Desarrollo de modelos de aprendizaje automático para la generación de ataques de seguridad

Development of machine learning models for generating security attacks

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2024–2025

Alumno - Nota Tribunal
David Candil Villacastín - 10
Sergio Lorente Bausela - 10

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, Enero de 2025

Agradecimientos

Nos gustaría expresar nuestra gratitud a todas las personas que nos han ayudado y apoyado a lo largo del tiempo de elaboración y desarrollo del Trabajo de Fin de Grado.

En primer lugar, a nuestros directores Luis Javier García Villalba y Luis Alberto Martínez Hernández, por sus contribuciones y sugerencias que han enriquecido este trabajo y han ayudado a llevarlo a cabo, y al Grupo de Investigación GASS de la Facultad de Informática por las facilidades para la realización del trabajo.

Por otra parte, aunque ajenos al desarrollo directo, agradecemos especialmente a aquellas personas que nos han brindado su apoyo y nos han animado a continuar en largas tardes de arduo trabajo, en especial, a nuestros familiares y a aquellas personas que más cerca tenemos, que nos brindan su cariño y su afecto. Estas personas han sido responsables, en gran medida, de que estemos hoy, finalizando los estudios de grado, agradeciéndoles su apoyo incondicional.

Finalmente, agradecemos a todas esas personas que de forma altruista y anónima aportan fuentes de información y recursos en línea, fomentando el conocimiento de forma libre y accesible a todos los que lo deseen.

De nuevo, a todos ellos, nuestro más sincero agradecimiento.

Índice General

Índice de Figuras

Índice de Tablas

Lista de Acrónimos

AA	*Aprendizaje Automático*
AI	*Artificial Intelligence*
AL	*Automatic Learning*
ANS	*Aprendizaje No Supervisado*
AP	*Aprendizaje Profundo*
AR	*Aprendizaje Por Refuerzo*
AS	*Aprendizaje Supervisado*
AST	*Abstract Syntax Tree*
BERT	*Bidirectional Encoder Representations from Transformers*
BPTT	Back Propagation Through Time
CNN	*Convolutional Neural Network*
CPG	*Code Property Graph*
CSV	Comma-separated Values
CVE	*Common Vulnerabilities Exposure*
CWE	*Common Weakness Enumeration*
DAST	*Dynamic Application Security Testing*
DL	*Deep Learning*

GAI *Generative Artificial Intelligence*

GAN *Generative Adversarial Network*

GPT *Generative Pre-trained Transformer*

GRU *Gated Recurrent Units*

IA *Inteligencia Artificial*

IAG *Inteligencia Artificial Generativa*

LLM *Large Language Model*

LSTM *Long Short Term Memory*

MIT *Massachusetts Institute of Technology*

ML *Machine Learning*

MLM *Masked Language Modeling*

NLP *Natural Language Processing*

OOV *Out Of Vocabulary*

RF *Random Forest*

RNN *Recurrent Neural Networks*

SAST *Static Application Security Testing*

SVM Support Vector Machine

T5 *Text-to-Text Transfer Transformer*

TPR *True Positives Rate*

Resumen

Este trabajo de fin de grado se enfoca en la exploración de diferentes técnicas para implementar modelos generativos de muestras de código de programación que presenten vulnerabilidades y que puedan engañar a un sistema que clasifique muestras entre vulnerables y no vulnerables.

Para desarrollar la idea, utilizamos bibliotecas preexistentes en Python, especialmente el API *Keras* de *Tensorflow*, y un conjunto de datos cuidadosamente seleccionado que contiene vulnerabilidades, amén de otra información relevante adicional. El trabajo se llevó a cabo en tres fases: *Investigación*, *Desarrollo* y *Estudio de los resultados*.

Inicialmente, una revisión exhaustiva de la literatura y la exploración de metodologías existentes nos permitió obtener información sobre las técnicas de vanguardia en la detección de vulnerabilidades y en las técnicas de entrenamiento de aprendizaje automático.

Durante la fase de desarrollo, se realizaron rigurosos experimentos y ajustes de parámetros para optimizar el rendimiento de los modelos de la manera más eficiente posible.

Los experimentos del proyecto culminaron en la fase de evaluación, donde los resultados obtenidos demuestran cuál de las técnicas ha proporcionado los resultados más aceptables en la generación de código vulnerable que puede engañar a un discriminador preentrenado para su detección.

Keywords: Análisis de sentimientos, Clasificación, Generación adversaria, Generación automática de código, Modelado Generativo, Machine Learning, RNN

Abstract

This *final degree project* focuses on the exploration of different techniques to implement generative models for producing programming code samples that present vulnerabilities and can deceive a system that classifies samples as vulnerable or non-vulnerable.

To develop the idea, we utilized existing Python libraries, particularly the *TensorFlow Keras API*, and a carefully selected dataset containing vulnerabilities along with additional relevant information. The work was conducted in three phases: *Research*, *Development*, and *Analysis of results*.

Initially, a comprehensive literature review and exploration of existing methodologies provided insights into the cutting-edge techniques in vulnerability detection and machine learning training methods.

During the development phase, rigorous experiments and parameter adjustments were conducted to optimize the models' performance as efficiently as possible.

The project's experiments culminated in the evaluation phase, where the results demonstrate which of the techniques provided the most acceptable outcomes in the generation of vulnerable code that can deceive a pretrained discriminator for detection.

Keywords: Adversarial Generation, Automatic code generation, Classification, Generative Modeling, Machine Learning, RNN, Sentiment Analysis

Capítulo 1

Introducción

1.1. Motivación

A día de hoy, no podemos obviar que la *Inteligencia Artificial* (IA) es uno de los campos emergentes en la tecnología que mayor impacto va a producir en la sociedad en los próximos años, y que de hecho, ya está produciendo. El rápido avance de la tecnología permite crear herramientas que se creían imposibles en su momento, o que siquiera se concebían. En especial, el avance de los sistemas informáticos y todo lo relacionado con estos no cesa y cada día se reinventan los límites de lo que estos son capaces.

El *Machine Learning* (ML) o *Aprendizaje Automático* (AA) aparece en el contexto de la sociedad de la información, una era en la que la comunicación y la digitalización han transformado todo lo que conocemos. Esta información es, pues, la pieza central de nuestras vidas; la digitalización provoca y permite que prácticamente todo pueda ser almacenado e interpretado, comprensible para una máquina; es aquí donde aparece la IA en juego. Podemos ver cómo estos sistemas en combinación con la información y el tratamiento adecuado pueden aplicarse en todos los campos del conocimiento humano, pudiendo inferir valoraciones en base a estos datos e incluso producir, como por ejemplo, crear imágenes de personas reales, dibujar cuadros al estilo de reputados artistas, manejar vehículos de forma autónoma o producir robots que se comuniquen en nuestro lenguaje natural.

Encontramos a la IA en una variedad inimaginable de usos, y en especial, sin salirse de su ámbito natural, consideramos su papel en la seguridad de los sistemas de la información como uno de los roles clave en su avance al futuro actuando, por ejemplo, como clasificador y detector de malware, o como analizador de imágenes en tiempo real, o también controlando el tráfico de una red. No obstante, la IA no deja de ser un algoritmo que ejecuta órdenes discretas según reciba unos datos de entrada, y por tanto, es susceptible a ataques y vulnerabilidades.

Nuestro objetivo en este trabajo es la investigación y su posterior implementación y uso de diferentes técnicas para la generación de ataques de seguridad en un sistema basándonos en el procesamiento en lenguaje natural de código fuente vulnerable, así como el posterior análisis de los resultados y las conclusiones que obtenemos tras los experimentos.

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).

Se encuentran más datos en:

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

https://lazarus-he.eu

1.3. Objeto de la Investigación

El objetivo de nuestro trabajo es el estudio de los conceptos de ML y *Deep Learning* (DL) y en especial los necesarios para poder entender la *Inteligencia Artificial Generativa* (IAG) y aplicarla a la creación de diferentes modelos de AA para comparar su desempeño en dicha tarea. Tras la investigación de esta materia, diseñaremos varios modelos capaces de generar muestras de código vulnerables que engañen a otro sistema discriminador de ML a modo de línea base, así como analizaremos el rendimiento de los resultados de estos sistemas y las limitaciones que se encuentran en las aproximaciones exploradas para conseguir la tarea.

1.4. Plan de Trabajo

El desarrollo de este trabajo se ha dado en tres fases:

1. **Investigación**: Se comenzó el trabajo realizando una reunión junto al equipo de investigación a mediados de julio de 2023, donde fuimos informados del objetivo del trabajo, los hitos que podíamos seguir en el desarrollo y la competencias que necesitaríamos para llevarlo a cabo. Por conveniencia, acordamos comenzar formalmente el trabajo a partir de septiembre, a partir del momento, establecimos reuniones semanales para comentar las dudas y llevar un seguimiento de la investigación. Se emplearon cuatro meses para la investigación del objetivo del trabajo y de las competencias básicas necesarias para llevarlo a cabo, ya que ambos partíamos de un gran desconocimiento del AA, este desconocimiento fue además un aliciente en la elección del trabajo con el objetivo de aprender sobre este campo de la ingeniería informática. Comenzamos con la lectura de diferentes

artículos relacionados con el estado del arte del AA en la detección y generación de vulnerabilidades de código y las distintas aproximaciones que se toman en trabajos similares relacionados, resumimos estos trabajos para extraer mejor las ideas y tener información en un futuro, dedicamos varias semanas a la búsqueda y extracción de información, tras esto, empezamos a aprender como desarrollar modelos sencillos y tener una toma de contacto con el desarrollo de modelos de aprendizaje automático usando librerías propias, como la librería de Google *TensorFlow*. El avance en la investigación era registrado en un repositorio común donde cada integrante subía los avances de su investigación y estos eran puestos en común posteriormente.

2. **Desarrollo**: El desarrollo de los modelos se estructuró en varias fases intermitentemente, primero se comenzó con la búsqueda y el análisis de datasets válidos que se adecuasen a nuestro objetivo, tras esto, se comenzó con la implementación de diferentes modelos generativos, los cuales fueron refinados poco a poco hasta obtener la mejor combinación, así como un modelo discriminador capaz de predecir si una muestra de código era vulnerable o no a modo de línea base de rendimiento para nuestros generadores. Se probaron diferentes combinaciones y modelos de aprendizaje automático durante el desarrollo para obtener el mejor discriminador. Los primeros modelos desarrollados fueron los generadores de muestras vulnerables basados en la siguiente predicción de una secuencia y, posteriormente, modelos más complejos como los *Large Language Model* (LLM) basados en transformadores, también se exploró y se realizó el desarrolló de una red *Generative Adversarial Network* (GAN), aunque no fuimos capaces de superar unos problemas en la retropropagación de la pérdida de la red del generador, por lo que no pudimos entrenarla completamente y cuantificar su rendimiento, si bien es cierto que gracias a la mayor complejidad de esta técnica en su teoría y desarrollo pudimos aprender en profundidad los secretos que residen en los algoritmos de AA y su desarrollo.

3. **Experimentación**: Finalizada la implementación y entrenamiento de los diferentes modelos y habiendo obtenido prototipos distintos de la fase anterior, se concluyó con el análisis de los resultados obtenidos y las métricas derivadas de este en relación con el discriminador, se continuó con la enumeración de los límites que encontramos en las técnicas aplicadas para la generación de código, finalmente, se sugirieron posibles ampliaciones y continuaciones al trabajo tomando como punto de partida el desarrollo efectuado en este.

1.5. Estructura del Trabajo

El resto del trabajo está organizado en capítulos con la estructura que se comenta a continuación:

El Capítulo 2 introduce conceptos que son elementales para comprender la IA como son el ML, describiendo sus algoritmos y categorización, así como se introduce en mayor profundidad la rama del DL, que es el campo del ML centrado en el desarrollo de redes neuronales, donde se explica en profundidad qué son estas redes, cómo es su funcionamiento, qué tipos de redes encontramos tradicionalmente y qué técnicas existen para optimizar el desempeño de las mismas. Por último, se introducen los conceptos de seguridad en el software y el análisis de vulnerabilidades.

El Capítulo 3 muestra las técnicas de detección y de generación de vulnerabilidades de código. Se presenta un estado del arte citando los trabajos más interesantes sobre el campo, exponiendo las técnicas y conceptos de mayor relevancia actuales.

El Capítulo 4 presenta las contribuciones del proyecto. Se expone de forma lineal el desarrollo que se ha seguido para la implementación de los modelos de DL en la tarea de generación y detección de vulnerabilidades, mostrando la aplicación de técnicas de *Natural Language Processing* (NLP) en código en lenguajes de programación C y C++. Se detallan las características de las técnicas utilizadas, así como el proceso de implementación de las mismas.

El Capítulo 5 describe los resultados obtenidos en la clasificación con el modelo discriminador y, a la vista de estos resultados, las limitaciones que se encuentran en el desarrollo de modelos de esta índole y el procesamiento de datos para generación de código de calidad.

El Capítulo 6 muestra las contribuciones y el trabajo llevado a cabo por los miembros del proyecto, indicando cada miembro sus aportaciones en su apartado.

El Capítulo 7 finaliza con las conclusiones de este trabajo y da una serie de sugerencias de interés para la ampliación o continuación de futuras líneas de investigación basadas en los resultados.

Los Capítulos 8 y 9 son las traducciones al inglés del Capítulo 1: Introducción y del Capítulo 7: Conclusiones.

Capítulo 2

Contexto de la Investigación

En este capítulo se muestra el contexto en el que se basa la investigación de este Trabajo de Fin de Grado. En la Sección 2.1 se exponen los conceptos básicos de ML, continuando con la Sección 2.2 sobre los fundamentos del DL, las redes neuronales y los diferentes tipos existentes, así como se explican técnicas de *Aprendizaje Profundo* (AP). Finalmente, en la Sección 2.4 profundizamos en el análisis de vulnerabilidades de software y los ataques de seguridad sobre este.

2.1. Machine Learning

Machine Learning es la rama de la de IA enfocada en la creación de algoritmos y técnicas por los cuáles mediante el procesamiento de datos, una computadora es capaz de aprender y mejorar en tareas específicas. Las máquinas utilizan estos algoritmos para identificar patrones y reglas en los datos tratando de obtener la distribución estadística real de los mismos, de forma que sean capaces de tomar decisiones autónomas basándose en el aprendizaje tomado.

Puede categorizarse en tres tipos según su aprendizaje:

- *Aprendizaje Supervisado* (AS): Se trata de algoritmos que mediante datos previamente 'etiquetados' pretenden encontrar una función por la cual sean capaces de obtener predicciones correctas para nuevas entradas. Este tipo de aprendizaje es principalmente utilizado en problemas de regresión y de clasificación. Sus algoritmos mas comunes serian los algoritmos de regresión, árboles de decisión, clasificación de Naïve Bayes, las Support Vector Machine (SVM), o las redes neuronales.
- *Aprendizaje No Supervisado* (ANS): Se tratan de algoritmos con un carácter exploratorio, es decir se centran en el estudio de datos no etiquetados, su objetivo es estudiar la estructura de los datos para intentar encontrar relaciones entre los valores y variables de estos, tratando de descubrir las características que comparten determinados conjuntos de estos. La técnica más empleada en este tipo de aprendizaje suele ser el clustering mediante la aplicación de algoritmos como la Descomposición de Valores Singulares o el Análisis de Componentes Principales. También se aplican técnicas y arquitecturas con redes neuronales.
- *Aprendizaje Por Refuerzo* (AR): Se basa en un modelo de retroalimentación, por cual

el sistema recibe como datos de entrada una información y produce una respuesta por la cual es recompensado o castigado en función de esta.

2.2. Deep Learning

El Aprendizaje Profundo o *Deep Learning*, es una rama dentro del ML que se centra en los modelos conocidos como redes neuronales. Las redes neuronales, como su nombre indica, se inspiran en el cerebro humano y su funcionamiento para generar un modelo computacional capaz de aprender patrones y distribuciones de datos complejos en tareas como, por ejemplo, reconocimiento de imágenes o procesamiento del lenguaje natural, entre otros. Estos modelos superan las limitaciones de las técnicas y algoritmos tradicionales en el procesamiento de datos en bruto, aplican transformaciones y obtienen abstracciones sobre los datos a un nivel que es imposible de conseguir aplicando otros métodos.

Existen diversas técnicas y algoritmos de DL sobre el desarrollo y aprendizaje de modelos de redes neuronales.

2.2.1. Redes neuronales

Las redes neuronales son la pieza fundamental sobre la que se cimienta el DL. De modo sencillo, se componen por neuronas, llamadas perceptrones, las cuales se agrupan por capas y aplican una transformación en base a unos datos de entrada, introduciendo una no linearidad en su salida para iterativamente tratar de obtener la distribución de los datos de salida correctos.

El funcionamiento de una neurona básica es sencillo. En la Figura 2.1 podemos ver su estructura. Recibe múltiples entradas donde cada entrada representa una característica del dato que se está procesando. Sobre cada entrada asigna un peso, que representa la importancia sobre la decisión final de la salida, estos pesos se ajustan automáticamente durante el proceso de aprendizaje para adaptarse a los datos. A continuación, se aplica una suma ponderada de las entradas, multiplicando cada entrada por su peso correspondiente y aplicando un sesgo final adicional que permite flexibilizar el resultado. Finalmente, al resultado se le suele aplicar o no una función de activación que regulariza los logits de salida según su actuación, así obtenemos la salida de la red neuronal.

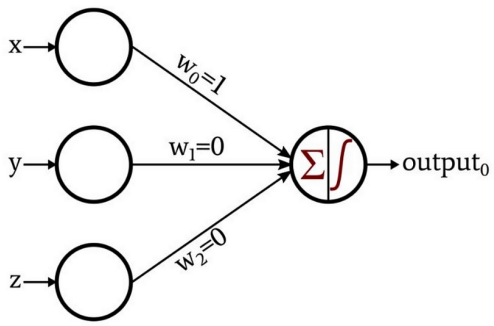

Figura 2.1: Perceptrón básico

Sin embargo, los perceptrones no pueden resolver problemas con una representación no lineal, así como no puede mantener una relación de complejidad suficiente entre los datos de entrada, es así como aparecen los perceptrones multicapa. Un perceptrón multicapa

no es más que una capa compuesta de perceptrones simples, en la Figura 2.2 podemos ver la estructura básica de un perceptrón multicapa completamente conectado entre sus neuronas. Por tanto, una red neuronal básica, en esencia, esta compuesta por una entrada, una serie de capas ocultas de perceptrones por las que pasa dicha entrada y una capa de salida que da el resultado final de los cálculos.

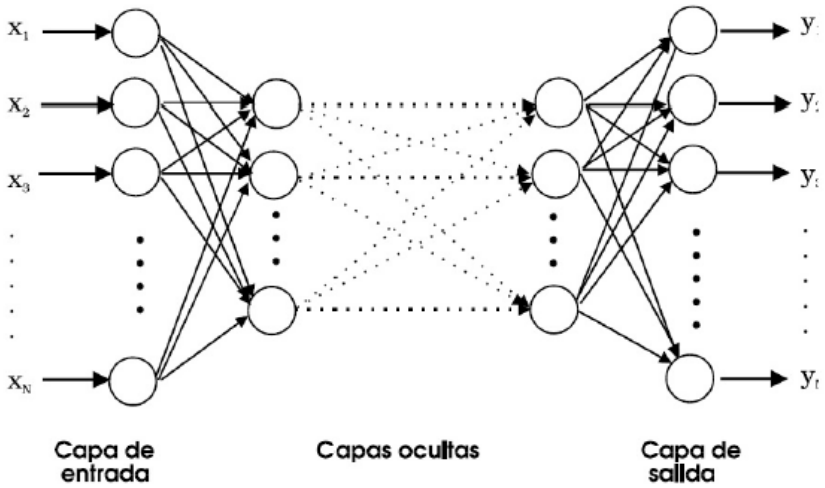

Figura 2.2: Perceptrón multicapa

Por último, para que una red neuronal sea capaz de ajustarse a los datos de entrada, debe realizar un proceso de entrenamiento. El entrenamiento es el proceso por el cual una red neuronal ajusta sus pesos y sesgos internos para realizar una tarea, como la clasificación, regresión o generación. Aplicando un proceso iterativo, la red neuronal se ajusta a los datos para mejorar en la tarea que se desee, actualizando los pesos en consecuencia para generar mejores salidas. Simplificando el proceso, este se compone de los siguientes pasos:

1. **Inicialización**: La red asigna de forma aleatoria los pesos y sesgos internos.
2. **Propagación hacia delante**: Conocida como *Forward Propagation*. Los datos atraviesan la red, capa por capa, hasta producir una salida final.
3. **Cálculo de la función de pérdida**: Para el caso del aprendizaje supervisado, esta función se encarga de medir la diferencia entre las predicciones de la red y los datos reales esperados. Mide por tanto, el error de la red.
4. **Propagación hacia atrás**: Conocida como *Backpropagation*. A partir del error calculado anteriormente, se cálcula el gradiente de la función de pérdida sobre cada peso de la red, tratando de minimizar el error.

Existen muchos tipos de redes neuronales según su estructura y forma de tratamiento de los datos de entrada. Cada tipo de red, debido a estas diferencias, obtiene mejores resultados en el procesamiento de algunos tipos de datos y tareas. Algunas de sus estructuras más conocidas son las siguientes:

- *Recurrent Neural Networks* (RNN): Las redes neuronales recurrentes son un tipo especial de redes especialmente diseñadas para procesar datos secuenciales, es

decir, secuencias ordenadas o que mantienen algún orden interno que pueda ser de relevancia para el contexto de la red. Este tipo de red mantiene el contexto durante el procesamiento de la secuencia de entrada y se ve influenciada en sus decisiones por este, manteniendo dependencias a lo largo del tiempo.

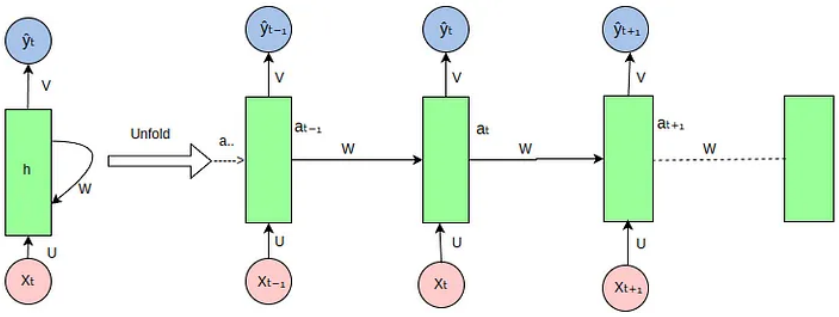

Figura 2.3: Estructura de RNN [Min25]

En la Figura 2.3 podemos ver la estructura con la que se asocian las neuronas dentro de una capa de la red. El secreto de las redes neuronales son estas conexiones recurrentes entre neuronas que permiten que la salida de una neurona influya en la entrada de la siguiente en la secuencia, he ahí su interés por los datos y procesamiento secuencial, este bucle de retroalimentación se conoce como .estado oculto", en cada paso del entrenamiento, este estado oculto se actualiza en función de la entrada y del estado oculto del paso anterior, pudiendo mantener ese recuerdo de la secuencia. La información en las RNN se propaga a través del tiempo en un algoritmo llamado Back Propagation Through Time (BPTT), que a diferencia del enfoque tradicional de retropropagación, este suma los errores en cada paso del tiempo, mientras que el algoritmo tradicional lo suma al final ya que las capas de las redes neuronales tradicionales no comparten parámetros en cada capa.

Sin embargo, las RNN presentan dos problemas a considerar:

1. **Desvanecimiento y explosión de gradiente:** Debido a la BPTT, es un problema inherente a estas redes, el gradiente es la pendiente de la función de pérdida sobre el error, si el gradiente es demasiado pequeño continuará haciéndose más pequeño, hasta desvanecerse a 0, del mismo modo pero al contrario ocurre con los gradientes grandes, que cada vez se hacen más grandes produciendo una explosión de gradiente. Estos problemas pueden paliarse eliminando complejidad de la red, como puede ser, por ejemplo, reduciendo sus capas ocultas, aunque este no es el objetivo.
2. **Memoria a largo plazo:** Generalmente, las RNN presentan problemas para mantener el contexto en secuencias de larga longitud debido a su funcionamiento y su susceptibilidad al problema de gradientes
3. **Procesamiento secuencial:** Debido a su estructura, al procesar los datos de forma secuencial se limita su capacidad y eficiencia sobre gran cantidad de estos, siendo más lentas y costosas computacionalmente.

No obstante, existen variantes de las RNN que tratan de superar algunas de las limitaciones que esta presenta, estas variantes de redes han demostrado ser más efectivas sobre tareas complejas:

- *Long Short Term Memory* (LSTM): Las redes LSTM se teorizan en un principio con el objetivo de superar los problemas de desvanecimiento de gradiente que presentan las RNN tradicionales, se componen de celdas de memoria como la que se puede ver en imagen de en medio de la figura 2.4, estas celdas se componen por una puerta de entrada, que controla que información se almacenará en la celda, una puerta de salida, que decide que información saldrá de esta, y una puerta de olvido, que trata que información debe eliminarse. Las LSTM superan a las RNN en rendimiento gracias a sus ventajas, aunque son mas costosas computacionalmente al componerse de más puertas y siguen siendo susceptibles a problemas de explosión de gradiente, aunque superan el desvanecimiento.
- *Gated Recurrent Units* (GRU): Esta red es un punto intermedio entre las RNN tradicionales y las LSTM por su rendimiento y eficiencia computacional gracias a componerse de menos puertas en cada neurona. Se compone de celdas con dos puertas las cuales son: una puerta de actualización (*update gate*) que controla cuanta información pasa de la celda y controla cuanto se actualiza el estado oculto y una puerta de reinicio (*reset gate*) que tiene como entrada el estado oculto y la actual entrada y determina cuanta parte del estado oculto previo se olvida en el paso. Las GRU también son resistentes a los problemas de desvanecimiento de gradiente.

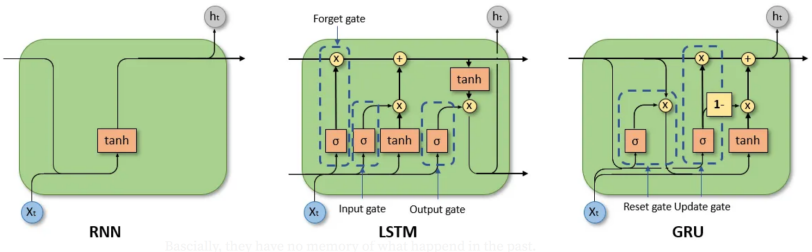

Figura 2.4: RNN y variantes

- *Convolutional Neural Network* (CNN): Las redes neuronales convolucionales están diseñadas especialmente para tratar datos matriciales, como su nombre indica realizan operaciones convolucionales aplicando filtros sobre la entrada que producen una salida en la capa siguiente por medio de una operación de pooling, tras aplicar el proceso sobre la entrada y pasar por las diversas capas de la red, se extraen características que pasan a través de capas completamente conectadas (*fully connected layers*) al igual que en las redes neuronales tradicionales y que finalmente se combinan para dar una predicción. Este tipo de redes neuronales es usado principalmente para el procesamiento de imágenes del ámbito que sea, el secreto de estas redes reside en la cantidad de parámetros por los que se compone y la forma en la que trata los datos, en un caso general, una imagen se compone de millones de pixeles que la conforman, si tratásemos de abordar su procesamiento de la forma habitual, la red debería establecer una basta cantidad de parámetros en su entrada y estos a su vez conectarían con más capas aún, generando un problema exponencial con un pobre resultado, para procesar este tipo de datos, sin embargo, gracias a las operaciones de convolución y la forma en la que se procesa, las CNN son perfectas para los datos matriciales, como son las imágenes.

2.3. Transformadores

Los transformadores o *Transformers* son redes neuronales especializadas en el aprendizaje de datos secuenciales y la generación de datos según estos. Es un concepto revolucionario en el NLP que aparece por primera vez en el año 2017 en *Attention is all you need* [AV17] desarrollado por Google. Toman su nombre por la tarea que realizan; dada una secuencia de entrada, transforman esta en una nueva secuencia de salida.

Se cimientan sobre la arquitectura *encoder-decoder*, por su traducción, codificador-decodificador. La estructura básica interna de un transformador se compone por varias capas codificador y decodificador, ambas con el mismo número total de elementos en cada lado, con la siguiente forma:

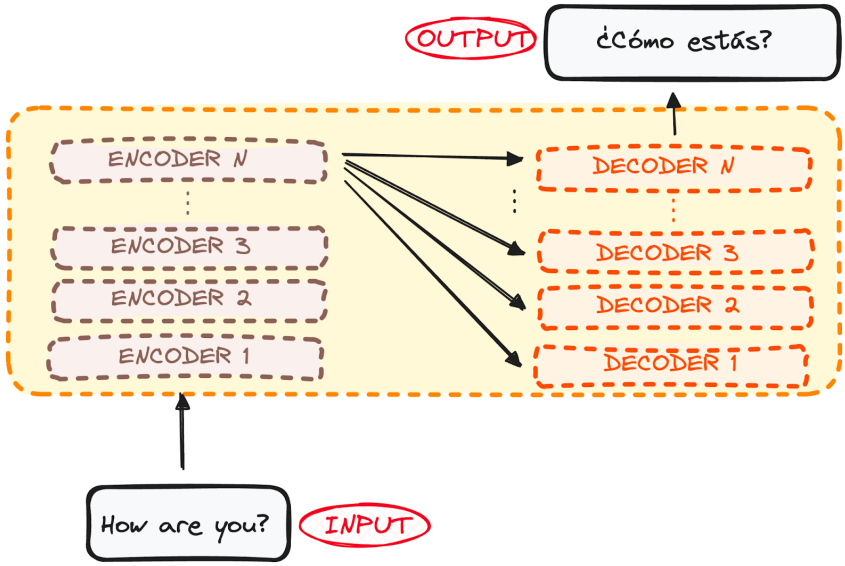

Figura 2.5: Estructura básica de un transformador [Fer24]

En la figura anterior podemos ver un ejemplo de un transformador como traductor de oraciones de un idioma a otro. Esta idea de los transformadores puede aplicarse en gran variedad de datos y tareas, dando resultados impresionantes. El secreto de los transformadores es su mecanismo de atención *Self-Attention*, este mecanismo es parte de cada codificador; podemos ver la estructura de uno en la siguiente figura:

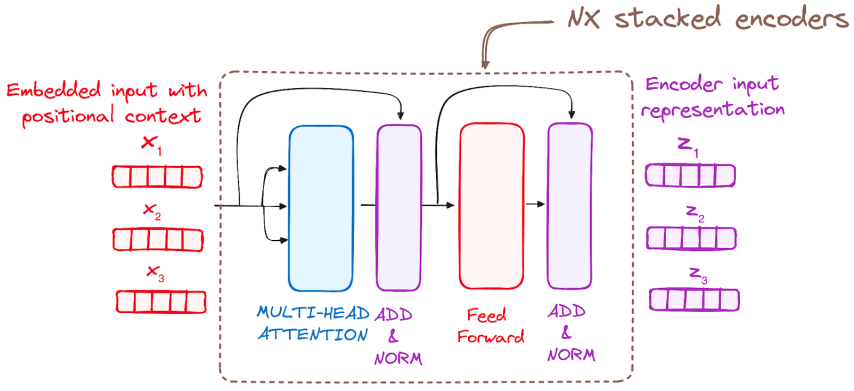

Figura 2.6: Funcionamiento interno de un codificador [Fer24]

Un codificador es capaz de obtener el contexto de cada token respecto a la secuencia total de entrada. El primer paso es transformar los tokens de entrada en vectores de longitud fija, esto es conocido como *embedding*. El primer codificador recibirá como entrada los vectores de los tokens, los siguientes codificadores recibirán la salida del codificador anterior. Tras esto, se añade un vector de codificación posicional que permite contextualizar el token por su posición.

A continuación, se procede a aplicar el llamado *Multi-Headed Self-Attention Mechanism*, este es un mecanismo que extiende el mecanismo de *Self-Attention* estándar en varias cabezas o *heads* que realizan la misma operación sobre la secuencia y se combina finalmente para obtener el resultado, mejorando el rendimiento. Tras esto continúan una serie de operaciones de normalización junto a una red neuronal lineal *feed-forward* que dan el resultado final del codificador. La salida de cada codificador se toma como entrada en el siguiente.

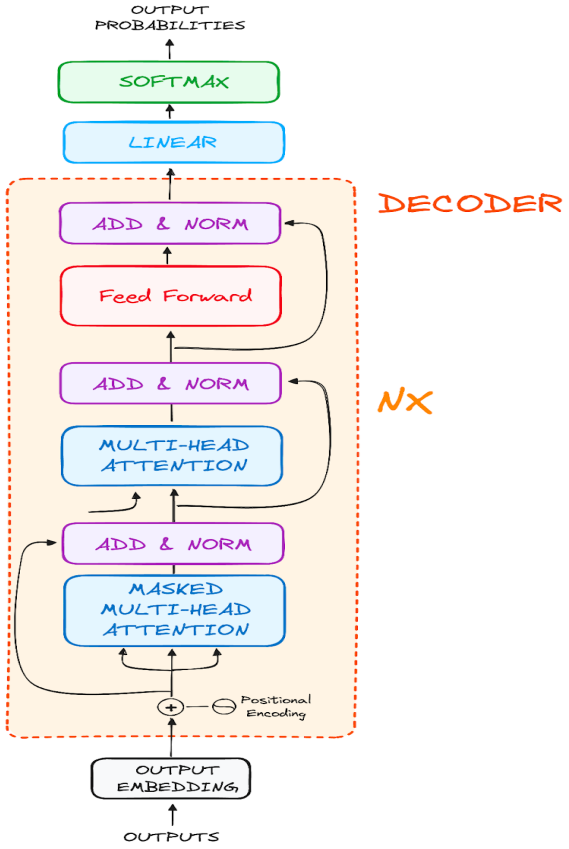

Figura 2.7: Funcionamiento interno de un decodificador [Fer24]

Por otra parte, los decodificadores actúan como espejo de los codificadores, la salida producida por los codificadores se incrusta en un vector de longitud fija y se combina con vectores de codificación posicional, se continúa aplicando un *Masked Multi-Headed Self-Attention Mechanism*, que difiere en un concepto esencial, el enmascaramiento, este se da para evitar que las predicciones dadas para un token en particular no se vean afectadas por tokens futuros y solo dependan de los tokens anteriores. Posteriormente, tras una normalización, otra capa de Self-Attention es aplicada con la salida del codificador y la salida de la anterior capa de atención se combinan y siguen el mismo proceso que en el codificador. Finalmente, la salida obtenida en el proceso atraviesa una red lineal del

tamaño del vocabulario de salida y se refina aplicando una función *softmax*, obteniendo las probabilidades para el siguiente token en la secuencia de salida.

Actualmente, los transformadores son el estado del arte del NLP. Presentan muchas ventajas en diferentes aspectos, superan a las redes neuronales de la RNN en la escalabilidad del aprendizaje con grandes conjuntos de datos, la obtención del contexto y dependencias entre los tokens de secuencias de entrada y, esencialmente, la paralelización en el proceso, la cual es imposible en las redes recurrentes al ser secuenciales. Los transformadores son además el corazón de modelos LLM como *Generative Pre-trained Transformer* (GPT) o *Bidirectional Encoder Representations from Transformers* (BERT).

2.3.1. BERT

BERT es una implementación de la arquitectura de transformadores desarrollado por Google, diseñada para la comprensión del lenguaje natural teniendo en cuenta tanto el contexto de las palabras a la izquierda de la procesada, como el contexto a la derecha de la misma, permitiendo comprender mejor las relaciones semánticas y sintácticas dentro de las oraciones.

Este modelo a diferencia de otros cuenta con un entrenamiento bidireccional, es decir, es capaz de analizar el contexto completo de cada palabra de forma simultánea. Su proceso de aprendizaje se realiza por medio de un entrenamiento con enmascaramiento *Masked Language Modeling* (MLM), en el cual se reemplazan una cantidad aleatoria de palabras de la oración por un token especial de máscara mask y el modelo, considerando las palabras anteriores y posteriores al token, debe dar una predicción del token enmascarado.

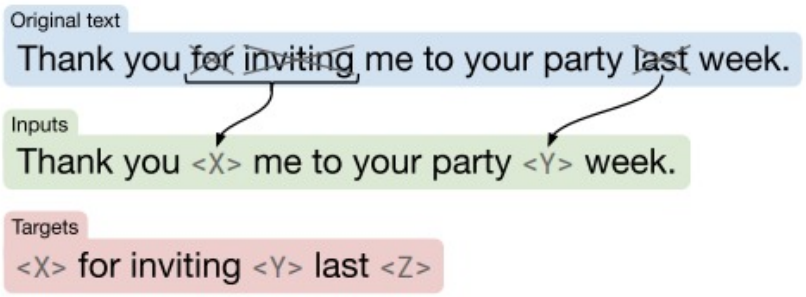

Figura 2.8: Método de entrenamiento C4

2.3.2. GPT

GPT son un conjunto de modelos con estructuras basadas en transformadores desarrolladas por OpenAI que han supuesto una revolución en el panorama NLP en los últimos años debido a su capacidad para comprender y generar texto coherente y contextual.

A diferencia de la mayoría de los modelos basados en transformadores, estos se construyen únicamente sobre la parte del decodificador, es decir, están optimizados para la para tareas generativas, prediciendo las siguientes palabras basándose en las palabras previas, permitiéndolo generar un texto fluido y continuo.

GPT es también, un ejemplo de LLM, los cuales son modelos de lenguaje entrenados sobre cantidades masivas de datos de texto con la capacidad de comprender, contextualizar y generar datos en consecuencia. Por ejemplo, GPT-3 es un modelo de alta complejidad y sus resultados lo avalan, se compone de 175 mil millones de parámetros y fue entrenado sobre un conjunto de más de 500 Gb de puro texto. Aplica un aprendizaje no supervisado sobre los datos para obtener patrones y enfocar su atención en los elementos clave del contexto. Tras su preentrenamiento, el modelo puede ajustarse a diferentes tareas específicas gracias a un proceso denominado *fine-tuning*.

2.4. Seguridad en el Desarrollo Software

La seguridad es siempre uno de los factores más importantes en el desarrollo de cualquier sistema. En el software, la seguridad en el programador, que decide qué prácticas tomar y aplicar y qué considera a su juicio como vulnerable o no. Sin embargo, como es habitual, el criterio humano en ocasiones no es suficiente y no únicamente esto, sino la posibilidad de error a pesar de todo, es por ello que aparecen herramientas de análisis de código auxiliares que permiten encontrar vulnerabilidades sobre este. Son habituales los siguientes análisis de vulnerabilidades de código:

- *Static Application Security Testing* (SAST): El análisis de de código estático busca vulnerabilidades en el propio código fuente, tratando de encontrar patrones, palabras o estructuras de código, como bucles o condicionales, que puedan ser susceptibles de una vulnerabilidad en su ejecución. Es más rápido que el análisis dinámico, aunque debido a su arbitrariedad provocada por su funcionamiento en su búsqueda de resultados, puede producir falsos positivos.

  Este análisis puede darse sobre el código fuente original o sobre el código compilado, analizando patrones de bytes que puedan corresponder a la ejecución de código vulnerable. Al no ejecutar código, no es capaz de encontrar vulnerabilidades más allá de lo que analiza en el momento, dejando sin considerar las posibles complicaciones de la ejecución de distintas ordenes u algoritmos en el programa, de ello se encarga su contraparte, el análisis dinámico.

- *Dynamic Application Security Testing* (DAST): El análisis de código dinámico, busca vulnerabilidades en tiempo de ejecución o compilación del programa, evaluando el desempeño de la aplicación. Realiza, por ejemplo, explotación de vulnerabilidades SQL, comprobación de desbordamientos de números, pruebas de inyección de datos aleatorios, etc. Por lo general, se realiza en entornos de pruebas controlados, trata de explotar comportamientos del programa ante diferentes entradas, buscando las vulnerabilidades que se producen ante diferentes situaciones. Por ejemplo, se prueban las entradas y la validación de los datos, tratando de prevenir inyección de código SQL/XSS, fallos de autorización o de redirección, etc.

  Ambos tipos de análisis no son excluyentes, sino complementarios, siendo la combinación de estos la mejor estrategia para analizar la seguridad de aplicaciones software.

Capítulo 3

Estado del Arte

En la actualidad, existen diversos trabajos sobre que indagan en las capacidades de detección de ataques de seguridad, es decir, problemas de clasificación de datos; no obstante, no se explora con tanto detalle la generación, que es un problema de regresión. Menos frecuente aún es ver estos conceptos aplicados a la detección y generación de vulnerabilidades software por medio del NLP.

En las siguientes secciones, trataremos de dar un vistazo general sobre el estado del arte actual respecto a los dos papeles que se dan en un ataque de seguridad, atacante y víctima. Es por tanto que la Sección 3.1 trata el papel de la generación y los trabajos que tratan de producir nuevos datos, mientras que la Sección 3.2 expondrá los trabajos y conceptos más relevantes actualmente en la detección de vulnerabilidades.

3.1. Generación en sistemas IA

La generación de código es una tarea que puede abordarse de muchas formas, ya sea tanto por la naturaleza del problema como por los resultados que se buscan producir, encontramos variedad de trabajos que realizan tareas con diversos objetivos. En [DDH+22] los autores analizan 37 publicaciones distintas relacionadas con la generación de código; se analiza la tarea que realizan según el objetivo que siguen, los datos que se utilizan, los modelos que se aplican, así como se discuten temas respecto a la calidad de los datos, la tokenización, el uso de los modelos e incluso los métodos de evaluación del rendimiento.

En el artículo, se aborda el análisis resumido de trabajos con diferentes objetivos, cada uno persigue una tarea diferente, ya sea realizar o generar tests para clases de código, traducir de un lenguaje a otro o generar documentación de funciones. Podemos clasificar los problemas según el procesamiento de su entrada y salida en:

- **Descripción a código**: En este paradigma, la descripción de la que se pretende obtener código puede venir de varias formas, la más habitual es en lenguaje natural, como se realiza en [AIZ19] donde se trata de generar código por medio de descripciones o en [SABP19] en el que se trata por medio de un árbol *Abstract Syntax Tree* (AST) de lenguaje natural la generación de código correspondiente. También se toman como descripciones ejemplos de código en diferentes variantes o incluso imágenes, en la figura 3.1 podemos ver un ejemplo donde se toma el aspecto de elementos HTML y se busca obtener el código que los produce.

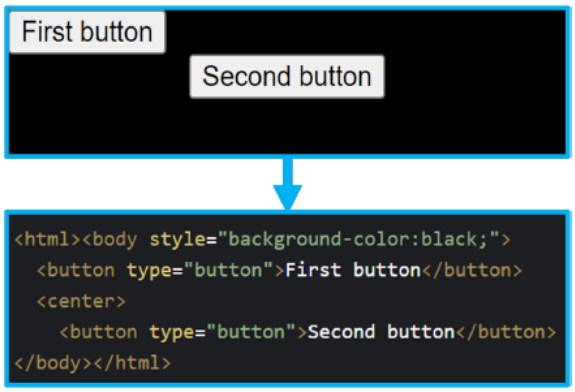

Figura 3.1: Problemas de descripción a código

- **Código a descripción**: Se trata del problema inverso al anterior, donde se da un fragmento de código y debe obtenerse su descripción, se busca obtener una definición en lenguaje natural para el código proporcionado, ya sea como comentarios del programa o como documentación aclaratoria.
- **Código a código**: Tratan problemas de generación o completación de código en base a otro código, también pueden ser problemas de traducción de código en un lenguaje a otro, como se realiza en [HMM+20] donde los autores tratan de traducir código de aplicaciones móviles en lenguaje Swift a Java.

3.1.1. Redes generativas adversarias

Las *Generative Adversarial Networks* son un concepto esencial en el plano de la IA generativa introducidas por el matemático Ian Goodfellow en el año 2014, en su trabajo [IJG14], muestran un concepto revolucionario, un modelo formado por dos redes neuronales, un generador y un discriminador de muestras que conforman un juego de suma cero. Un juego de suma cero es un concepto de teoría de juegos donde el beneficio de un jugador es exactamente igual a la pérdida de otro, esto significa que el objetivo de una GAN es la convergencia de ambas redes neuronales hasta que ambas no puedan mejorar más su rendimiento, dando lugar a esta situación.

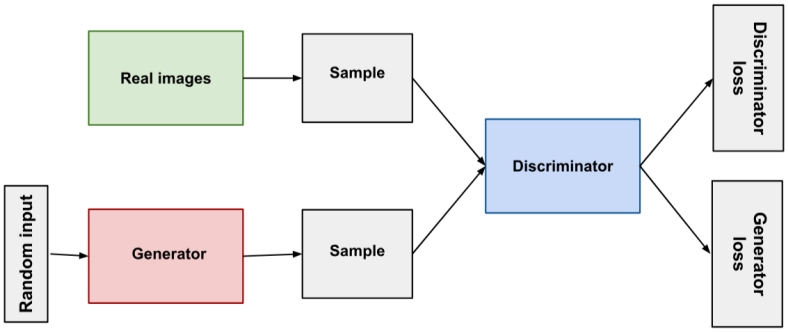

Figura 3.2: Estructura de una GAN

Su estructura se conforma como se muestra en la figura 3.2, disponemos de un generador que produce muestras en base a un vector de ruido aleatorio, llamada la dimensión

latente, y un discriminador que recibe estas muestras junto a otras muestras reales y da la probabilidad de que estas sean reales o falsas, es decir, provenientes del generador. El generador y discriminador se optimizarán según la pérdida de los resultados predichos, el proceso continúa iteración tras iteración y converge de forma ideal en el punto en que el generador crea muestras tan cercanas a las reales que el discriminador no puede diferenciar si estas son reales o no, dando lugar a la llamada "suma cero".

El discriminador trata de maximizar la probabilidad de asignar correctamente una muestra, mientras que el generador busca minimizar la probabilidad de que sus muestras sean identificadas, tratando de imitar a las reales. Este concepto se expresa de forma matemática en la siguiente fórmula:

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z\sim p_z(z)} [\log(1 - D(G(z)))]$$

- $G(z)$: Es el dato generado por el generador a partir de un vector aleatorio z (normalmente extraído de una distribución)
- $D(x)$: Es la probabilidad de que el discriminador clasifique correctamente un dato x
- $p_{\text{data}}(x)$: Es la distribución de los datos reales
- $p_z(z)$: Es la distribución del espacio latente (el ruido de entrada al generador para producir muestras)

Las GAN son aplicadas en muchos ámbitos, abundan los trabajos relacionados con la generación de imágenes por medio de estas redes, no obstante, aunque no encontramos información de trabajos con GAN en los que se realice la generación de código en lenguaje natural de forma explícita, sí que existen trabajos que han explorado sus posibilidades en la generación de ataques de seguridad aplicados a la detección de malware, en [HT17] los autores proponen *MalGAN*, donde tratan de entrenar un generador de malware y prueban sus resultados frente a diferentes algoritmos de detección, como son *Random Forest*, máquinas SVM o *regresión logística*. Realizan un entrenamiento de caja negra o *blackbox*, es decir, el atacante no conoce nada más que las predicciones que produce el modelo, esto suele ser la estrategia más habitual en las redes GAN, aunque de forma parcial, ya que en una GAN tradicional, el generador tiene también acceso al gradiente y función de pérdida que se produce y el discriminador evoluciona en paralelo junto a él. Por último, los autores constatan que logran un engaño prácticamente total sobre los algoritmos anteriores, obteniendo una tasa *True Positives Rate* (TPR) de casi cero en todos los casos; finalmente, se comprueba el rendimiento de la red *MalGAN* reentrenando los algoritmos para reforzarlos contra los ataques adversarios y se vuelven a realizar los experimentos, obteniendo los mismos resultados y constatando la vulnerabilidad de los algoritmos de AP frente a los ataques adversarios y su capacidad de mimetización de los datos.

3.2. Detección en sistemas IA

La detección automática de vulnerabilidades ha sido una rama muy estudiada desde la década de los 70, aunque nos centraremos en las últimas técnicas desarrolladas. El primer

estudio moderno con relevancia sería el estudio del año 2014 [YGAR14]. Este estudio nos acerca al concepto de la detección desde un punto no enfocado en DL sino en uno basado en la representación de código mediante un CPG, es decir, un grafo que representa tanto el código y su comportamiento, como los datos que se utilizan, de modo que, combinado con un conjunto de reglas estáticas, sea capaz de reconocer patrones.

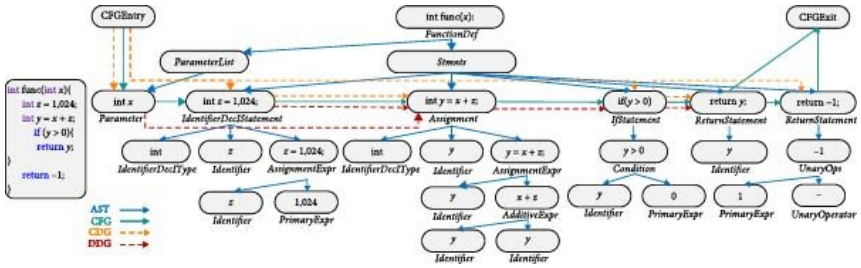

Figura 3.3: Ejemplo CPG

La metodología demostró ser efectiva para encontrar vulnerabilidades con patrones similares a los del dataset, aunque encontró limitaciones al tratarse de un enfoque estático, como sería la poca flexibilidad de encontrar vulnerabilidades si estas no han sido explícitamente incluidas en las reglas.

Por su parte la detección automática de patrones mediante sistemas IA aparece de forma posterior y presenta muchas y muy variadas técnicas que intentan ser capaces de resolver estos problemas de una forma más dinámica y flexible. Para nuestro estudio, nos centraremos en aquellas técnicas y estudios enfocados en la detección de vulnerabilidades en código fuente.

En el contexto de detección de vulnerabilidades, no se presenta un trabajo con verdadera relevancia hasta el año 2018, con el trabajo [SFHC24], donde los autores plantearon distintos modelos de forma que pudieses averiguar cuál proporcionaba mejores resultados.

Tras pruebas con modelos CNN, RNN, *Random Forest* (RF) además de combinaciones de los mismos, determinaron que, aunque todos ofrecían unos resultados suficientemente robustos para la detección de vulnerabilidades, el modelo basado en la combinación de CNN y RF ofreció el mejor resultado para conseguir el equilibrio más óptimo entre precisión y recall.

	PR AUC	ROC AUC	MCC	F_1
BOW + RF	0.459	0.883	0.462	0.498
RNN	0.465	0.896	0.501	0.532
CNN	0.467	0.897	0.509	0.540
RNN + RF	0.498	0.899	0.523	0.552
CNN + RF	**0.518**	**0.904**	**0.536**	**0.566**

Figura 3.4: Resultados detección de vulnerabilidades

Capítulo 4

Metodología del trabajo

En este capítulo se expone el proceso tomado para el desarrollo de los objetivos y modelos en el trabajo, así como se muestran en detalle los obstáculos encontrados y decisiones tomadas sobre estos.

En el proyecto se realizó la implementación de diversos modelos de IAG, así como se desarrolló un modelo discriminador capaz de clasificar muestras en vulnerables y no vulnerables para tener una línea base en el rendimiento obtenido por estos. Nuestro objetivo es que los generadores sean capaces de producir muestras de código vulnerable por medio de técnicas NLP y que el discriminador actúe como un sistema de detección de código vulnerable, también aplicando técnicas de NLP, que dictaminará si una muestra de código proporcionada es vulnerable o no. Trataremos de engañar al discriminador con nuestras muestras generadas porque eso significará que hemos producido un ataque de seguridad en el sistema.

El desarrollo del proyecto completo fue realizado en Jupyter Notebooks haciendo uso de Python 3.10.14, el API Keras de Tensorflow 2.10.1 y Nvidia CUDA 12.4 sobre una máquina Intel 13600-KF, Nvidia 4070 RTX y 32 GB de memoria RAM. Es importante matizar las versiones anteriores ya que no existe retrocompatibilidad entre algunas librerías en su última versión, en especial, en la ejecución de código en GPU con *Tensorflow*, lo cual puede incrementar exponencialmente la ejecución de los entrenamientos, que ya son de por sí costosos. Los cuadernos están diseñados para ser ejecutados completamente sin inconvenientes y están enumerados para su ejecución; no obstante, se proporcionan los modelos entrenados para eliminar el tiempo requerido para el entrenamiento y facilitar la ejecución.

En la Sección 4.1 se explica la elección y análisis del dataset. En la siguiente sección, se detalla el desarrollo del modelo discriminador y todo lo inherente a este. La Sección 4.2 trata en cada punto el desarrollo de cada modelo generativo, la idea estructural que cimienta cada uno y se detallan el procesamiento de datos y elección de los modelos en cuestión. Por último, en la Sección 4.3 se muestra la implementación del modelo discriminador de muestras que utilizaremos para cuantificar los resultados.

4.1. Estudio, búsqueda de un dataset

El primer paso para abordar cualquier tipo de desarrollo de aprendizaje automático es obtener datos de una calidad y cantidad suficientes para realizar el entrenamiento de

los modelos de una forma óptima. Los datos son la pieza fundamental de cualquier tipo de aprendizaje, tanto es así que estos son los protagonistas en el auge del *Aprendizaje Automático*, propiciados por la era de la información.

Para nuestro proyecto, buscaremos datasets de recopilaciones de fragmentos de código vulnerables y no vulnerables; necesitaremos que nuestros generadores puedan aprender la distribución de los datos vulnerables, mientras que nuestro discriminador sea capaz de diferenciar entre aquellas muestras que sí o no lo son. Encontramos que la cantidad de datasets disponibles al público para esta tarea se reduce a unos pocos, y en su mayoría, estos datasets se componen de recopilaciones de código C/C++, lenguaje en el cual, por su bajo nivel, se producen vulnerabilidades habitualmente.

La creación de un dataset sobre código es una ardua tarea puesto que se requiere de un análisis individual de cada muestra y su contexto para confirmar que esta es vulnerable, estos análisis pueden realizarse por medios como se detallan en la Sección 2.4 del Capítulo 2. Los análisis dan un veredicto sobre la muestra ya sea de forma estática, es decir, buscando patrones de código vulnerable, o de forma dinámica, ejecutando el código con diferentes situaciones y parámetros para ver su comportamiento. Sin embargo, estos análisis no son suficientes para dictaminar si una muestra es vulnerable o no y se requiere en la mayoría de casos el juicio de un experto para clasificar las muestras, es por ello que no hay una variedad de datasets sin una calidad cuestionable, ya que requieren una obtención y clasificación manual de los datos en gran parte de los casos.

Encontramos el trabajo de Grahn, D. y Zhang J. [GZ21] sobre el análisis de datasets de vulnerabilidades de código C/C++ una exposición en profundidad de los conjuntos de datos en el que se aborda la forma en la que se generaron, su clasificación, su composición y formato, y un análisis de su diversidad.

Dataset	License	Granularity	Compiles?	Cases	# of Vulns
Big-Vul	MIT	Functions	✗	348 Projects	3,754
Draper VDISC	CC-BY 4.0	Functions	✗	1.27M Functions	87,804
IntroClass	BSD	Scripts	✓	6 Assignments	998
Juliet 1.3	CC0 1.0	Scripts	✓	64,099 Cases	64,099
ManyBugs	BSD	Projects	✓	5.9M Lines	185
SVCP4C	GPLv3	Files	✗	2378 Files	9,983
Taxonomy...	MIT	Scripts	✓	1,164 Cases	873
Wild C/C++	CC-BY 4.0	Files	✗	12.1M Files	Unknown

Figura 4.1: Características sobre los datasets existentes [GZ21]

En la tabla de la figura 4.1 podemos ver algunas características generales de los datasets que existen actualmente. Nos centramos especialmente en los datasets *BigVul* y *Draper VDISC* puesto que estos datasets se componen de muestras de funciones, mientras que el resto tratan de scripts, proyectos o archivos completos.

Por una parte, el dataset *BigVul* se compone de muestras vulnerables y no vulnerables de tamaño variable en diferentes lenguajes de programación, especialmente en C y C++. Fue creado a mano sobre el análisis de repositorios públicos y dispone de mucha información adicional a la vulnerabilidad, por ejemplo, la fecha de publicación, un resumen

de la vulnerabilidad, el link al proyecto original, el mensaje de commit en el que se introdujo dicha vulnerabilidad o el número de vulnerabilidad *Common Vulnerabilities Exposure* (CVE) asociado; nosotros trataremos de realizar una clasificación binaria sobre las muestras, es decir, obtener si una muestra es vulnerable o no, independientemente de su clasificación CVE.

Por otro lado, el dataset *Draper VDISC* se compone de 1.27 millones de funciones de código, siendo 82.411 de ellas vulnerables, obtenidas de la clasificación estática con herramientas como *Clang*, *CppCheck* o *Flawfinder*, mientras que el resto de funciones no son vulnerables. Los autores combinaron los resultados de la clasificación de los tres programas y los mapearon en cinco vulnerabilidades *Common Weakness Enumeration* (CWE) distintas.

Finalmente el dataset que seleccionamos fue el conocido como *BigVul*, expuesto en [JF20]. A pesar de que *Draper VDISC* es un dataset prometedor y de un tamaño reseñable, buscamos un compromiso entre el tamaño y la calidad de los datos para una primera aproximación a la tarea que también nos permita realizar el entrenamiento acorde a nuestras capacidades, aunque no perdemos de vista la posibilidad de continuar el entrenamiento sobre *Draper VDISC* en un futuro. *BigVul* es un dataset compuesto por 188.637 muestras de código, mayoritariamente en C/C++ con un total de 10.900 funciones vulnerables y 177.737 no vulnerables, recogidas de proyectos de código abierto en GitHub. Para cada muestra se indica si es vulnerable o no, así como si tiene un número de vulnerabilidad CVE asociado.

4.1.1. Preprocesamiento del dataset

En esta sección detallamos el proceso aplicado en el cuaderno *00_preprocesamiento_dataset*. Nuestro primer objetivo fue determinar la cantidad de muestras en código C/C++ de las que disponemos para poder realizar una división independiente sobre la cual entrenarán los generadores y el discriminador.

El primer paso fue filtrar las muestras por dos condiciones: ser del lenguaje C o C++ y no tener una longitud mayor a 500 palabras. Eliminamos las muestras que no cumplieran con las condiciones anteriores y obtuvimos un total de 181.939 funciones sobre C/C++, siendo 9.620 de ellas vulnerables. Tras esto, debemos distribuir los conjuntos destinados a los generadores y al discriminador. Es esencial para el rendimiento y la calidad de los resultados que la división entre ellos sea independiente, no pueden entrenar sobre las mismas muestras puesto que eso conllevaría que han aprendido la misma información y daría resultados erróneos.

Se dispone del dataset original en formato .csv. Nos centraremos en el procesamiento de la función vulnerable original, correspondiente en la columna *func_before* del dataset.

Establecemos que una mitad de las muestras vulnerables será para el entrenamiento de los generadores y la otra mitad será para el discriminador; además, el discriminador se complementará con otra mitad de muestras no vulnerables de igual tamaño, de esta forma se cumple un equilibrio ideal entre las clases. Todos los generadores se entrenarán con el mismo dataset, que tiene un total de 4.810 muestras.

Guardaremos los datasets en dos formatos diferentes por conveniencia: en texto plano (.txt) y en formato *Comma-separated Values (CSV)* (.csv). En este proceso también aplicaremos la estandarización para los archivos en formato .txt de modo que los modelos solo tendrán

que ocuparse de la tokenización y vectorización; no haremos esto para los archivos .csv. No todos los archivos generados se usan en el proyecto, pero existen en caso de que se desee variar la lectura de los datos o las características que se aprenden de estos con facilidad. En total, generamos cuatro archivos diferentes:

- generator_dataset_raw.txt: Archivo de texto plano que contiene las funciones de entrenamiento del generador en bruto. No aplica una estandarización en caso de que deseemos aprender sobre estos datos.
- generator_dataset_standarized.txt: Archivo de texto plano que contiene las funciones de entrenamiento del generador estandarizadas, escritas una por una en cada línea para facilitar su procesamiento.
- generator_dataset.csv: Archivo que contiene las muestras con la información original que usa el dataset. Puede sernos de utilidad si deseamos considerar características adicionales.
- discriminator_dataset.csv: Archivo que contiene las muestras del discriminador vulnerables y no vulnerables para su entrenamiento. Las muestras están mezcladas en un orden aleatorio.

La estandarización debe cumplir varias condiciones que nos aseguren que no se altera la sintaxis ni el léxico de la programación original; aplicamos una serie de expresiones regulares que formalizan los datos según las siguientes condiciones:

- Se consideran operadores, palabras y caracteres propios de la programación como tokens únicos, se introduce un espacio entre cada carácter para aislarlo en su tokenización.
- Se eliminan comentarios
- Se eliminan los caracteres de saltos de línea y tabulaciones
- No debe haber más de un espacio entre palabras

Aplicando las reglas anteriores, obtenemos las muestras estandarizadas escritas línea por línea, facilitándonos en gran medida su futura lectura y procesamiento en los modelos.

4.2. Desarrollo de modelos generativos

Para la obtención de muestras de código vulnerable, exploraremos las posibilidades que nos ofrece el desarrollo de modelos de aprendizaje generativos con técnicas de NLP. Procesaremos sobre las propias muestras sin obtener elementos derivados, pues tratamos de obtener resultados similares a los reales sin necesidad de aplicar un proceso posterior de refinamiento o reconstrucción, como podría ocurrir si utilizásemos enmascaramiento de nombres de funciones, variables, o si procesáramos la muestra en un formato derivado, como es un árbol AST.

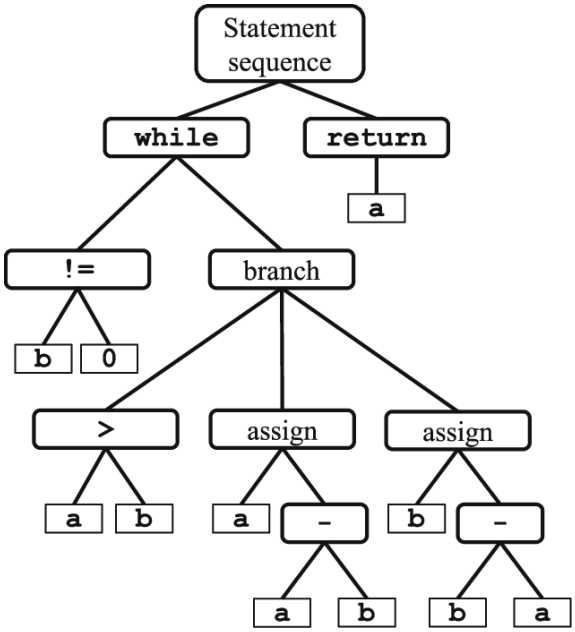

Figura 4.2: Ejemplo de AST

Aunque las técnicas anteriores son perfectamente válidas para explorar la generación de ataques de seguridad en su término más puro, nuestro objetivo también es estudiar las posibilidades que ofrecen las técnicas de NLP en la generación de código, el estudio de su calidad y las limitaciones que producen.

Sobre los modelos que produciremos, generaremos un total de 500 muestras que se almacenarán estandarizadamente en archivos para su posterior procesamiento en el discriminador y así obtendremos el rendimiento de cada modelo en la tarea.

A continuación, exponemos los modelos que hemos creado y el proceso de entrenamiento que se ha aplicado, así como las técnicas que implementa cada modelo.

4.2.1. Modelos de generación en bucle

Los que denominamos modelos de generación en bucle se basan en la generación de secuencias por medio de la retroalimentación de las predicciones, es decir, dada una secuencia de entrada, producen una predicción sobre el siguiente token, se toma esta predicción y se añade a la cadena, volviendo a entrar al modelo para predecir sobre la nueva cadena el siguiente token más probable.

Exploramos esta técnica en dos niveles de tokenización distintos en los cuales, aunque se aplica esencialmente el proceso general de entrenamiento, sus ligeras diferencias hacen que sea apreciable su forma de procesar y conseguir resultados. Además de lo anterior, entrenaremos dos modelos distintos en cada tokenización, uno sencillo que implementará una red neuronal recurrente y otro de mayor complejidad, que se compondrá de dos redes neuronales recurrentes conectadas en secuencia y que además tendrá dimensiones mayores en sus capas.

Cada modelo se divide por orden en las secciones ‘Preprocesamiento de datos’, ‘Creación y entrenamiento de los modelos’ y ‘Generación de muestras’

4.2.1.1. Modelos con tokenización por caracteres

Los modelos con tokenización por caracteres fueron los primeros de los que implementamos en el proyecto, se desarrollaron mediante el uso de la librería *TensorFlow* para la implementación de redes LSTM. Como ya se ha explicado en el Capítulo 2, las redes neuronales recurrentes son un tipo de red especial de gran utilidad en el procesamiento de datos secuenciales que requieren mantener un contexto. Las LSTM son un tipo especial diferente a la RNN nativa que solucionan el problema de desvanecimiento de gradiente a lo largo de la secuencia, especialmente cuanto mayor tamaño tenga esta, este mecanismo permite capturar mejor las dependencias a largo plazo, pero del mismo modo introduce un coste computacional mayor al tener una arquitectura más compleja.

Por otra parte, además de las LSTM, existe otro tipo muy común y de gran interés que deseamos aplicar, las GRU. Estas son otro tipo de red de la familia RNN que mantiene un equilibrio entre el rendimiento y la capacidad de la red. Su arquitectura es más sencilla y tiene un menor costo computacional al componerse internamente de menos puertas lógicas por unidad.

Quisimos explorar el rendimiento de ambas redes, sin embargo, al tokenizar por caracteres, las secuencias tienen una longitud variable de alrededor de mil caracteres. Como las redes LSTM tienen una complejidad y capacidad mayor de mantener el contexto a largo plazo, las seleccionamos para la tarea, aunque no quisimos conformarnos sin averiguar el rendimiento que puede darnos una GRU, esto fue un aliciente más para la creación de los modelos con tokenización por palabra, ya que al procesarlas de esta manera la secuencia era mucho menor, permitiéndonos aplicar la red GRU en un problema más óptimo.

Preprocesamiento de muestras

Partiendo del preprocesamiento del dataset dado en la Sección 4.1.1, ambos modelos se entrenarán sobre el archivo *generator_dataset_standarized.txt* el cual contiene las muestras de código vulnerable escritas en cada línea y realiza previamente el primer paso del procesamiento de datos, la estandarización.

A continuación, realizamos la tokenización de las muestras, hacemos uso de una capa *StringLookup* proporcionada por *TensorFlow* para convertir cada carácter único del dataset de entrenamiento en un índice numérico, de esta forma nuestro modelo podrá procesar muestras de texto. Obtenemos finalmente que el vocabulario se compone de un total de 99 caracteres.

El siguiente paso es la vectorización, en el proceso de vectorización, debemos generar las muestras y etiquetas como índices numéricos que procesará el modelo. No obstante, antes de vectorizar el conjunto, debemos abordar la tarea de etiquetado.

Estamos tratando un problema de generación de secuencias espontáneo, es decir, partimos desde una secuencia de entrada cualquiera y debe generarse una secuencia de salida que se complete en base a esta. Aplicaremos una técnica sencilla pero efectiva para el aprendizaje en contexto de las muestras. Dada una secuencia de tamaño *sequence_length*, establecemos que la muestra tomará los primeros *sequence_length - 1* tokens, mientras que la etiqueta tomará los *sequence_length - 1* tokens desplazados una posición hacia la derecha desde la primera posición original.

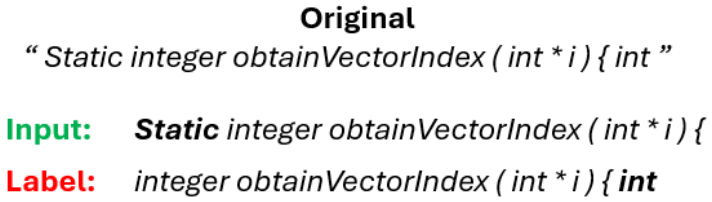

Figura 4.3: Etiquetado de muestras

Es decir, por ejemplo, que si tenemos un tamaño de división de secuencia establecido en 256 elementos, la entrada tomará las posiciones [0,254] de la original, mientras que al etiqueta serán las posiciones [1,255]. Esta técnica sencilla nos permite generar tokens manteniendo el contexto de forma sólida a la vez que permite generalizar en el aprendizaje. En función del tamaño de secuencia escogido se obtendrá una complejidad mayor en la captura del contexto, lo cual también debe implicar un modelo con una estructura más compleja para capturarlas.

Establecemos en nuestros modelos un tamaño de secuencia de 1024 tokens y, tras realizar el etiquetado como se ha comentado, aplicamos la vectorización al conjunto de datos para obtener el dataset listo para ser procesado.

Creación y entrenamiento de los modelos

Tras obtener los datos preprocesados, el siguiente paso es implementar los modelos, produciremos dos modelos distintos aunque iguales en su núcleo ya que aplican el mismo proceso pero con diferente nivel de complejidad, los modelos recibirán un vector de entrada de 1024 tokens y darán como resultado un vector de predicciones del tamaño del vocabulario. Los modelos se componen de las siguientes capas en el orden que se comentan:

- Embedding: La capa de incrustación produce un vector de dimensión fija para cada carácter de la entrada, permitiendo tomar una complejidad de relaciones mucho mayor entre los tokens.
- LSTM: La salida de las incrustaciones entra en una capa *Long-Short Term Memory*. Esta capa devolverá también el estado de las celdas y el estado oculto, pues serán los valores a los que se inicializará la red en el siguiente paso cuando vuelva a repetir el proceso, de esta forma se mantiene el contexto
- Dropout: La capa de Dropout trata de prevenir el sobreajuste del modelo a los datos de entrenamiento haciendo que un porcentaje *dropout-rate* de neuronas se establezca a 0 de forma aleatoria durante el entrenamiento con el objetivo de que el modelo no se apoye demasiado en el cálculo de ninguna neurona en especial. Establecemos el ratio de desconexión de neuronas a la salida de la red LSTM en un 20 %
- Dense: Finalmente, una capa densa del tamaño del vocabulario será la salida del modelo que dará una predicción en logits para cada carácter.

El modelo básico se compondrá de todas las capas anteriormente mencionadas, para las cuales establece un tamaño de dimensión de incrustación de 64 elementos y una cantidad

de 512 unidades en la capa LSTM. Por otro lado, el modelo complejo se compondrá de una capa doble LSTM-Dropout seguida en las que establecemos un tamaño de dimensión de incrustación de 128 elementos y un total de 1024 unidades de celdas para cada capa LSTM.

Usaremos una función de pérdida *Sparse Categorical Cross Entropy* que se calcula directamente sobre los logits de salida del modelo, y un optimizador *Adam* para actualizar los parámetros de la red. Definimos también un callback *EarlyStopping* que monitorizará la pérdida del entrenamiento y devolverá al modelo al mejor estado anterior si se contabilizan ún número *patience* establecido en 3 épocas sin mejorar la pérdida en un valor *delta* no mayor de 0.075

Finalmente, compilamos el modelo y lo entrenamos sobre un número indefinido de épocas hasta que se encuentre el mejor punto haciendo uso del callback anterior. Obtenemos tras el entrenamiento los siguientes resultados:

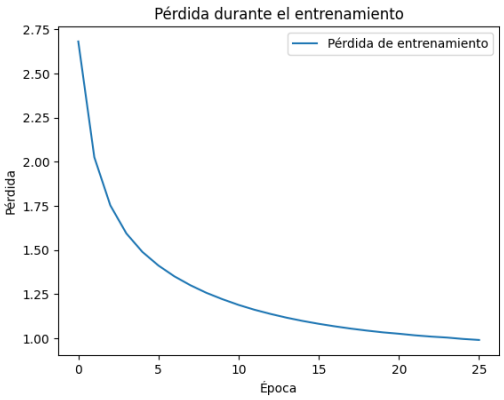

Figura 4.4: Pérdida modelo básico

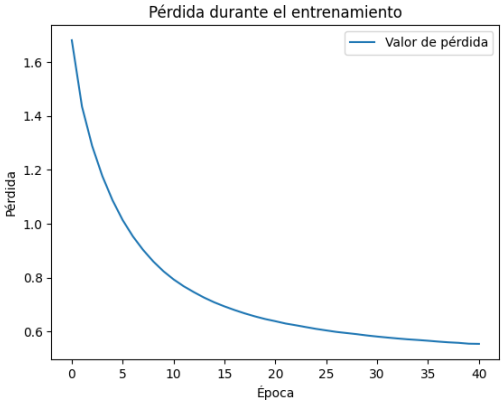

Figura 4.5: Pérdida modelo complejo

El entrenamiento continúa hasta que se cumplen las condiciones el callback *EarlyStopping* y se detectan tres épocas sin mejorar la pérdida en un nivel significativo. El modelo básico termina tras 26 épocas con una pérdida de 0.98, mientras que el modelo complejo finaliza tras 41 épocas, con una pérdida de 0.55. A la vista de ambos datos, podremos esperar que el modelo complejo cumple correctamente su función por el momento, ya que consigue un mejor valor de pérdida y debería obtener mejores resultados en la generación. Confirmaremos si esto se cumple realmente tras averiguar su rendimiento con el discriminador.

Generación de muestras

Una vez completado el entrenamiento, disponemos de modelos capacitados para producir la siguiente predicción dada una secuencia de caracteres; sin embargo, buscamos generar secuencias, no únicamente la siguiente predicción, solucionaremos esto por medio de la creación de una clase generadora que haga uso de cada modelo y actúe como interfaz para usarlo. Esta clase nos permitirá generar de forma sencilla secuencias de código de longitud variable aplicando el proceso el número de veces que le indiquemos, devolviendo finalmente la secuencia producida por el generador en sus correspondientes caracteres. Además, también nos permitirá aplicar una máscara para evitar la generación de tokens no deseados, como pueden ser caracteres especiales o saltos de línea. El proceso es sencillo y se divide en los siguientes pasos:

1. Se obtiene la secuencia de entrada al modelo y se vectoriza para ser procesada.
2. Se obtienen las predicciones y el estado interno de la red para la siguiente predicción.
3. Se aplica la máscara sobre las predicciones del modelo, previniendo los tokens indeseados
4. Se obtiene el siguiente token muestreando sobre una distribución categórica de los logits de salida. Es esencial realizar el muestreo de salida aplicando alguna distribución, pues esto permite la aleatoriedad de las muestras. En caso de que se buscase, como puede ser lógico pensar, obtener siempre el siguiente token más probable aplicando la función *argmax*, se obtendría un modelo determinista, puesto que las salidas siempre serán las mismas para cada secuencia de entrada. Esto es un problema esencial a paliar en problemas de NLP para introducir aleatoriedad en la generación.
5. Se convierte el token en su correspondiente carácter, se devuelve finalmente el carácter junto al estado de la red.
6. Se repite el proceso el número de veces indicado según la longitud de la secuencia.

Finalmente, para ambos modelos, guardaremos un total de 500 muestras para cuantificar su posterior rendimiento en un archivo de texto. Para decidir la secuencia de comienzo de las muestras usaremos un método simple, volveremos a leer el dataset guardando cada palabra única de comienzo de cada muestra, obtenemos una lista compuesta de 1331 palabras distintas sobre las que elegiremos aleatoriamente una para comenzar la predicción de la secuencia, además de esto, introducimos variaciones en el tamaño de las muestras generadas estableciendo un tamaño mínimo y máximo de secuencia sobre los cuáles se tomará la longitud aleatoriamente. Por último, se escriben las muestras en cada archivo por cada modelo.

4.2.1.2. Modelos con tokenización en palabras

El modelo tokenizado por palabras se fundamenta en la misma base que el tokenizado por caracteres, trataremos de dar explicación a sus diferencias antes que volver a tratar de explicar lo que es común a ambos.

Preprocesamiento de muestras

La primera diferencia que encontramos se da en la forma de procesar las muestras, consideramos cada token como una unidad dividida por espacios, es por ello que es vital aplicar la estandarización para aislar cada elemento esencial de las muestras de entrenamiento. Como ahora se tokenizará por palabras completas, encontramos que el vocabulario crece en gran tamaño, cada palabra única se considerará un token del vocabulario, además, gracias a la estandarización eliminamos tokens duplicados con el mismo significado, los siguientes tokens son un ejemplo del incremento de tamaño que encontraríamos si no aplicásemos una estandarización correcta

En la tabla 4.1 vemos que al no estar separados por espacios, cada uno se consideraría una unidad diferente. Limitamos el tamaño de vocabulario en consecuencia en 20.000 tokens. Tras esto procedemos a la vectorización, establecemos que el tamaño máximo de secuencia será de 256 elementos. Disponemos de muestras de longitud variable de tamaños

Token real	Tokens duplicados
void	void(, void() , void(){
return	return;
string	string.separate, string.strip(), ...

Tabla 4.1: Tokens duplicados con mismo significado

mayores y menores que 256, para controlar estos casos aplicaremos una transformación que truncará las muestras de tamaño mayor para reducirlas o rellenará el tensor con tokens de relleno [PAD], los cuales se enmascararán durante el entrenamiento y no influirán en el aprendizaje.

Por último, generamos las etiquetas para las entradas con la misma estrategia de desplazamiento de posición que se aplicó en la tokenización por caracteres.

Creación y entrenamiento de los modelos

Los modelos de tokenización por palabras se componen de capas siguiendo la misma estructura que el modelo por caracteres, se realiza un *embedding* a la entrada del modelo sobre los tokens de la secuencia, se continúa aplicando una red GRU en lugar de una red LSTM con *dropout* a su salida para el modelo básico y una doble red para el modelo complejo, aunque vemos una diferencia fundamental en la capa final, la salida se devolverá en una capa *Dense* con un total de *vocab_size* elementos, por lo cual se darán predicciones sobre 20.000 tokens.

El tamaño de vocabulario es una diferencia a destacar en el procesamiento de esta forma, pues ahora nuestra salida será mucho mayor y, aunque puedan parecer equivalentes, podemos esperar que el comportamiento con el modelo de caracteres sea explícitamente diferente a la hora de ser entrenado.

El modelo básico se compone de una única red GRU de 128 unidades. La dimensión de *embedding* será de 128 elementos y el *dropout* aplicado a la salida de la red GRU será de 0.2. Por otra parte, el modelo complejo se compondrá de una doble red GRU de 256 unidades cada una, con una dimensión de *embedding* de 256 elementos y un ratio de *dropout* de 0.2.

Entrenamos el modelo indefinidamente hasta que obtengamos una pérdida que no mejore por más de un valor mínimo *delta* de 0.075 aplicando el mismo callback para parar el entrenamiento cuando se cumpla.

En las figuras 4.6 y 4.7 el modelo básico entrenó durante 87 épocas logrando una pérdida de 0.9819, mientras que el modelo complejo entrenó 89 épocas finalizando con una pérdida de menor valor de 0.5615.

Generación de muestras

La generación de muestras se realizará de la misma forma, muestreando sobre las probabilidades usando una distribución categórica. La única diferencia esencial es que ahora no consideramos el espacio como un token, sino que lo añadimos entre cada par de tokens al escribir las muestras. Guardamos 500 muestras de nuevo para su posterior análisis.

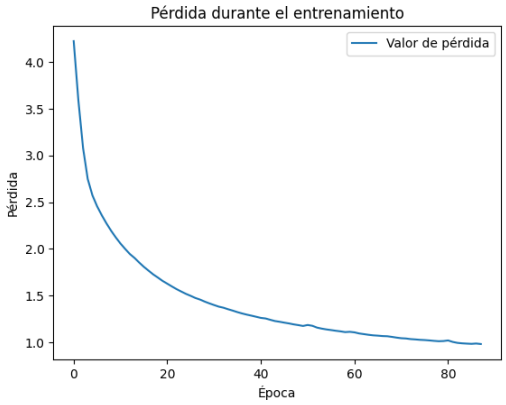

Figura 4.6: Pérdida modelo básico

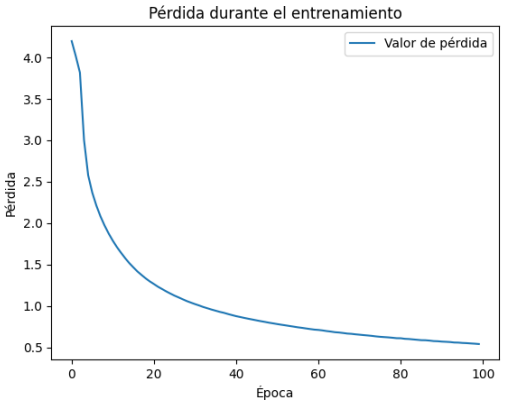

Figura 4.7: Pérdida modelo complejo

4.2.2. Modelos LLM basados en transformadores

Los modelos LLM son un tipo de modelos especialmente diseñados para comprender y generar sobre lenguaje natural con una alta sofisticación. Se caracterizan por estar entrenados con cantidades masivas de datos y aplicar técnicas especializadas en la comprensión y generación de texto. Se componen de miles de millones de parámetros que les permiten capturar las complejidades de datos de tal cantidad. Estos modelos no se entrenan de forma nativa como podemos hacer con los anteriores que hemos conceptualizado y entrenado desde su base, sino que se entrenan previamente con máquinas con la capacidad suficiente y nosotros los ajustamos en un proceso llamado *fine-tuning* para tareas específicas, como por ejemplo, la generación de código.

La mayoría de modelos LLM se basan en arquitecturas de transformadores; estas estructuras son especialmente eficientes en el procesado de datos en lenguaje natural aplicando la atención donde más relevante es de la secuencia. Gracias a su estructura no son secuenciales en el procesamiento, permitiendo un mayor rendimiento y velocidad que las redes neuronales recurrentes. Las arquitecturas con transformadores son explicadas en más detalle en la Sección 2.3 del Capítulo 2

4.2.2.1. GPT-2

El primer modelo LLM que ajustaremos para la generación de muestras vulnerables será GPT-2, se compone de hasta 1.5 mil millones de parámetros en su versión completa y se entrena de forma no supervisada, aplicando un enmascaramiento a tokens aleatorios y dando la predicción según el contexto de los tokens que lo rodean.

El uso de un modelo preentrenado nos facilita mucho el proceso en muchos aspectos. El preprocesamiento de las muestras se realiza de forma sencilla, cargando el dataset y aplicando un *data collator*, que se encarga de tokenizar todo el dataset, dividirlo en secuencias de tamaño fijo y enmascarar tokens aleatoriamente para su predicción.

El siguiente paso es cargar el modelo que vamos a entrenar. GPT-2 nos es dispuesto en diferentes versiones según el tamaño del modelo, podemos elegir entre las siguientes: 'gpt2', 'gpt2-medium', 'gpt2-large' y 'gpt2-xl', cada uno de ellos compuesto por 117, 345, 774 millones y 1.5 mil millones de parámetros respectivamente; entrenaremos para este trabajo el modelo básico, especialmente por las capacidades computacionales de las que

disponemos.

El entrenamiento se realizó en 20 épocas, momento a partir de que el modelo deja de mejorar la pérdida, la cual se sitúa en 1.0664 al final de la ejecución para el conjunto de entrenamiento. El entrenamiento es computacionalmente más costoso que en el resto de modelos anteriores debido a su tamaño, pero no está linealmente relacionado con el tiempo, ya que gracias a la arquitectura de transformadores se permite la paralelización, a diferencia del proceso secuencial en las redes neuronales recurrentes.

La generación de muestras se realizará de la misma forma que en el resto de modelos, GPT-2 es un modelo capaz de completar secuencias de texto dada una secuencia de entrada, daremos al modelo una de las primeras palabras de cada muestra como comienzo y el modelo generará a partir de esta. En la generación con GPT-2 podemos establecer diversos parámetros generales de los transformadores proporcionados por *HuggingFace* que nos permiten probar diferentes opciones de generación, establecemos los siguientes:

- *do_sample*: Este parámetro indica que para la generación se realizará *sampling*, es decir, se muestreará la siguiente palabra aleatoriamente según su probabilidad en lugar de elegir siempre la más probable. Esto es lo que venimos haciendo en los modelos tokenizados y se utiliza para dar mayor creatividad a las predicciones y eliminar el determinismo como ya se ha comentado anteriormente. Establecemos a 'true' su valor

- *top_k*: Este parámetro va en combinación con el anterior y nos indica cuantas palabras de las de mayor probabilidad se deben considerar a elegir como la siguiente palabra en la secuencia, es decir, que se muestreará sobre cualquiera de las *top_k* palabras más probables. Establecemos *top_k* en 50. Un valor mayor o menor de este parámetro influirá en la aleatoriedad de la siguiente predicción

- *top_p*: El parámetro *top_p* se combina con *top_k* limitando el conjunto de palabras del muestreo a aquellas que cumplan un porcentaje *top_p* del espacio de predicciones. Establecemos *top_p* en 0.95, por lo cual se seleccionarán del conjunto las palabras que sumen un 95 % de las probabilidades de predicción. Un valor mayor o menor influye en la flexibilidad y las posibilidades del modelo.

Finalmente, a las muestras que generamos les aplicamos una estandarización para estructurarlas correctamente a la hora de guardarlas, eliminando los saltos de línea, espacios en blanco duplicados y comentarios. Guardamos 500 muestras en un archivo para su posterior procesamiento.

4.2.2.2. CodeBERT

BERT es una familia de modelos de aprendizaje automático que, a diferencia de GPT son capaces de obtener el contexto de una secuencia recorriéndola en ambas direcciones, es decir, es bidireccional. Su aprendizaje, al igual que GPT, es no supervisado, aplicando un entrenamiento MLM. Su funcionamiento se explica con mayor detalle en el Capítulo 2 en la Sección 2.3.1

Sobre BERT se han desarrollado variantes que aplican el modelo a diferentes tareas. CodeBERT es una de ellas, fue introducido por Microsoft Research y la Universidad de

Hong Kong en [LGR+21] como una herramienta optimizada para comprender y resolver problemas que involucran tareas de lenguaje natural y lenguaje de programación.

Usamos CodeBERT para tratar de generar código vulnerable de la misma forma que hacemos en el resto de modelos. Al basarse en la arquitectura de transformador y estar estandarizado, la mayoría del proceso es similar al que aplicamos en la sección anterior para GPT-2. Cargamos el dataset de entrenamiento y el tokenizador se encargará de enmascarar tokens aleatoriamente, usamos un tokenizador *RobertaTokenizer* puesto que CodeBERT internamente es un modelo RoBERTa ajustado para las tareas de código, por lo cual del mismo modo, el modelo que cargaremos será un modelo RoBERTa especializado en código.

Nuestro modelo CodeBERT se compone de una cantidad de parámetros similar al que hemos utilizado en GPT-2, con 125 millones de parámetros internos. Su entrenamiento se realizó sobre un total de 10 épocas, punto en el que encontramos que se estabiliza su pérdida de entrenamiento, la cual alcanza un valor de 0.654 al terminar.

La generación de muestras se dará como se viene haciendo habitualmente, al modelo se le proporcionará una secuencia de comienzo y se le pedirá predecir el próximo token más probable. Esto se logra añadiendo un token de máscara *mask* y obteniendo la predicción del modelo, repitiendo el proceso tantas veces como deseemos para obtener una longitud de muestra aceptable. El modelo devuelve logits en su salida, aplicaremos para cada predicción una función *softmax* y un muestreo sobre una distribución multinomial gracias a la librería *Torch* para obtener el siguiente token más probable de la secuencia. Finalmente, con el nuevo token predicho, se añade a la secuencia y se repite el proceso, tratando de obtener el siguiente. Guardamos un total de 500 muestras en un archivo de texto para su posterior procesamiento.

4.3. Desarrollo del discriminador

El discriminador tendrá como objetivo cuantificar el rendimiento de los modelos generativos. Estará entrenado sobre la mitad de muestras vulnerables disponibles en el dataset y otra mitad de muestras no vulnerables equivalentes en tamaño para equilibrar las clases. Realizamos un problema de clasificación binaria, daremos una predicción entre vulnerable y no vulnerable para cada secuencia de entrada.

Preprocesamiento de muestras

El modelo hará uso del archivo obtenido en *discriminator_dataset.csv*, tomando las columnas *func_before* y *vul* como entrada y etiqueta de cada muestra respectivamente. El primer paso que realiza es dividir el conjunto de datos en tres conjuntos independientes: entrenamiento, validación y test, aplicando una división habitual entre cada uno de 80-10-10 respectivamente.

Tras obtener los conjuntos independientes, haremos uso de la capa *TextVectorization* proporcionada por Tensorflow para aplicar el proceso de estandarización, tokenización y vectorización cómodamente. La estandarización que aplicará la capa de vectorización a la entrada del modelo es la misma que se realiza en el preprocesamiento de datos para los generadores; sin embargo, dividimos la estandarización para el discriminador por si se desea probar nuevos caminos sin necesidad de alterar y generar archivos diferentes en cada ocasión, manteniendo los datos originales.

Nuestro discriminador aplicará una **tokenización por palabras**, al igual que se ha realizado en los generadores. Limitaremos el tamaño de vocabulario del discriminador a 30.000 tokens (el tamaño total de nuestro conjunto de entrenamiento se compone de alrededor de 120.000 tokens).

Concepción y entrenamiento del modelo

El discriminador procesará una muestra de código y tratará de predecir si esta es vulnerable o no vulnerable. Tratamos de realizar una clasificación binaria entre [0,1], estableciendo el umbral de aproximación en 0.5. El modelo se compondrá de varias capas: una capa de embedding a la entrada para vectorizar los tokens de las muestras, con una dimensión fija de 256 elementos, una capa LSTM bidireccional, la cual procesará la secuencia en ambos sentidos en cada paso, internamente se comporta como dos capas LSTM que recorren la secuencia en un sentido u otro y ponderan el resultado en la salida. Aplicamos también un regularizador de kernel L2 que aplica una penalización a los pesos de salida de la red para tratar de prevenir el sobreentrenamiento, así como una capa *dropout* de valor 0.3 para tratar de solventar el sobreajuste y finalmente una capa *Dense* con una única neurona de salida, que junto a una función de activación sigmoide nos dará la probabilidad entre [0,1] de que la entrada sea vulnerable o no vulnerable.

Se entrenará el modelo en un número de épocas indefinido, sin embargo, la estrategia que seguiremos será detenerlo como hemos hecho en el resto de modelos, pero en esta ocasión considerando la pérdida del conjunto de validación, cuando esta no mejore en un valor mínimo *delta* de 0.02 durante 3 épocas seguidas, el entrenamiento se detendrá.

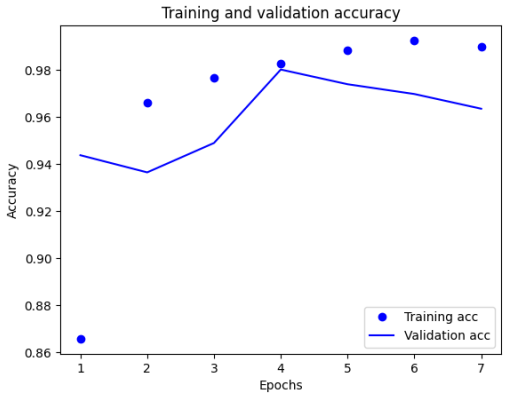

Figura 4.8: Precisión

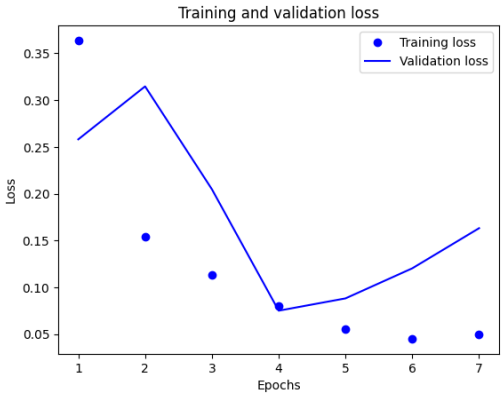

Figura 4.9: Pérdida del discriminador

Como se puede ver en las figuras 4.8 y 4.9 nuestro modelo entrena durante un total de 7 épocas, obteniendo su mejor rendimiento en la cuarta época, punto al cual se restauran sus pesos. Podemos apreciar que la precisión y la pérdida del entrenamiento y validación crecen hasta la cuarta época y a partir de este punto el modelo comienza a mejorar únicamente para el conjunto de entrenamiento, indicándonos que el modelo comienza a sobreajustarse a los datos de entrenamiento.

Capítulo 5

Experimentos y Resultados

En este capítulo analizamos los experimentos realizados y exponemos sus resultados y las conclusiones que obtenemos. En la Sección 5.1 analizamos las métricas obtenidas de los modelos aplicados sobre el juicio del discriminador. En la Sección 5.2 exponemos las limitaciones y problemas que encontramos tras extraer conclusiones de los resultados producidos por los generadores.

5.1. Métricas obtenidas y resultados

Tras haber obtenido un conjunto de muestras de cada generador, mediremos su capacidad para engañar al discriminador. Nos interesa obtener la cantidad de falsos negativos (FN) que se producen en la clasificación. Los falsos negativos, como indica su nombre, son las muestras positivas que se han clasificado como negativas erróneamente; aplicado a nuestro problema, son las muestras vulnerables que se han contabilizado como no vulnerables. Las posibilidades de la clasificación van en función de las etiquetas dadas por la muestra y las obtenidas en la predicción. Vemos esto visualmente en la **matriz de confusión**

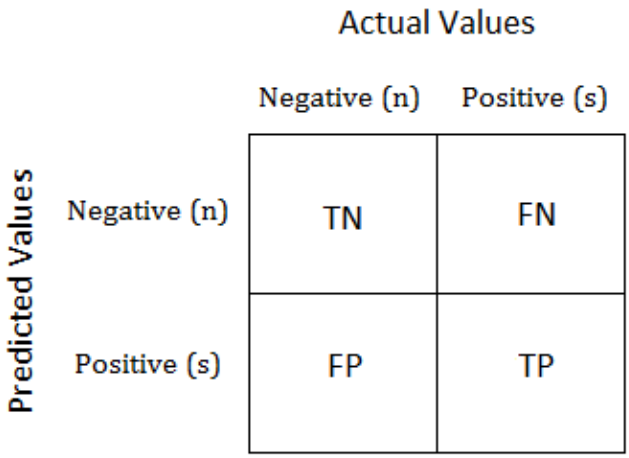

Figura 5.1: Matriz de confusión

En la figura 5.1 podemos ver los casos que se dan en la clasificación en función de la etiqueta

real y la predicción. Dado que siempre generaremos valores vulnerables, es decir, positivos, nos interesará obtener una predicción negativa, por tanto, obtener falsos negativos. Los valores verdaderos positivos (TP) serán los casos en los que el discriminador ha acertado con la clasificación.

Respecto a cuantificar los resultados, existen varias métricas habituales en los problemas de clasificación, a continuación explicamos su objetivo y cuáles hemos considerado:

- **Exactitud**: La exactitud o *accuracy* nos da la proporción de predicciones correctas sobre el total de predicciones, por tanto, el resto de la proporción son las predicciones incorrectas.

$$\frac{TP + TN}{TP + TN + FP + FN}$$

Figura 5.2: Cálculo exactitud

  Como vemos en la siguiente figura, la fórmula se simplifica, ya que nunca habrá verdaderos negativos (TN), pues solo disponemos de muestras de una clase.

- Precisión: Nos da la proporción de verdaderos positivos sobre las predicciones positivas. Como nuestras predicciones positivas siempre serán verdaderas positivas ya que solo hay muestras vulnerables, esta métrica no nos es de utilidad.

- Recall: Nos indica que proporción de casos positivos reales fueron identificados sobre todas las muestras positivas. En nuestro problema siempre coincidirá con la exactitud ya que solo disponemos de muestras vulnerables.

- F1-Score: Es la media armónica entre la precisión y recall. Puesto que la precisión no es un valor útil, no será válido por la misma lógica F1-Score

Evaluaremos entonces la **exactitud** de la detección, ya que al generar muestras de una única clase es la única métrica con resultados válidos. Exponemos en '06_resultados' el cálculo de los siguientes mostrados en la figura 5.3 tras usar el discriminador para predecir sobre las 500 muestras de cada modelo:

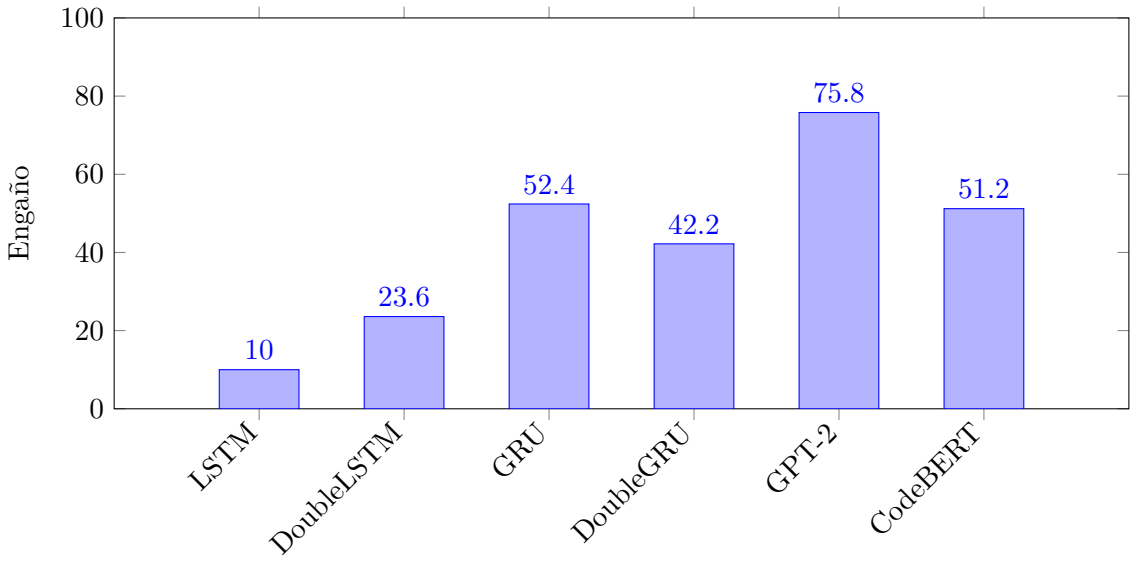

Figura 5.3: Tasa de engaños al discriminador por modelo

A la vista de los resultados, vemos que el rendimiento va ligado a la naturaleza de los modelos, por una parte tenemos los modelos tokenizados por caracteres, que son los que peor rinden en la tarea de generación, a continuación los modelos tokenizados por palabra que logran un engaño intermedio y por último los modelos LLM sobre transformadores, que en relación con su potente estructura también consiguen los mejores resultados.

Sobre los resultados podemos apreciar varios puntos de interés, por una parte, para los modelos tokenizados por caracteres, observamos cómo el modelo *DoubleLSTM* de mayor complejidad logra unos resultados que duplican al de su versión básica *LSTM*, esto significa que los modelos tokenizados por caracteres necesitan de una estructura más compleja para obtener dependencias coherentes y mejorar los resultados, no obstante, son los que peor rinden en comparación con el resto. Por otra parte, para los modelos tokenizados por palabras observamos que el modelo básico *GRU* obtiene los segundos mejores resultados de las pruebas, logrando un 52.4 % de engaño al discriminador pero su versión compleja *DoubleGRU* no logra un mejor desempeño, indicándonos que no necesita una estructura mayor para capturar dependencias y que esta puede haberse sobreajustado a los datos de entrenamiento. Por último, encontramos los modelos LLM, estos modelos se corresponden en sus resultados con su complejidad, el modelo GPT-2 obtiene el resultado más alto de todos, mientras que CodeBERT, también superando a los modelos anteriores a excepción del modelo *GRU* de palabras, logra engañar en más de la mitad de las ocasiones.

De esto podemos constatar varios hechos: primero, que los modelos transformadores son los que mejor rinden en la tarea de generación en consonancia con su capacidad, segundo, que una estructura más compleja no implica un mejor desempeño, pues debe considerarse el conjunto del problema y la complejidad a la que puede llegar, por una parte vemos como una estructura más compleja ayuda a capturar mejores dependencias en el generador por caracteres, mientras que esto no se cumple para los generadores por palabra, el tamaño de vocabulario juega un papel esencial en esta diferencia de rendimiento.

5.2. Análisis del rendimiento

En esta sección tratamos de exponer el desempeño de los modelos más allá de las métricas, en la Sección 5.2.1 exponemos el rendimiento y tiempo de los modelos en el entrenamiento y las razones que hay detrás del coste computacional de cada modelo. En la Sección 5.2.2 hacemos un breve análisis de las muestras producidas por cada modelo y señalamos sus fortalezas e inconvenientes.

5.2.1. Rendimiento computacional

En primer lugar, quisiéramos comentar que el tiempo de entrenamiento de los modelos, aunque varía según factores, como principalmente la máquina en la que han sido entrenados o la arquitectura de los modelos, puesto que han sido todos ellos entrenados sobre la misma máquina que se describe al comienzo del Capítulo 4 y usando el mismo conjunto de datos, podemos comparar la proporción del tiempo de entrenamiento entre ellos, señalando los inconvenientes de cada arquitectura. Cabe destacar que todos los modelos fueron entrenados hasta el punto en el que su pérdida no mejoró más sobre el conjunto de entrenamiento.

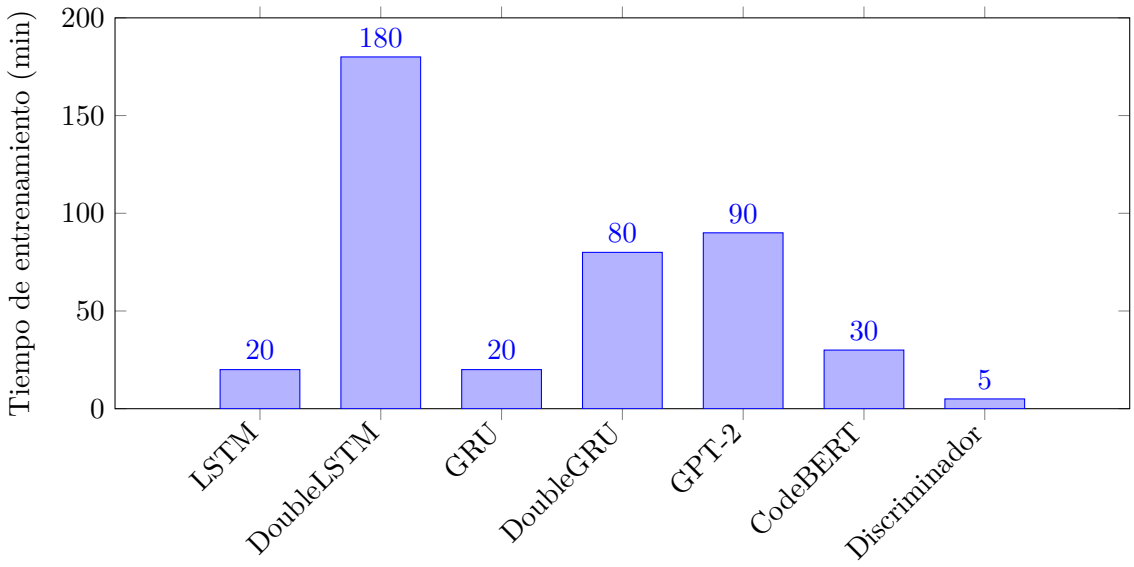

Figura 5.4: Tiempos de entrenamiento por modelo

En la figura 5.4 vemos el tiempo en minutos que ha durado el entrenamiento de cada modelo. A primera vista observamos como el modelo *DoubleLSTM* destaca en los tiempos, doblando al siguiente en duración. Esto es principalmente debido a su estructura, por un lado realizan un procesamiento secuencial al cual se añade la falta de paralelismo, las redes LSTM necesitan de los resultados del paso anterior para poder ejecutar los del siguiente, este proceso aunque como decimos es secuencial, puede paralelizarse y replicarse como hacemos al procesar un batch de muestras, sin embargo, al combinar dos redes LSTM la ejecución de una depende directamente de la salida de la otra, aumentando exponencialmente el tiempo en función del número de redes presentes. Esto ocurre también en el modelo *DoubleGRU*, aunque aquí destaca la eficiencia computacional de las redes GRU, que se componen de celdas más sencillas que las de las redes LSTM y así obtienen

un resultado que aunque mayor, es la mitad del resultado de la red *DoubleLSTM*.

Para los modelos LLM observamos el rendimiento de las estructuras de transformadores, aunque ambos GPT-2 y CodeBERT disponen de alrededor de 120 millones de parámetros internos, son muy rápidos en relación con el entrenamiento gracias al paralelismo que permiten los transformadores. Apreciamos cómo los transformadores son mucho más potentes que las redes neuronales recurrentes tradicionales.

En lo respectivo al discriminador, que usa una red LSTM bidireccional y que realiza una tarea de detección para dar un resultado binario a la salida, vemos que obtiene un tiempo de entrenamiento mucho menor que el resto de modelos, siendo capaz de obtener los patrones internos de las muestras con rapidez. Esto es un claro ejemplo de las diferencias en la naturaleza de ambas tareas, es decir, la detección y la generación de muestras, aunque pueden prácticamente similares al ser una la contraparte de la otra, difieren radicalmente en su complejidad interna, siendo la generación una tarea mucho más compleja.

5.2.2. Análisis de las muestras

El último análisis que podemos dar sobre el rendimiento de los modelos generativos es exponer la calidad de su generación sus muestras. El análisis de código es una tarea donde el ojo humano suele tener la última palabra, especialmente en la detección de vulnerabilidades, realizamos a continuación una breve puntualización donde exponemos ejemplos de resultados concretos producidos por cada modelo e incluimos un fragmento de ejemplo de código de cada uno de ellos:

- **Modelos tokenizados por caracteres**: En ambas versiones encontramos una generación estructuralmente similar, sin embargo, el modelo básico tiende a perder caracteres de palabras con mayor facilidad que el modelo complejo o abre y cierra llaves y paréntesis sin corresponderse, por lo general, también mantiene peor el contexto, podemos apreciar como las palabras del modelo básico guardan menor relación contextual que las del modelo complejo con sus adyacentes.

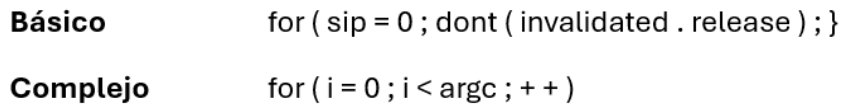

Figura 5.5: Ejemplos de código en modelos de caracteres

En la figura 5.5 podemos ver un ejemplo de lo que comentamos, son ejemplos extraídos de los archivos de muestras de los resultados de cada modelo, de la línea 29 y de la línea 27 de cada archivo respectivamente. El modelo básico es capaz de abrir un bucle *for* e inicializar la variable, sin embargo pierde el contexto rápidamente a mitad de secuencia y no es capaz de continuar coherentemente. Por otra parte, el modelo complejo sí que es capaz de declarar un bucle perfecto y coherente, aunque falla en un único carácter, al asignar la variable autoincremental i al final del bucle. La generación de estos modelos presenta resultados prometedores en relación con la complejidad del modelo, no obstante, pecan en el rendimiento debido a la incapacidad de paralelizar el proceso, aunque en caso de que se pudiese entrenar los modelos sin las limitaciones computacionales, la tokenización por caracteres es un enfoque

poderoso a la hora de mantener contexto en la red y entre las propias palabras que genera.

- **Modelos tokenizados por palabra**: Los modelos tokenizados por palabras tienden a producir tokens que denominamos 'comodín' con mayor frecuencia, estos son tokens con una frecuencia de aparición mayor que otros como por ejemplo 'if' o ')' y que por ende obtienen una mayor probabilidad en la predicción. La razón de esto tiene relación con la tokenización, el vocabulario de predicción se compone de 20.000 palabras sobre los 100 caracteres de los anteriores, provocando que se necesiten unos embbeddings con gran carga de memoria junto a un dataset de gran tamaño de igual manera para poder ajustar las dependencias correctamente, siendo más difícil obtener resultados prometedores sin disponer de un dataset de tamaño suficiente ni de la capacidad de procesamiento necesaria. Encontramos que este tipo de tokenización se ve penalizada por la frecuencia de las palabras del vocabulario puesto que favorecerá la predicción de tokens que aparecen en más ocasiones frente a aquellos que no son tan frecuentes, esto es un problema derivado de la naturaleza de la programación, donde las palabras se crean, se combinan o se inventan frecuentemente.

Básico	`if = ( char ) ( ( unsigned ) ) ;`
Complejo	`if = ( const + 6 ) ; break ; }`

Figura 5.6: Ejemplos de código en modelos de palabras

Las muestras de la figura 5.6 han sido extraídas de los resultados, de las líneas 34 y 46 respectivamente. Vemos que ambos modelos no son capaces de abrir las llaves necesarias para el bloque condicional e introducen un signo '=' en la zona incorrecta. Hay una presencia de tokens 'comodín' mucho mayor que el resto, en gran parte por la frecuencia de las palabras, ya que ahora existen palabras que solo aparecen una vez debido a que se consideran únicas y se ven penalizadas por esta forma de tokenización, donde otras frecuentes como *for* o *if* se ven recompensadas.

- **Modelos LLM**: Por último, los modelos transformadores, con una estructura mucho más compleja añadida a su preentrenamiento, dan resultados prometedores, especialmente GPT-2, el cual es capaz de mantener el contexto durante largas secuencias, sin embargo, para CodeBERT vemos que se penaliza en la generación de muestras, probablemente por su procesamiento bidireccional en una tarea que es esencialmente unidireccional como es la generación espontánea.

GPT-2

```
PolygonExport int GetPolygonOpacity ( VncState * Vnc, const
int Opacity, int HasPolygonOpacity ) {
        Bool IsDefined ( Vnc -> HasPolygonOpacity ) ; return
        Opacity > = VP9_OPEN_OPEN_NONE ;
}
```

CodeBERT

```
case RT_BUG: return -EFAULT ;
case RT_ERROR: return -EFAULT ;
case RT_PRESS: return -EFAULT ;
case RT_USER: return -EFAULT ;
case RT_GROUP: return -EFAULT ;
```

Figura 5.7: Ejemplos de código en modelos LLM

A pesar de todo, vemos en la figura 5.7 que ambos son capaces de generar código manteniendo el contexto, es remarcable la capacidad de GPT-2 a la hora de obtener dependencias, en el fragmento extraído de su archivo de muestras en la línea 460, podemos ver una declaración de función casi perfecta en su lógica, por otra parte, para CodeBERT podemos apreciar de la linea 79 un ejemplo de estructura *switch-case* con diferentes opciones.

5.3. Limitaciones extraídas sobre los resultados

A la vista de los resultados y de los análisis derivados de estos, extraemos una serie de limitaciones que enumeramos a continuación:

- Contexto y dependencias: Encontramos que la tokenización por caracteres es capaz de construir sentencias con pocos fallos lógicos manteniendo un contexto general, no obstante, encuentra dificultades a la hora de mantener el contexto en largas secuencias debido a la cantidad de tokens a considerar, necesitando de una arquitectura mucho más costosa computacionalmente. En cuanto a la tokenización por palabras, resulta difícil establecer un aprendizaje correcto sobre el lenguaje natural debido al tamaño de su vocabulario, además, el problema del vocabulario se torna inasumible cuando este puede tener un tamaño infinito al poderse definir nombres, variables, funciones, clases,etc. El modelo se verá limitado siempre por estos nombres, es de interés explorar los resultados que pueda ofrecer un procesamiento con enmascaramiento de nombres, enfocándose en la creación de muestras estructuralmente correctas, aunque todas las posibilidades conllevan la dificultad de generar muestras funcionales en bruto sin necesidad de procesamiento posterior. Respecto a los modelos LLM basados en transformadores, observamos una gran capacidad en estas estructuras en la obtención de contexto, es comprensible el éxito en el NLP de los transformadores, especialmente de modelos como GPT, que sin estar concretamente diseñados para la tarea de generación de código son capaces de realizar muestras de tanta calidad como la vista en la figura 5.7
- Generación de código compilable: A las limitaciones de contexto y dependencias se

añade las fuertes restricciones para generar código compilable, cualquier fallo en la posición, falta de token o falta de inclusión de las dependencias de librerías y archivos resultará en la imposibilidad de compilar el código. El procesamiento de lenguaje natural en bruto de la misma forma que se realiza con el lenguaje humano dificulta una tarea como esta, que tiene muchas más restricciones, aunque puede tener un gran potencial, es probable que necesite de herramientas externas o de pasos de procesamiento adicionales para refinar los resultados.

- Problema OOV: El problema *Out Of Vocabulary* (OOV) se produce cuando al tokenizar un elemento este no pertenece al vocabulario, tiene una relación directa con el problema del tamaño de vocabulario dado por la tokenización, especialmente al tomar una tokenización por palabra. A diferencia del lenguaje natural, el lenguaje de programación puede tener un tamaño indefinido puesto que no existe un lenguaje finito, esto afecta a todos los modelos en varios aspectos, por un lado, los modelos de caracteres son susceptibles a generar palabras nuevas inexistentes, los modelos por palabra solo pueden generar sobre el vocabulario que conocen, mientras que el discriminador será incapaz de reconocer palabras nuevas, clasificándolas siempre como desconocidas con el token [UNK] al procesarlas. Esto provoca que la calidad tanto de la generación como de la discriminación se vea comprometida si se considera únicamente el procesamiento por NLP, especialmente en modelos sin una magnitud ni datos masivos que les permitan tener un mayor contexto.

- Falta de datos de calidad: A diferencia de la generación de texto que encuentra en internet una fuente libre y prácticamente infinita de información para aprender, la generación de código es una tarea especial que aunque se fundamenta en el mismo procesamiento, necesita de un arduo trabajo de recopilación de código y está limitado por la dificultad de clasificar estos como vulnerables o no vulnerables, es por esto que uno de los mayores condicionantes es la falta de datasets de calidad y de un tamaño suficiente, especialmente en muestras vulnerables, puesto que además los pocos que existen presentan una gran desigualdad entre el número de muestras de cada clase. Sin la existencia de datasets de calidad, difícilmente se podrá obtener modelos prometedores.

Capítulo 6

Capítulo de Contribución

Antes de continuar con las conclusiones, exponemos en este capítulo las aportaciones y experiencias obtenidas por los miembros sobre el proyecto. Cada sección siguiente corresponde a las aportaciones de cada miembro en particular sobre el trabajo.

6.1. David Candil Villacastín

Comencé a abordar el trabajo partiendo por las bases. Tras dejar claro el objetivo del trabajo y entender en qué debíamos centrar nuestra investigación para obtener el producto final, que era realizar sistemas IAG que fuesen capaces de engañar a otro sistema como discriminador, mi primer objetivo fue entender las bases de esta rama de la informática, la Inteligencia Artificial, así como el *Machine Learning* y consecuentemente el *Deep Learning*. Partía desde el total desconocimiento más allá de la propia cultura informática y lo que se puede suponer de cómo es el desarrollo de modelos de aprendizaje automático, comencé por las bases de cada concepto que debía obtener, estudié diversos papers sobre trabajos similares que pudieran darme una idea del desarrollo que iba a encontrar y aprendí los conceptos teóricos que fundamentan el *Deep Learning* gracias al curso anual gratuito [MIT23] en vídeos que imparte el *Massachusetts Institute of Technology* (MIT).

Tras lo anterior, continué acercándome a la implementación de estos conceptos aprendiendo a manejar las herramientas necesarias que necesitaríamos para desarrollar los modelos, así como seguí informándome y buscando trabajos que pudieran servir de utilidad para el desarrollo. Continué con la mejora de mis habilidades en Python y el aprendizaje de librerías sencillas con grandes capacidades de manipulación de datos como son Numpy o Pandas, esenciales en el desarrollo de Machine Learning. Mientras que aprendí sobre todos estos conceptos estudiaba el núcleo de implementación de nuestro trabajo, el API Keras de *TensorFlow*, desarrollado por Google. Este API proporciona un interfaz de desarrollo de Machine Learning potente y moderno con el que se pueden crear y experimentar todo tipo de modelos y técnicas que se deseen. Tras obtener una base e idea de los conceptos teóricos, gracias a *TensorFlow* y los tutoriales sobre Machine Learning que se proporcionan en su página, pude ver en detalle y de forma práctica cómo funcionan la mayoría de modelos y redes internamente, por ejemplo, las redes RNN, CNN o arquitecturas GAN.

Continúe junto a mi compañero en la búsqueda de un dataset válido para el objeto de nuestro trabajo así como desarrollamos los modelos que exponemos en el trabajo, estudiando la mejor forma de estandarizar, tokenizar y vectorizar las muestras para

obtener resultados. Tras establecer el dataset, traté de aplicar nuestras conclusiones sobre el tratamiento de muestras y desarrollar los modelos generativos basados en la siguiente predicción. Como siguiente nivel en el desarrollo y aprendizaje de los conceptos, estudié la estructura detrás de las GAN y su implementación con el ya familiar API Keras, estudié ejemplos aplicados a la generación de imágenes, los cuales son muy comunes en la aplicación de la arquitectura dados sus buenos resultados. Tratamos de aplicar este concepto a la generación de código vulnerable, aunque desarrollamos una GAN completa, tras dedicar mucho tiempo a tratar de resolver un error producido por la retropropagación del gradiente de la pérdida, no fue incluida en el trabajo.

Durante el trabajo y el desarrollo de los modelos, dediqué parte de este tiempo a la redacción de esta memoria de proyecto y a tratar de garantizar que los conceptos y trabajos expuestos en esta son claros y no dan lugar a confusión.

6.2. Sergio Lorente Bausela

En un primer momento, aunque teniendo mucho conocimiento previo sobre Python, la seguridad informática o la IA, decidí que debía repasar y profundizar en los conceptos que había aprendido mediante la lectura de artículos y la práctica con sistemas simples, de modo que los siguientes conceptos e ideas que aprendiese ya estuvieran más relacionados con el TFG.

Una vez comprendidas las bases, centré mis esfuerzos en la lectura y comprensión de papers, con el objetivo de entender los conceptos básicos y, posteriormente, avanzados en la generación de código mediante el uso de los distintos tipos de modelos que habían sido desarrollados. Esta primera fase supuso tres etapas: una primera enfocada en la recopilación y lectura de estudios sobre los distintos tipos de aprendizaje que se pueden aplicar al entrenamiento de una IA, llegando a la conclusión del uso de un DL debido a la naturaleza misma del problema que queríamos solucionar; la segunda etapa consistió en la comprensión del funcionamiento de los modelos utilizados para la detección de patrones, lo que me ayudó a entender cuál era el modelo más adecuado para la detección de vulnerabilidades en código; por último, la tercera fase consistió en el estudio de la generación de texto, más específicamente, en la generación de código, en la cual me centré en un principio en los modelos GAN y de tokenización, aunque luego pivoteé a modelos con preentrenamiento como son GPT, BERT o *Text-to-Text Transfer Transformer* (T5).

Tras toda la investigación, mis esfuerzos se centraron en la generación de código y la detección de vulnerabilidades. Una vez mi compañero y yo decidimos qué modelos desarrollar y qué pruebas hacer, comencé con el desarrollo y entrenamiento de los mismos, encontrando grandes problemas principalmente en el modelo GAN, el cual descubriría que no es el más adecuado para este tipo de problemas estructurados y secuenciales. Mi trabajo se centró principalmente en los modelos con preentrenamiento, donde estudié el comportamiento que tienen sin un entrenamiento posterior y con él, de forma que pudiese averiguar si eran tipos de modelos adecuados para lo que requeríamos.

Como último y tras la investigación y posterior desarrollo, empecé a plasmar todos los conceptos, resultados y conclusiones que había adquirido, de modo que se viese una conclusión en las ventajas y desventajas de cada modelo y cuáles consiguen mejores resultados.

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

En este trabajo hemos estudiado y planteado el uso de técnicas y métodos de *Deep Learning* para la generación de código vulnerable y también para la detección de este. Hemos podido estudiar el estado del arte actual y la eficacia de las técnicas para la generación de muestras con el procesamiento de lenguaje natural. Tras haber llevado a cabo diferentes experimentos y a la vista de los resultados y las limitaciones encontramos que, en primer lugar, faltan datasets abiertos al público sobre vulnerabilidades de código en condiciones para ser procesados por modelos de aprendizaje automático, especialmente que cumplan las condiciones para el aprendizaje de la generación de muestras. También sobre la compilación de código, a diferencia de un texto en una lengua hablada, el código de programación no permite fallos, ni el más mínimo. Los modelos que entrenamos dentro de nuestras posibilidades no alcanzan la suficiente complejidad para generar código compilable sin aplicar transformaciones posteriores, aunque vemos un enfoque prometedor en la generación de muestras de código, en especial para los modelos con tokenización por caracteres. Los modelos con tokenización por palabras también pueden lograr grandes resultados, aunque necesitan de una capacidad computacional mucho mayor al tener una relación directa con el tamaño del vocabulario. Consideramos que el procesamiento en bruto de las muestras por medio de NLP es una ardua tarea, que aunque puede lograrse, es probable que la combinación de diversos sistemas y procesos previos y posteriores a la generación de muestras de unos resultados con potencial y de gran interés.

Por último, ya que en este trabajo también se realiza un sistema de detección de vulnerabilidades, observamos también que la clasificación es una tarea más sencilla que la generación, puesto que es independiente de las fuertes restricciones que imponen las normas de programación y puede centrarse con mayor facilidad en encontrar las características en común entre los datos y la distribución de cada grupo de elementos a clasificar. No obstante, también es susceptible a problemas como el problema OOV en función del procesamiento que se aplique.

7.2. Trabajo Futuro

Tras el desarrollo del trabajo y a la vista de los resultados, enumeramos las siguientes propuestas como posibles líneas de investigación al trabajo de interés. Estas ideas son fruto

de las limitaciones que se han comentado anteriormente y las conclusiones que sacamos de ellas, así como de las posibilidades que hubiéramos deseado poder explorar.

- **Realizar una clasificación multiclase**: Actualmente, los modelos presentados son capaces de detectar cuando un código es vulnerable, pero no han sido entrenados para categorizar entre los distintos tipos de vulnerabilidades. En futuras versiones, los modelos podrían ser capaces generar muestras según el tipo de vulnerabilidad indicado y el discriminador de identificarlas según el tipo de vulnerabilidad, pudiendo asociarla a un CVE específico o a la clasificación que se desee según los datos de entrada, por ejemplo: clasificar muestras en CVE-2023-12345 (inyección SQL) o CVE-2022-6789 (desbordamiento de búfer), entre otras posibilidades.

- **Modificar la generación de muestras**: Actualmente los modelos se entrenan procesando datos en lenguaje natural y generan muestras de forma espontánea, proponemos explorar los resultados que se obtendrían en la generación y el estudio de su calidad aplicando diferentes métodos como el enmascaramiento de tokens sin significado léxico como son los nombres de funciones personalizados o variables, también técnicas de postprocesamiento de las muestras para corregir los pequeños fallos que devuelve el modelo o controlar el final de las muestras de alguna forma, como por ejemplo, añadir un token especial para indicar el final de las muestras como se realiza en modelos transformadores.

- **Crear y entrenar estructuras de transformadores de forma nativa**: Los transformadores son, en esencia, traductores. Son modelos capaces de obtener una transformación de unos datos de entrada. Es probable que el mayor desafío fuese encontrar un dataset que se ajustase a nuestro objetivo. Actualmente no existen muchos datasets sobre la recopilación de muestras de código vulnerable, y menos aún que estas muestras contengan alguna característica con la que poder asociar la traducción secuencia a secuencia, como podría ser una explicación en detalle de la vulnerabilidad. Si entrenásemos un transformador capaz de traducir una petición de generación de código vulnerable, podríamos hacer que este reciba en el entrenamiento muestras compuestas por un mensaje de explicación en detalle como entrada y la vulnerabilidad como etiqueta. Proponemos estudiar si es posible entrenar un generador que nos de ejemplos de distintas vulnerabilidades si se le indica únicamente el tipo de vulnerabilidad.

- **Probar diferentes entrenamientos y datos**: Al finalizar el proyecto quisimos probar el entrenamiento sobre otro dataset de interés: Draper VDISC. En el caso en el que se contase con los recursos y el tiempo necesarios para un entrenamiento sobre un conjunto de mayor tamaño y/o datos de distintas fuentes como lo es este, sería interesante analizar si observamos que los modelos logran un mejor desempeño así como una mejora en la generalización del aprendizaje sobre el conjunto de entrenamiento.

- **Tokenizar con subpalabras**: En este trabajo hemos explorado la tokenización por diferentes niveles, en caracteres y en palabras, queda explorar un enfoque intermedio, la tokenización por subpalabras, las subpalabras divide cada palabra en pequeñas unidades compuestas por los segmentos con mayor frecuencia de la palabra (Ej. La palabra *Automatización* se convierte en 'auto', 'matiza' y 'ción') y aplica una serie de prefijos y sufijos con significado especial que relacionan estas subpalabras. Este enfoque es aplicado por modelos como GPT o BERT por medio de la aplicación de un

algoritmo interno durante la tokenización para extraer los segmentos más comunes de las palabras del dataset.

- **Desarrollo de una red GAN**: Las redes generativas adversarias son un tipo particular de estructuras de aprendizaje automático, se componen de dos modelos, un generador de muestras y un discriminador de estas, que tratan de jugar un juego de suma cero. El objetivo del generador es producir muestras tan similares las reales que el discriminador no pueda diferenciarlas, de modo que será recompensado si lo consigue, por otra parte, el objetivo del discriminador es detectar estas muestras y será recompensado de la misma manera, ambos convergen idealmente en un punto en el que el generador y discriminador están igualmente entrenados, de esta forma se obtiene un generador con capacidad de producir muestras bien ajustado. Proponemos el estudio del desarrollo de técnicas adversarias para la generación de muestras de código.

- **Arquitecturas híbridas**: La idea de estas arquitecturas consiste en la unión de varios modelos y/o programas que se complementen, aprovechando así las fortalezas de ambas técnicas. Un ejemplo sería la creación de una arquitectura híbrida transparente al usuario entre un modelo generativo y una red neuronal (o un sistema no necesariamente de inteligencia artificial), de modo que mediante el modelo fuésemos capaces de crear muestras de calidad y que estas fueran refinadas y mejoradas por un sistema especializado en eliminar los problemas que presentan las muestras generadas en bruto para que sean prácticamente compilables en la mayoría de los casos, como pueden ser la falta de cierre de llaves y paréntesis, la falta de puntos y comas y otros elementos esenciales en la sintaxis, la inclusión de las importaciones necesarias para el funcionamiento, etc.

Capítulo 8

Introduction

8.1. Motivation

Nowadays, we cannot ignore that *Artificial Intelligence* (AI) is one of the emerging fields in technology that will have the greatest impact on society in the coming years and, in fact, is already doing so. The rapid advancement of technology enables the creation of tools that were once thought impossible or even inconceivable. In particular, the progress of computer systems and all things related to them does not cease; every day, the boundaries of what they are capable of are redefined.

The ML emerges in the context of the information society, an era in which communication and digitalization have transformed everything we know. This information is, therefore, a central part of our lives; digitalization causes and allows almost everything to be stored and interpreted, comprehensible to a machine. It is here that AI comes into play. We can see how these systems, combined with information and proper processing, can be applied to all fields of human knowledge, enabling evaluations based on this data and even producing outputs, such as creating images of real people, drawing paintings in the style of renowned artists, autonomously operating vehicles, or producing robots that communicate in our natural language.

We find AI in an unimaginable variety of applications. Not straying from its natural domain, we consider its role in information system security as one of the key roles in its future advancements—for example, acting as a classifier and detector of malware, analyzing real-time images, or controlling network traffic. However, AI is still just an algorithm that executes discrete instructions based on input data, making it susceptible to attacks and vulnerabilities.

The goal of this work is the research, subsequent implementation, and use of different techniques for generating security attacks on a system based on the natural language processing of vulnerable source code, as well as the subsequent analysis of the results and conclusions obtained from the experiments.

8.2. Context

This Final Degree Project is part of a research project titled *Platform for Analysis of Resilient and Secure Software – LAZARUS*, approved by the European Commission

under the Horizon Framework Program (HORIZON-CL3-2021-CS-01 call) under grant agreement number 101070303. The project involves the GASS Group at the Complutense University of Madrid (Group for Analysis, Security, and Systems, https://gass.ucm.es, group 910623 from the catalog of research groups recognized by the UCM).

In addition to the Complutense University of 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 (Bulgaria), Binare Oy (Finland), Fundación APWG European Union Foundation (Spain), and Maggioli Spa (Italy).

Further information can be found at:

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

https://lazarus-he.eu

8.3. Research Objective

The objective of our work is the study of the concepts of ML and DL, particularly those necessary to understand *Generative Artificial Intelligence* (GAI) and apply it to create different generative models to compare their performance in this task. Following the research into this subject, we will design several models capable of generating samples of vulnerable code to deceive another ML discriminator system as a baseline, and we will analyze the performance of these systems' results and the limitations encountered in the approaches explored to achieve the task.

8.4. Work Plan

The development of this work has been carried out in three phases:

1. **Research**: The work began with a meeting with the research team in mid-July 2023, where we were informed about the objectives, the milestones to follow during development, and the competencies we would need to carry it out. For convenience, we agreed to formally start the work in September. From that point on, we held weekly meetings to address questions and track progress. Four months were spent researching the project's objectives and acquiring the basic competencies needed, as we both started with little knowledge of *Automatic Learning* (AL). This lack of knowledge was, in fact, a motivating factor in choosing this work to learn about this field of computer engineering. We began by reading various articles related to the state of the art in AL for detecting and generating code vulnerabilities and exploring approaches taken in related works. We summarized these works to better extract ideas and have information available for future use. We dedicated several weeks to gathering and extracting information. Afterward, we started learning how to develop simple models and familiarize ourselves with developing machine learning models using libraries like Google's *TensorFlow*. Progress was recorded in a shared repository where each member uploaded their findings, which were later discussed collectively.

2. **Development**: The development of the models was structured in several iterative phases. First, we began by searching for and analyzing valid datasets suited to our objectives. Following this, we implemented different generative models, refining them gradually to achieve the best combination. We also developed a discriminator model capable of predicting whether a code sample was vulnerable, serving as a performance baseline for our generators. Different combinations and machine learning models were tested during the development to achieve the best discriminator. The initial models developed were generators of vulnerable samples based on predicting the next sequence. Later, more complex models like LLM based on transformers were explored and developed. We also worked on developing a GAN network; however, we encountered challenges in backpropagating the loss of the generator network, which prevented us from fully training and quantifying its performance. Nonetheless, the greater complexity of this technique allowed us to gain in-depth knowledge of AL algorithms and their development.

3. **Experimentation**: Once the implementation and training of the different models were completed and prototypes were obtained from the previous phase, we concluded with the analysis of the results obtained and the derived metrics concerning the discriminator. We enumerated the limitations encountered in the techniques applied for code generation. Finally, we suggested potential extensions and continuations of the work based on the development carried out in this project.

8.5. Work Structure

The rest of the work is organized into chapters with the following structure:

Chapter 2 introduces fundamental concepts to understand AI, such as ML, describing its algorithms and categorization. It delves deeper into the DL branch of ML, focused on the development of neural networks. It explains in detail what these networks are, how they function, their traditional types, and the techniques available to optimize their performance. Finally, the concepts of software security and vulnerability analysis are introduced.

Chapter 3 presents techniques for detecting and generating code vulnerabilities. It provides a state-of-the-art review, citing the most relevant works in the field and discussing the most significant current techniques and concepts.

Chapter 4 outlines the project's contributions. It sequentially describes the development process followed to implement DL models for the task of generating and detecting vulnerabilities. It shows the application of NLP techniques to programming languages such as C and C++. It details the features of the techniques used and their implementation process.

Chapter 5 discusses the results obtained in classification with the discriminator model. Based on these results, it highlights the limitations encountered in developing such models and processing data for generating quality code.

Chapter 6 describes the contributions and work carried out by the project members, with each member indicating their specific contributions.

Chapter 7 concludes with the findings of this work and provides recommendations for extending or continuing future research lines based on the results obtained.

Chapters 8 and 9 provide English translations of Chapter 1: Introduction and Chapter 7: Conclusions.

Capítulo 9

Conclusions and Future Work

9.1. Conclusions

In this work, we have studied and proposed the use of *Deep Learning* techniques and methods for the generation of vulnerable code as well as for its detection. We have reviewed the current state of the art and the effectiveness of techniques for generating samples using natural language processing. After conducting various experiments and analyzing the results and limitations, we found that, first, there is a lack of publicly available datasets on code vulnerabilities that are suitable for processing by machine learning models, especially those meeting the requirements for training sample generation. Regarding code compilation, unlike text in a spoken language, programming code does not allow even the slightest errors. The models we trained within our possibilities do not reach sufficient complexity to generate compilable code without applying subsequent transformations. However, we see a promising approach in the generation of code samples, especially for character-based tokenization models. Word-based tokenization models can also achieve great results but require much greater computational capacity due to their direct relationship with vocabulary size. We consider that raw processing of samples using NLP is a challenging task that, while achievable, would likely yield promising and highly interesting results by combining various pre- and post-processing systems and techniques for sample generation.

Finally, since this work also develops a vulnerability detection system, we observed that classification is a simpler task than generation. This is because it is not subject to the strict constraints imposed by programming standards and can more easily focus on identifying common features among the data and the distribution of each group of elements to classify. However, it is also susceptible to issues such as the OOV problem, depending on the processing applied.

9.2. Future Work

After completing this work and analyzing the results, we outline the following proposals as potential research directions of interest. These ideas stem from the limitations discussed earlier and the conclusions drawn from them, as well as the possibilities we would have liked to explore.

- **Perform multiclass classification**: Currently, the presented models can detect when a piece of code is vulnerable, but they have not been trained to categorize different types of vulnerabilities. In future versions, the models could be capable of generating samples according to a specified type of vulnerability and detecting them based on that type, associating them with a specific CVE or a desired classification according to the input data. For example, classifying samples as CVE-2023-12345 (SQL injection) or CVE-2022-6789 (buffer overflow), among other possibilities.

- **Modify sample generation**: Currently, the models are trained on natural language data and generate samples spontaneously. We propose exploring the results that could be obtained by applying different methods, such as masking tokens with no lexical meaning (e.g., custom function or variable names), post-processing techniques to correct minor errors in the generated samples, or controlling the end of samples in some way, such as adding a special token to indicate the end of the samples as done in transformer models.

- **Develop and train native transformer architectures**: Transformers are, in essence, translators. They are models capable of transforming input data. The main challenge would likely be finding a dataset that matches our objective. Currently, there are few datasets on vulnerable code samples, and even fewer where these samples include characteristics suitable for sequence-to-sequence translation, such as detailed explanations of the vulnerabilities. Training a transformer capable of translating a request to generate vulnerable code might involve feeding it during training with samples consisting of detailed explanations as input and the vulnerability as the label. We propose studying whether it is possible to train a generator to provide examples of different vulnerabilities by specifying only the type of vulnerability.

- **Experiment with different training datasets**: At the end of the project, we wanted to test training on another dataset of interest: Draper VDISC. If sufficient resources and time were available for training on a larger dataset and/or data from various sources, such as this one, it would be interesting to analyze whether the models achieve better performance and improved generalization of learning on the training set.

- **Tokenization with subwords**: In this work, we explored tokenization at different levels, including characters and words. It remains to explore an intermediate approach: subword tokenization. Subwords divide each word into small units composed of the most frequent segments of the word (e.g., the word *Automation* becomes 'auto,' 'mation,' and 'tion') and apply a series of prefixes and suffixes with special meaning to relate these subwords. This approach is used by models like GPT and BERT by applying an internal algorithm during tokenization to extract the most common segments from the dataset's words.

- **Development of a GAN**: Generative Adversarial Networks are a specific type of machine learning architecture composed of two models: a sample generator and a sample discriminator, playing a zero-sum game. The generator's goal is to produce samples so similar to real ones that the discriminator cannot distinguish them, rewarding the generator if successful. Conversely, the discriminator's goal is to detect these samples and is rewarded similarly. Both ideally converge at a point where the generator and discriminator are equally trained, resulting in a well-adjusted

generator capable of producing high-quality samples. We propose studying the development of adversarial techniques for generating code samples.

- **Hybrid architectures**: These architectures combine multiple models and/or programs that complement each other, leveraging the strengths of both techniques. For example, a hybrid architecture transparent to the user could combine a generative model and a neural network (or a non-AI-based system). The generative model could create high-quality samples, which are then refined and improved by a specialized system to eliminate issues in raw samples, making them nearly compilable in most cases. Examples include fixing missing braces and parentheses, missing semicolons, and other essential syntax elements, as well as including necessary imports for functionality.

Bibliografía

[AIZ19] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. Juice: A large scale distantly supervised dataset for open domain context-based code generation. *CoRR*, abs/1910.02216, 2019.

[AV17] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. Attention is all you need. 2017.

[DDH+22] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *IEEE Access*, 10:82434–82455, 2022.

[Fer24] Josep Ferrer. Cómo funcionan los transformadores: Una exploración detallada de la arquitectura de los transformadores, 2024.

[GZ21] D. Grahn and J. Zhang. An analysis of c/c++ datasets for machine learning-assisted software vulnerability detection. In *Proceedings of the Conference on Applied Machine Learning for Information Security*, 2021.

[HMM+20] Mohammed H. Hassan, Omar A. Mahmoud, Omar I. Mohammed, Ammar Y. Baraka, Amira T. Mahmoud, and Ahmed H. Yousef. Neural machine based mobile applications code translation. In *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pages 302–307, 2020.

[HT17] Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on GAN. *CoRR*, abs/1702.05983, 2017.

[IJG14] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial networks. 2014.

[JF20] Shaohua Wang Tien N. Nguyen Jiahao Fan, Yi Li. A c/c++ code vulnerability dataset with code changes and cve summaries. 2020.

[LGR+21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. 2021.

[Min25] AI Mind. Rnns: The precursor of llms, 2025.

[MIT23] MIT. Intro deep learning, 2023.

[SABP19] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. Program synthesis and semantic parsing with learned code idioms. *CoRR*, abs/1906.10816, 2019.

[SFHC24] Christoforos Seas, Glenn Fitzpatrick, John A. Hamilton, and Martin C. Carlisle. Automated vulnerability detection in source code using deep representation learning. In *2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0484–0490, 2024.

[YGAR14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.