals in the stat of art, Chapter 4 will serve to focus on one of them, which in this case will be the Vundenc model. Furthermore, it will explain which is its performance and how the two models that compose it, are generated and trained. Moreover, analysis on the code using these tools will be conducted to see what results it produces.

Chapter 5 will describe the experiments performed to evaluate the effectiveness of the used algorithms in the previous phase, presenting the sets of data that have been selected. Likewise, the results that come out will be analyzed and compared with the ones obtained on other models.

Chapter 6 will be used to reflect on the conclusions that will be developed from the work. Consequently, it will present the future studies that can help complement this study.

The contribution of the team members will be shown in Chapter 7, indicating which parts and what functions have carried out each.

Lastly, Chapter 8 and 9 will be the English translation of the introduction (Chapter 1) and the conclusions (Chapter 6).

8.6. Gantt's Diagram

Figura 8.1: *Gantt's Diagram*

Tasks	Dur	Beg. Date	End Date	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Ago
Curses Python, ML	4 weeks	05/09/22	30/09/22												
DevSecOps	2 weeks	01/10/22	15/10/22												
GANS	4 weeks	16/10/22	15/11/22												
Ofensive & Defensive Applications	2 weeks	1/11/22	16/11/22												
Study of First Model	4 weeks	16/11/22	15/12/22												
State of Art	8 weeks	01/12/22	31/01/23												
Search of Data Sets	4 weeks	01/01/23	31/01/23												
Execution of Code Models:															
Njsscan	8 weeks	01/02/23	31/03/23												
Vuden	10 weeks	16/03/23	31/05/23												
Adapting Vudenc to C/C++	2 weeks	01/06/23	15/06/23												
Training Data	4 weeks	16/06/23	15/07/23												
Experiments	6 weeks	16/07/23	31/08/23												
Memory Writing	24 weeks	01/03/23	31/08/23												

Capítulo 9

Conclusions and Future Work

9.1. Conclusions

Once all the previous tests have been completed and analyzed, and after comparing between them, we can draw out a series of conclusions in relation to what was presented in this study.

In first place, and as has been mentioned during the entire work, the technologies are booming, and due to their fast development, it is causing that both the analysis and the manual detection of vulnerabilities to be ever more complex, leading to the use of other computer applications or new technologies, to be able to solve this type of problems; by making this idea the principle part of the engine in which this study is based.

In this way, the development of the model created with DL has resulted very useful and practical to elaborate this work, since, through the adjustment of different parameters, has made it possible to identify a bigger number of code vulnerabilities than it could be done manually, saving a great quantity of human mistakes.

Furthermore, thanks to the implementation of the colored image according to the degree of vulnerability, the simulation of the test could be more clearly seen and be a very beneficious tool for the programmer, since they can access more quickly thus detecting which are the most problematic areas of the code, with the goal of fixing them and improving the security of the systems.

Moreover, as has been seen in Chapter 5, both the length of the reading window and the size of the fragments of code have an important relation, given that the correct functioning of this model fundamentally depends on both parameters keeping a similar scale. This means, for small files, the window will have to have a reduced size and vice versa, because in any other way, the simulation and the image about the vulnerabilities will come out completely altered and very different from reality.

Lastly, it can be noticed that if all parameters are met, the model works to perfection. As has been seen in the tests carried out in the previous chapter, the simulations of the files that were not part of the training set, were very similar to those that were in the training set. In addition, both the precision and the accuracy of the successes, have been very elevated values, indicating that this tool has not only been achieve but exceeded the looked-for goals, and with the use of some complementary application it can be of great use in the technological world.

9.2. Future work

As the last point of this memory, different alternatives will be presented that could be done in a complementary way to this work. These future works will have the goal of improving the statistics that we have obtained to elaborate a more in depth project and of major level, since, although good results have been generated, the model has its limitations such as problems with the surplus of the memory or the size of the window according to the file being analyzed.

In the first place, there would be a search for a way to increase both the number of files that can be part of the configuration set as well as their size. With this implementation, the model could be more complete and the simulations like the values would come out more adjusted, making the detection of vulnerabilities more precise. In this way, the lines of code that are vulnerable could be better identified, and thus the images clearer. However, to carry out this development, there would be a need to dispose of a very powerful computer equipment which would have to be able to withstand great quantity of information.

One of the other alternatives, and related to the test made previously, have to do with regular size of files and the reading window. With this point, reference is made to the model itself having enough intelligence so it can change the length of the window and adapt it according to the characters that each fragment of code has. Thus, the simulations will be made correctly, but above all, there would be no distinction at the time of training and neither would it have to be modified each time the size of file doesn't match with the window's, facilitating the programmer's work.

Finally, this work will have a final goal, which consists in developing a plugin for a few tools, such as VisualStudio, with which to analyze the code in real life, that is to say, a plugin that warms the programmer if he is writing vulnerable lines of code, with the aim to correct them in the moment and thus avoid future problems in the program.

Bibliografía

[DGC+20] Indira Kalyan Dutta, Bhaskar Ghosh, Albert Carlson, Michael Totaro, and Magdy Bayoumi. Generative adversarial networks in security: A survey. In *2020 11th IEEE Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, pages 0399–0405, 2020.

[FLWN20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[Gra12] Alex Graves. *Long Short-Term Memory*, pages 37–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[GWXW20] Zhibin Guan, Xiaomeng Wang, Wei Xin, and Jiajie Wang. Code property graph-based vulnerability dataset generation for source code detection. In Guangquan Xu, Kaitai Liang, and Chunhua Su, editors, *Frontiers in Cyber Security*, pages 584–591, Singapore, 2020. Springer Singapore.

[Hsu18] Tony Hsiang-Chih Hsu. *Hands-On Security in DevOps: Ensure continuous security, deployment, and delivery with DevSecOps*. Packt Publishing Ltd, 2018.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[KRC+18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[LLY+22] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. A survey of convolutional neural networks: Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, 33(12):6999–7019, 2022.

[LST20] Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, January 2020.

[LZX+18] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, 2018.

[MFBJ21] William Melicher, Clement Fung, Lujo Bauer, and Limin Jia. Towards a lightweight, hybrid approach for detecting dom xss vulnerabilities with machine learning. WWW '21, page 2684–2695, New York, NY, USA, 2021. Association for Computing Machinery.

[NIS17] NIST, 2017.

[NZHZ07] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 529–540, New York, NY, USA, 2007. Association for Computing Machinery.

[RKH+18a] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. pages 757–762, 12 2018.

[RKH+18b] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018.

[Rou18] Lasse Rouhiainen. Inteligencia artificial. *Madrid: Alienta Editorial*, 2018.

[RVPO21] Jesús Estupiñán Ricardo, Maikel Yelandi Leyva Vázquez, Alex Javier Peñafiel Palacios, and Yusef El Assafiri Ojeda. Inteligencia artificial y propiedad intelectual. *Universidad y Sociedad*, 13(S3):362–368, 2021.

[RZBS22] Roshan N. Rajapakse, Mansooreh Zahedi, M. Ali Babar, and Haifeng Shen. Challenges and solutions when adopting devsecops: A systematic review. *Information and Software Technology*, 141:106700, 2022.

[WM21] Michael E Whitman and Herbert J Mattord. *Principles of information security.* Cengage learning, 2021.

[WNV+22] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. Vudenc: Vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology*, 144:106809, 2022.

[WPC+21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[WYT+21] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 16:1943–1958, 2021.

[WZ95] Ronald J. Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. 1995.

[WZD+22] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. Vulcnn: An image-inspired scalable vulnerability detection system. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2365–2376, 2022.

[WZY22] Bolun Wu, Futai Zou, and Xuehu Yan. Code vulnerability detection based on deep sequence and graph models: A survey. *Sec. and Commun. Netw.*, 2022, jan 2022.

[YSHZ19] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures. *Neural Computation*, 31(7):1235–1270, 07 2019.

[ZHL+22] Deqing Zou, Yutao Hu, Wenke Li, Yueming Wu, Haojun Zhao, and Hai Jin. mvulpreter: A multi-granularity vulnerability detection system with interpretations. *IEEE Transactions on Dependable and Secure Computing*, pages 1–12, 2022.

[ZWX+21] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Transactions on Dependable and Secure Computing*, 18(5):2224–2236, 2021.