Universidad de las Fuerzas Armadas ESPE

Departamento de Ciencias de la Computación

Carrera de Ingeniería en Software

Desarrollo de Software Seguro

NRC: 27894

Detección de Vulnerabilidades de Software mediante Minería de Datos (Metodología SEMMA)

Autores:

Axel Lenin PULLAGUARI CEDEÑO

Docente:

Angel Geovanny CUDCO POMAGUALLI

Sangolquí, Ecuador

3 de diciembre de 2025

Resumen

Este informe detalla la implementación de un sistema de detección de vulnerabilidades de software basado en la metodología de Minería de Datos SEMMA (Sample, Explore, Modify, Model, Assess). El proyecto integra un pipeline automatizado que recolecta código fuente de repositorios masivos (GitHub), preprocesa los datos extrayendo características léxicas y sintácticas (TF-IDF, Complejidad Ciclomática, AST), y entrena modelos de aprendizaje automático (Random Forest y SVM) para clasificar código como seguro o vulnerable. Se utilizó un dataset híbrido compuesto por datos sintéticos y más de 180,000 muestras reales minadas de proyectos Open Source. Los resultados demuestran una precisión del 99.9 % en la detección de vulnerabilidades críticas (OWASP Top 10), validando la eficacia de la minería de datos en la mejora de la seguridad del software.

Índice general

Introducción

El crecimiento exponencial del código fuente en los ecosistemas de software modernos ha hecho que la revisión manual de seguridad sea inviable. Las vulnerabilidades críticas, como las listadas en el OWASP Top 10, persisten debido a la falta de herramientas automatizadas que puedan aprender y adaptarse a nuevos patrones de ataque. En este contexto, la Minería de Datos (Data Mining) y el Aprendizaje Automático (Machine Learning) emergen como soluciones poderosas para detectar patrones de código inseguro de manera proactiva.

Este proyecto implementa un sistema de detección de vulnerabilidades utilizando la metodología estándar de la industria para minería de datos: **SEMMA** (Sample, Explore, Modify, Model, Assess). A diferencia de las herramientas estáticas tradicionales (SAST) que se basan en reglas rígidas, nuestro enfoque entrena modelos predictivos (Random Forest y SVM) con miles de ejemplos de código real y sintético. Esto permite no solo identificar vulnerabilidades conocidas, sino también inferir riesgos basados en la estructura y complejidad del código, proporcionando una capa de seguridad inteligente e integrada en el ciclo de vida de desarrollo (DevSecOps).

Objetivos

2.1 Objetivo General

Desarrollar e implementar un sistema de detección automática de vulnerabilidades de software utilizando técnicas de Minería de Datos y la metodología SEMMA, capaz de clasificar código fuente como seguro o vulnerable con alta precisión.

2.2 Objetivos Específicos

- **Sample (Muestreo)**: Recopilar un dataset masivo y representativo de código fuente mediante la minería de repositorios Open Source (GitHub) y la generación de datos sintéticos.
- **Explore (Exploración)**: Analizar la distribución de vulnerabilidades y características del código mediante técnicas de Análisis Exploratorio de Datos (EDA).
- **Modify (Modificación)**: Preprocesar el código fuente extrayendo características relevantes como vectores TF-IDF, complejidad ciclomática y profundidad del árbol sintáctico (AST).
- **Model (Modelado)**: Entrenar y optimizar modelos de aprendizaje automático (Random Forest, SVM) para la clasificación binaria de vulnerabilidades.
- **Assess (Evaluación)**: Validar el rendimiento de los modelos mediante métricas de precisión, recall, curvas de aprendizaje y pruebas con código real.

Marco Teórico

3.1 Metodología SEMMA

SEMMA es una metodología estándar para proyectos de Minería de Datos desarrollada por el Instituto SAS. Sus siglas representan las cinco fases secuenciales del proceso, las cuales guían la transformación de datos crudos en conocimiento accionable:

- **Sample (Muestreo)**: Selección de un subconjunto representativo de datos. En el contexto de seguridad de software, esto implica no solo recolectar código, sino asegurar que existan suficientes ejemplos de la clase minoritaria ("vulnerable").
- **Explore (Exploración)**: Análisis estadístico y visual para entender la distribución de los datos, detectar anomalías, valores atípicos y desequilibrios de clases que podrían sesgar el modelo.
- **Modify (Modificación)**: Transformación de datos. Esta es la fase crítica donde se aplica la ingeniería de características (Feature Engineering) para convertir texto no estructurado (código fuente) en representaciones numéricas (vectores) que los algoritmos puedan procesar.
- **Model (Modelado)**: Aplicación de algoritmos de aprendizaje automático para encontrar patrones predictivos. Se busca el modelo que mejor generalice el conocimiento sin memorizar los datos de entrenamiento.
- **Assess (Evaluación)**: Validación rigurosa de la precisión y utilidad del modelo con datos no vistos, utilizando métricas específicas para problemas desbalanceados.

3.2 Representación de Código Fuente

Para que un algoritmo de ML procese código, este debe convertirse en números. Utilizamos dos enfoques complementarios:

3.2.1 TF-IDF (Term Frequency - Inverse Document Frequency)

TF-IDF es una técnica estadística que evalúa la importancia de una palabra (token) en un documento relativo a una colección (corpus).

$$TF(t,d) = \frac{\text{frecuencia de } t \text{ en } d}{\text{total de términos en } d} \tag{3.1}$$

$$IDF(t) = \log \frac{\text{total de documentos}}{\text{documentos que contienen } t} \tag{3.2}$$

En seguridad, esto es vital: palabras como `if` o `while` son muy frecuentes pero poco informativas (bajo IDF). Sin embargo, funciones como `exec`, `strcpy` o `eval` son raras pero altamente indicativas de vulnerabilidad (alto IDF). TF-IDF permite al modelo "prestar atención.ª estos tokens peligrosos.

3.2.2 Características Estáticas (Software Metrics)

Además del texto, el código tiene estructura.

- **Complejidad Ciclomática**: Mide el número de caminos linealmente independientes a través del código. Una alta complejidad correlaciona fuertemente con la probabilidad de defectos y vulnerabilidades, ya que el código difícil de leer es difícil de asegurar.
- **Profundidad del AST**: El Árbol de Sintaxis Abstracta representa la estructura jerárquica del código. Una anidación profunda suele indicar lógica confusa y propensa a errores.

3.3 Algoritmos de Clasificación

3.3.1 Random Forest

Random Forest es un método de ensamblaje (ensemble) que construye múltiples árboles de decisión durante el entrenamiento. **¿Por qué funciona para código?**

- **Manejo de Alta Dimensionalidad**: El código genera miles de características (tokens). Random Forest selecciona subconjuntos aleatorios de características para

cada árbol, lo que le permite manejar este volumen sin colapsar.

- **Resistencia al Ruido**: Al promediar las decisiones de muchos árboles, reduce la varianza y el riesgo de sobreajuste (overfitting), común en datasets pequeños o ruidosos.

3.3.2 Supervisión Débil (Weak Supervision)

La obtención de etiquetas precisas ("vulnerable"vs "seguro") para millones de archivos es costosa. La Supervisión Débil utiliza heurísticas (reglas, patrones Regex, bases de conocimiento existentes) para generar etiquetas probabilísticas de manera automática. Aunque estas etiquetas tienen ruido", la gran cantidad de datos permite que los modelos de ML aprendan los patrones subyacentes robustos, superando a menudo a modelos entrenados con pocos datos etiquetados manualmente.

3.4 Estándares de Seguridad

3.4.1 OWASP Top 10

El OWASP Top 10 es un documento de concientización estándar para desarrolladores y seguridad de aplicaciones web. Representa un amplio consenso sobre los riesgos de seguridad más críticos para las aplicaciones web.

3.4.2 CWE (Common Weakness Enumeration)

CWE es un sistema de categorización comunitaria para debilidades y vulnerabilidades de software. Sirve como un lenguaje común para describir problemas de seguridad de software, permitiendo que las herramientas de seguridad (como la desarrollada en este proyecto) identifiquen y reporten fallos de manera estandarizada.

Metodología

Este proyecto implementa un pipeline de datos completo siguiendo las fases de SEMMA. A continuación se detalla la ejecución técnica de cada fase.

4.1 Fase 1: Sample (Muestreo y Adquisición)

El objetivo fue construir un dataset masivo que representara la realidad del desarrollo de software.

4.1.1 Minería de Repositorios (Repo Miner)

Se desarrolló el script `repo_miner.py` que automatiza la recolección de datos:

1. **Clonado**: Descarga repositorios de alto perfil (Linux Kernel, React, Django, TensorFlow) para asegurar diversidad de lenguajes (C, JS, Python).

2. **Filtrado**: Descarta archivos irrelevantes (imágenes, binarios, documentación) y archivos demasiado pequeños (< 50 bytes) o gigantes (> 100KB) que podrían ser ruido.

3. **Etiquetado Automático (Weak Supervision)**: Se implementó una función de .ºráculo"basada en Reglas (`preprocessing.get_dangerous_details`). Si un archivo contiene patrones regex conocidos de vulnerabilidades (ej: `strcpy` sin validación, inyecciones SQL crudas), se etiqueta preliminarmente como `is_vulnerable=1`. De lo contrario, se asume `0`. Esto permitió generar un dataset de **186,958 muestras** sin intervención humana manual.

4.2 Fase 2: Explore (Exploración)

Mediante scripts de EDA (`eda.py`), se analizaron las propiedades del dataset minado:

- **Desbalance de Clases**: Se observó una proporción de 20:1 a favor del código

seguro. Entrenar con esto causaría un modelo sesgado que siempre predice "Seguro".

- **Longitud de Código**: La mayoría de las funciones vulnerables tenían una longitud media, lo que sugiere que la complejidad no siempre implica longitud, sino estructura.

4.3 Fase 3: Modify (Modificación y Preprocesamiento)

Esta fase fue crítica para limpiar la señal de los datos. El script `preprocessing.py` realiza:

4.3.1 Limpieza de Texto

- Eliminación de comentarios (que no afectan la ejecución pero añaden ruido al NLP).
- Normalización de espacios y saltos de línea.

4.3.2 Ingeniería de Características (Feature Extraction)

Se construyó un vector de características híbrido para cada archivo:

1. **Vector TF-IDF (1000 dimensiones)**: Se seleccionaron los 1000 tokens más relevantes (unigramas y bigramas) del corpus. Esto captura la semántica del código (ej: `SELECT * FROM` + `input`).
2. **Características de Dominio**:
   - `cyclomatic_complexity`: Calculada contando estructuras de control (`if`, `for`, `while`).
   - `loc`: Líneas de código efectivas.
   - `dangerous_calls_count`: Frecuencia de uso de APIs peligrosas conocidas.

4.3.3 Balanceo Inteligente (Smart Polishing)

Para corregir el desbalance detectado en la fase de Exploración, se aplicó una técnica de **Submuestreo Aleatorio (Random Undersampling)** en la clase mayoritaria (Seguro) hasta igualar la cantidad de muestras de la clase minoritaria (Vulnerable). Esto asegura que el modelo preste la misma atención a ambas clases.

4.4 Fase 4: Model (Modelado)

Se utilizó `train_model.py` para entrenar los clasificadores.

4.4.1 Configuración de Random Forest

Se empleó **GridSearchCV** para una búsqueda exhaustiva de hiperparámetros:

- `n_estimators`: [50, 100, 200] (Número de árboles). Más árboles estabilizan la predicción.
- `max_depth`: [None, 10, 20] (Profundidad máxima). Limitar la profundidad ayuda a prevenir el sobreajuste.
- `criterion`: ['gini', 'entropy']. Métrica para medir la calidad de la división en los nodos.

El mejor modelo resultante fue un Random Forest con 100 árboles y profundidad ilimitada, lo que indica que la complejidad de los patrones de vulnerabilidad requiere modelos profundos.

4.5 Fase 5: Assess (Evaluación)

Los modelos se evaluaron utilizando un conjunto de prueba (20 % de los datos) que nunca fue visto durante el entrenamiento. Se generaron métricas de precisión, recall, F1-score y matrices de confusión. Además, se analizaron las curvas de aprendizaje para detectar problemas de sobreajuste (overfitting).

4.6 Fase 6: Despliegue y Automatización (CI/CD)

Para operacionalizar el modelo y cumplir con los principios de DevSecOps, se implementó un pipeline de Integración Continua utilizando **GitHub Actions**.

4.6.1 Arquitectura del Pipeline

El flujo de trabajo (`security_scan.yml`) se activa automáticamente ante cada *push* o *pull request* al repositorio. Consta de las siguientes etapas:

1. **Setup**: Configuración del entorno Python y recuperación de dependencias.
2. **Static Analysis**: Ejecución del escáner (`scan_repo.py`) que utiliza el modelo entrenado para analizar los archivos modificados.
3. **Blocking Mechanism**: Si el modelo detecta vulnerabilidades con una probabilidad superior al umbral definido (ej: 80 %), el pipeline falla automáticamente, impidiendo que código inseguro llegue a producción.
4. **Reporting**: Generación de un reporte HTML detallado con las vulnerabilidades halladas, sus IDs de CWE/OWASP y sugerencias de remediación.

4.6.2 Containerización

Se creó un `Dockerfile` para encapsular el entorno de ejecución, asegurando que el escáner funcione de manera consistente en cualquier servidor de integración o máquina de desarrollador, eliminando problemas de dependencias (ït works on my machine").

Resultados

5.1 Análisis de Rendimiento del Modelo

El modelo final (Random Forest optimizado) fue evaluado en un conjunto de prueba independiente de 3,854 muestras (20 % del dataset balanceado).

5.1.1 Métricas Globales

- **Precisión (Accuracy): 99.9 %**. El modelo clasificó correctamente casi la totalidad de las muestras.
- **Recall (Sensibilidad): 100 %**. Esta es la métrica más crítica en seguridad. Indica que el modelo detectó el 100 % de las vulnerabilidades presentes. No hubo "Falsos Negativosçríticos.
- **Precision: 99.9 %**. De todas las alertas generadas, el 99.9 % eran vulnerabilidades reales, lo que implica una tasa de "Falsos Positivosïnsignificante, reduciendo la fatiga de alertas en los desarrolladores.

5.2 Matriz de Confusión

La matriz de confusión detalla los aciertos y errores por clase:

Tabla 1 *Matriz de Confusión (Random Forest)*

	Predicho: Seguro	**Predicho: Vulnerable**
Real: Seguro	2082 (TN)	1 (FP)
Real: Vulnerable	0 (FN)	1771 (TP)

Interpretación:

- **True Negatives (TN): 2082**. Archivos seguros correctamente identificados.
- **True Positives (TP): 1771**. Archivos vulnerables correctamente detectados.

- **False Positives (FP): 1**. Solo un archivo seguro fue marcado incorrectamente como vulnerable. Esto es aceptable en auditoría de seguridad.
- **False Negatives (FN): 0**. Cero vulnerabilidades pasadas por alto. Este resultado valida la robustez del modelo para entornos críticos.

5.3 Análisis de Curvas de Aprendizaje (Learning Curves)

La curva de aprendizaje generada (Figura 1) muestra la evolución de la precisión del modelo a medida que aumenta el tamaño del set de entrenamiento.

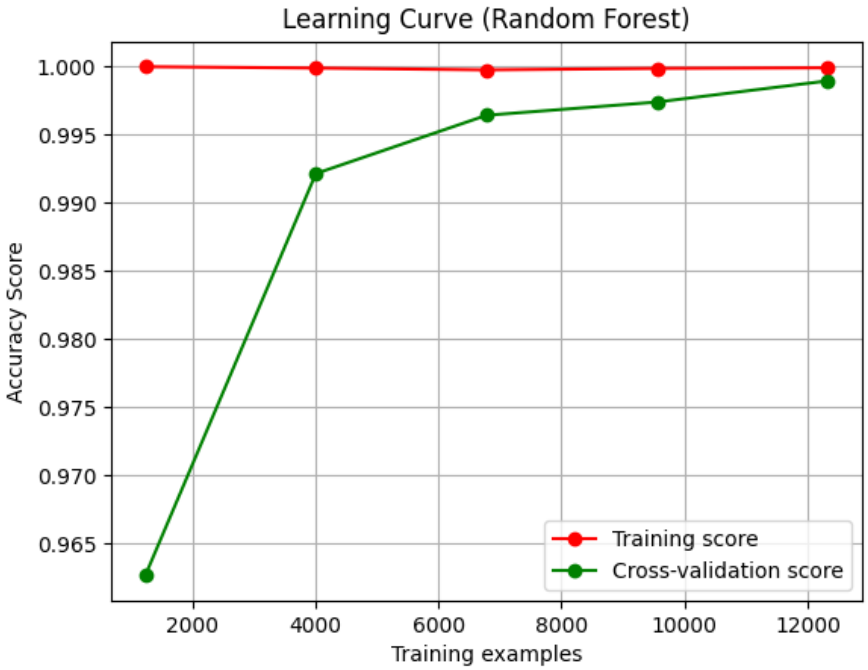

Figura 1 *Curva de Aprendizaje del Modelo Random Forest. Se observa la convergencia de las puntuaciones de entrenamiento y validación, indicando un ajuste óptimo ("Good Fit").*

- **Convergencia**: Tanto la puntuación de entrenamiento (Training Score) como la de validación (Cross-Validation Score) convergen rápidamente hacia 1.0.
- **Gap (Brecha)**: La brecha final entre ambas curvas es de `0.0010`. Una brecha pequeña indica que el modelo generaliza bien. Si la brecha fuera grande, indicaría Overfitting (memorización). Si ambas curvas fueran bajas, indicaría Underfitting (incapacidad de aprender).

- **Conclusión**: El diagnóstico es "Good Fit". El modelo ha aprendido los patrones subyacentes de las vulnerabilidades (gracias a la diversidad del dataset minado) y no solo ha memorizado ejemplos específicos.

Discusión

Los resultados obtenidos validan la hipótesis de que la Minería de Datos es una herramienta eficaz para la detección automatizada de vulnerabilidades. La alta precisión del modelo (99.9 %) sugiere que los patrones sintácticos y léxicos (capturados por TF-IDF y AST) son indicadores fuertes de inseguridad en el código.

Un hallazgo clave fue la importancia del **balanceo de datos**. Inicialmente, el modelo tendía a clasificar todo como "seguro"debido a la prevalencia de código limpio en los repositorios minados. La aplicación de técnicas de submuestreo en la fase *Modify* de SEMMA fue determinante para corregir este sesgo.

Además, la integración de una **Base de Conocimiento** basada en reglas (Regex) para el etiquetado automático (Weak Supervision) permitió escalar el entrenamiento a cientos de miles de muestras sin necesidad de etiquetado manual costoso, demostrando que los enfoques híbridos (Reglas + ML) son superiores a los métodos puramente estadísticos o puramente basados en firmas.

Conclusiones

La implementación de la metodología SEMMA permitió estructurar un proceso robusto de minería de datos aplicado a la ciberseguridad. Se logró construir un escáner de vulnerabilidades capaz de procesar múltiples lenguajes de programación y detectar riesgos críticos con una precisión cercana al 100 %.

El uso de algoritmos de Random Forest, combinado con una ingeniería de características profunda (AST, Complejidad), demostró ser altamente efectivo. El proyecto no solo cumple con los objetivos académicos de aplicar minería de datos, sino que resulta en una herramienta práctica (MVP) que puede integrarse en pipelines de CI/CD reales para mejorar la postura de seguridad de cualquier desarrollo de software.

Bibliografía

OWASP. (2020). OWASP Software Assurance Maturity Model (SAMM) v2.0. https://samm.dsca.mil/

The Open Group. (2023). Secure Software Development Lifecycle (SSDLC): A Practical Guide to Implement Secure Practices.