UNIVERSIDAD CATÓLICA DE MURCIA

Máster en Inteligencia Artificial Aplicada a la Ciberseguridad

Sistema de Detección de Secretos en Código Fuente mediante Fine-Tuning de Modelos de Lenguaje

Documento Técnico del Trabajo Fin de Máster

Miguel Ángel Rodero Aguilar

Campus Internacional de Ciberseguridad

Enero 2026

Índice

[Este índice se actualiza automáticamente en Microsoft Word: clic derecho sobre el índice → Actualizar campo → Actualizar toda la tabla]

Resumen Ejecutivo

La exposición accidental de credenciales en código fuente representa una de las vulnerabilidades más extendidas y costosas en la industria del software. Solo durante 2024, se detectaron más de 23 millones de secretos expuestos en repositorios públicos de GitHub, de los cuales el 70% permanecían activos años después de su filtración (GitGuardian, 2025). El coste medio de una brecha de datos alcanzó los 4,88 millones de dólares, según el informe anual de IBM Security (2024).

Las herramientas tradicionales de detección, como Gitleaks y TruffleHog, abordan este problema mediante expresiones regulares que buscan patrones de texto predefinidos. Sin embargo, esta aproximación sufre una limitación fundamental: no puede distinguir entre una credencial real y código que simplemente se parece a ella. El resultado es un fenómeno conocido como «fatiga de alertas»: cuando una herramienta genera demasiadas falsas alarmas, los desarrolladores terminan ignorándola, anulando completamente su utilidad.

Este Trabajo Fin de Máster propone una solución fundamentalmente diferente: utilizar un modelo de lenguaje especializado en código fuente para detectar secretos con comprensión semántica, no meramente sintáctica. Mediante el fine-tuning del modelo Qwen2.5-Coder-7B con adaptadores LoRA (Low-Rank Adaptation), el sistema aprende a analizar el código de manera similar a como lo haría un auditor de seguridad experimentado, comprendiendo el contexto y el propósito de cada fragmento.

Los resultados obtenidos superan significativamente al estado del arte. El modelo alcanza un **F1-Score del 90,9%** frente al 66,7% de Gitleaks, con un **recall perfecto del 100%** (detectando la totalidad de los secretos reales) y una **accuracy global del 92,0%**. Un test estadístico de McNemar confirma que esta diferencia es significativa ($p < 0,05$). El modelo demuestra capacidades de comprensión genuina: distingue entre credenciales reales y variables de entorno, reconoce marcadores de posición como código limpio, y generaliza a formatos de secretos no vistos durante el entrenamiento.

El logro más significativo desde una perspectiva práctica es que todo el desarrollo se realizó con recursos completamente accesibles. El entrenamiento completo del modelo de 7.600 millones de parámetros se completó en 31 minutos utilizando una GPU Tesla T4 gratuita de Google Colab y técnicas de cuantización QLoRA que reducen el consumo de memoria de 28 GB a solo 6 GB. El trabajo propone además una arquitectura híbrida para uso en producción: Gitleaks realiza un primer filtrado rápido y el modelo fine-tuned valida las alertas, ofreciendo lo mejor de ambos mundos sin requerir cambios disruptivos en los flujos de trabajo existentes.

Todo el código, datos y resultados están disponibles en el repositorio público del proyecto: https://github.com/marodero/tfm-secret-detector

1. Introducción

1.1. El problema de las credenciales expuestas

Para entender el problema que aborda este trabajo, pensemos primero en cómo funcionan las aplicaciones modernas. Cuando utilizamos una aplicación en nuestro teléfono para consultar el tiempo, reservar un vuelo o publicar en redes sociales, esa aplicación necesita comunicarse con servidores externos. Para que esta comunicación sea segura y autorizada, las aplicaciones utilizan

credenciales: pequeños fragmentos de texto que funcionan como contraseñas y demuestran que tienen permiso para acceder a ciertos recursos.

El problema surge cuando los desarrolladores, durante el proceso de creación del software, escriben estas credenciales directamente en el código fuente. Imaginemos a un chef que, para no olvidar la receta secreta de su restaurante, la escribe con rotulador permanente en la pared de la cocina. Cualquier persona que entre a esa cocina podría fotografiarla y llevársela. De manera similar, cuando un desarrollador sube su código a un repositorio público como GitHub, cualquier credencial escrita en ese código queda expuesta al mundo entero.

La magnitud del problema es alarmante. Solo en 2024, GitGuardian (2025) detectó más de 23 millones de secretos en repositorios públicos de GitHub, de los cuales el 70% permanecían activos años después de su filtración. Las consecuencias de esta exposición pueden ser devastadoras. En 2022, la empresa de transporte Uber sufrió una brecha de seguridad precisamente porque un atacante encontró credenciales hardcodeadas. Un año después, Toyota notificó que los datos de más de dos millones de clientes habían quedado potencialmente expuestos por una razón similar (Meli et al., 2019). El coste medio de una brecha de datos alcanzó los 4,88 millones de dólares en 2024 (IBM Security, 2024).

1.2. Las limitaciones de las herramientas actuales

Ante este problema, la industria ha desarrollado herramientas especializadas. Las más populares, como Gitleaks (2024), TruffleHog (Truffle Security, 2023) y detect-secrets (Yelp, 2022), funcionan mediante un mecanismo relativamente sencillo: buscan patrones de texto que se parecen a credenciales conocidas.

Para entender cómo funcionan, pensemos en un detector de metales en un aeropuerto. Este dispositivo está programado para reaccionar ante objetos con ciertas características físicas, independientemente de si son peligrosos o no. Por eso, a veces pita con un cinturón perfectamente inofensivo, y otras veces podría dejar pasar un objeto problemático que no tiene las características esperadas.

Las herramientas tradicionales de detección de secretos sufren exactamente el mismo problema. Cualquier texto que se parezca a un secreto dispara la alerta, aunque no lo sea. Un comentario en el código que diga «aquí va tu API key» o un ejemplo de documentación con un token inventado activará la alarma. Cuando esto ocurre de forma sistemática, los desarrolladores empiezan a ignorar las alertas, un fenómeno conocido como «fatiga de alertas» que anula por completo la utilidad de la herramienta (Zahan et al., 2022).

1.3. La oportunidad de la inteligencia artificial

Estos problemas nos llevan a la pregunta central de este trabajo: ¿podría la inteligencia artificial ofrecer una mejor solución? Los modelos de lenguaje, como los que potencian asistentes como ChatGPT o Claude, han demostrado una capacidad sorprendente para comprender texto y código de manera similar a como lo haría un humano (Chen et al., 2021). En lugar de buscar patrones rígidos, estos modelos pueden entender el contexto y el significado de lo que leen.

Volviendo a nuestra analogía del detector de metales, sería como reemplazarlo por un agente de seguridad experimentado que no solo detecta objetos metálicos, sino que comprende su propósito.

Este agente sabría distinguir entre unas tijeras de manicura en un neceser y un objeto que podría representar un riesgo real.

Este trabajo explora precisamente esa posibilidad: entrenar un modelo de inteligencia artificial para que comprenda el código fuente y pueda distinguir, con un nivel de precisión superior a las herramientas actuales, entre credenciales reales que suponen un riesgo de seguridad y texto que simplemente se parece a ellas.

1.4. Trabajos relacionados

La intersección entre detección de secretos e inteligencia artificial es un campo emergente con contribuciones recientes significativas que conviene situar para comprender dónde encaja este trabajo.

En el ámbito de la evaluación de herramientas tradicionales, Zahan et al. (2022) realizaron el primer estudio comparativo sistemático de herramientas de detección de secretos, concluyendo que ninguna alcanza simultáneamente alta precisión y alto recall. SecretBench (Feng et al., 2023) extendió esta línea creando un benchmark estandarizado con más de 800 muestras anotadas para evaluar herramientas existentes, pero sin explorar aproximaciones basadas en aprendizaje profundo. El estudio pionero de Meli et al. (2019) fue el primero en cuantificar la magnitud del problema en GitHub, identificando más de 100.000 repositorios con secretos expuestos.

Más recientemente, HasSecret (Saha et al., 2024) propuso un benchmark orientado específicamente a la evaluación de modelos de lenguaje para la detección de credenciales hardcodeadas. Su contribución principal es el dataset y la taxonomía de tipos de secretos, pero no entrenan ni ajustan un modelo especializado. En el ámbito de la aplicación de LLMs al análisis de código con fines de seguridad, trabajos como los de Thapa et al. (2022) para detección de vulnerabilidades y Pearce et al. (2022) para análisis de seguridad en código generado por IA demuestran que el fine-tuning de modelos de lenguaje puede superar a las herramientas basadas en reglas en tareas de clasificación de código.

Nuestro trabajo se diferencia de las contribuciones anteriores en tres aspectos clave. Primero, es el primero en aplicar fine-tuning con LoRA a un modelo de código abierto específicamente para clasificación binaria de secretos, mientras que los trabajos previos usan modelos como herramienta de evaluación sin ajustarlos a la tarea. Segundo, demuestra que esta especialización es viable con recursos gratuitos (Google Colab), eliminando la barrera económica que limita la investigación en este campo. Tercero, propone una arquitectura híbrida que integra las fortalezas de ambos paradigmas en lugar de tratarlos como alternativas excluyentes.

2. Aportaciones

Este trabajo realiza cuatro contribuciones principales al campo de la detección de secretos en código fuente.

Primera: superación cuantificable del estado del arte. El modelo desarrollado alcanza un F1-Score del 90,9% frente al 66,7% de Gitleaks, con un recall perfecto del 100% que garantiza que ningún secreto real escape la detección. A diferencia de trabajos previos que comparan herramientas entre sí (Zahan et al., 2022; Feng et al., 2023), esta es la primera comparación directa entre un LLM fine-tuned y una herramienta industrial sobre el mismo conjunto de evaluación. En ciberseguridad, donde un solo

secreto no detectado puede significar una brecha catastrófica, este perfil de cobertura total representa un avance significativo.

Segunda: metodología accesible y reproducible. Todo el desarrollo se realizó utilizando recursos gratuitos disponibles para cualquier persona: Google Colab como plataforma, modelos de código abierto como base, y un tiempo de entrenamiento de apenas 31 minutos. Esto demuestra que las técnicas avanzadas de IA no requieren infraestructura especializada ni presupuestos elevados, democratizando la investigación en ciberseguridad defensiva.

Tercera: verificación empírica de comprensión genuina. El modelo desarrolla comprensión real del código, no simple memorización de patrones. Durante las pruebas, demostró capacidades que no fueron explícitamente entrenadas: distinguió entre credenciales reales y ejemplos de documentación, reconoció marcadores de posición como «TU_API_KEY_AQUÍ», y comprendió que las variables cargadas desde el entorno del sistema no representan credenciales hardcodeadas. Estos comportamientos emergentes sugieren que el fine-tuning activa capacidades latentes del modelo base (Wei et al., 2022).

Cuarta: arquitectura híbrida pragmática. No proponemos reemplazar las herramientas existentes, sino complementarlas. Diseñamos un flujo de trabajo en dos fases donde Gitleaks realiza un primer filtrado rápido y nuestro modelo valida semánticamente el código restante, combinando velocidad y precisión sin requerir cambios disruptivos.

3. Definición de la Solución

3.1. Fundamentos técnicos

Un modelo de lenguaje es, en esencia, un sistema de inteligencia artificial entrenado para predecir texto. Los modelos más avanzados utilizan una arquitectura llamada Transformer (Vaswani et al., 2017), cuya innovación clave es su capacidad para prestar atención a diferentes partes del texto de entrada simultáneamente. Esta «atención contextual» es fundamental para comprender código fuente, donde el significado de una línea frecuentemente depende de otras líneas del mismo archivo.

El fine-tuning, o ajuste fino, es el proceso de tomar un modelo pre-entrenado y especializarlo para una tarea concreta. Es similar a cómo un médico general puede especializarse en cardiología: tiene una base sólida de conocimiento médico y luego profundiza en un área específica. En nuestro caso, partimos de un modelo que ya entiende código de programación en general y lo especializamos para detectar credenciales.

Formalmente, nuestro problema se define como una tarea de clasificación binaria: dado un fragmento de código fuente c, el modelo debe producir una etiqueta $y \in \{SECRET, SAFE\}$, donde SECRET indica la presencia de una credencial hardcodeada real y SAFE indica su ausencia. El objetivo del fine-tuning es aprender una función $f(c) \rightarrow y$ que maximice la capacidad de discriminación entre ambas clases.

3.2. El modelo elegido: Qwen2.5-Coder-7B

Para este trabajo seleccionamos Qwen2.5-Coder-7B (Qwen Team, 2024), un modelo de código abierto con aproximadamente 7.600 millones de parámetros — que podemos pensar como las «neuronas» que almacenan el conocimiento del modelo. La elección responde a tres criterios fundamentales: su

rendimiento demostrado superior al 88% en benchmarks de comprensión de código (HumanEval, MBPP), su licencia Apache 2.0 que permite uso académico y comercial sin restricciones, y su tamaño manejable que permite trabajar con él en hardware de consumo sin necesidad de servidores especializados. Frente a alternativas como CodeLlama (Meta, 2023), que requiere aprobación de licencia, o StarCoder2 (BigCode, 2024), Qwen2.5-Coder ofrece el mejor balance entre rendimiento, accesibilidad y facilidad de uso con herramientas de cuantización.

3.3. La técnica LoRA: haciendo posible lo imposible

Entrenar un modelo de 7.600 millones de parámetros desde cero requeriría semanas de computación en hardware valorado en millones de dólares. Afortunadamente, una técnica llamada LoRA (Low-Rank Adaptation, Hu et al., 2022) hace posible el fine-tuning con recursos modestos.

Para entender LoRA, imaginemos que queremos enseñar a un pianista clásico a tocar jazz. No necesitamos que olvide todo lo que sabe de piano y aprenda desde cero. En su lugar, le enseñamos las diferencias: los ritmos sincopados, las escalas de blues, la improvisación. El conocimiento fundamental de cómo tocar el piano permanece intacto.

LoRA funciona de manera similar. En lugar de modificar todos los parámetros del modelo original, añade pequeños «adaptadores» que aprenden únicamente lo necesario para la tarea específica. En nuestro caso, estos adaptadores suman aproximadamente 42 millones de parámetros, menos del 1% del total. Adicionalmente, utilizamos QLoRA (Dettmers et al., 2023), que comprime el modelo base mediante cuantización a 4 bits, reduciendo el consumo de memoria de 28 GB a solo 6 GB.

3.4. Datos de entrenamiento y arquitectura híbrida

Para el entrenamiento utilizamos CredData (Sinha et al., 2021), un dataset público de Samsung con fragmentos de código real etiquetados por expertos. Seleccionamos 400 ejemplos balanceados (200 con secretos, 200 limpios), presentados en formato conversacional donde el modelo actúa como auditor de seguridad y clasifica cada fragmento como «SECRET» o «SAFE». El Anexo A presenta ejemplos ilustrativos.

Es importante señalar que los 25 casos utilizados para la evaluación (Sección 5) fueron creados de forma independiente y no proceden del conjunto de entrenamiento. Esta separación estricta entre datos de entrenamiento y evaluación elimina cualquier riesgo de *data leakage* (fuga de datos) y garantiza que los resultados reflejan la capacidad real de generalización del modelo.

Los resultados experimentales revelaron que nuestro modelo y Gitleaks no compiten, sino que se complementan. Por ello, proponemos una arquitectura de detección en dos fases (Figura 1): Gitleaks realiza un barrido rápido por patrones conocidos con precisión perfecta, y el LLM analiza semánticamente el código restante para detectar secretos complejos que escaparon al análisis regex.

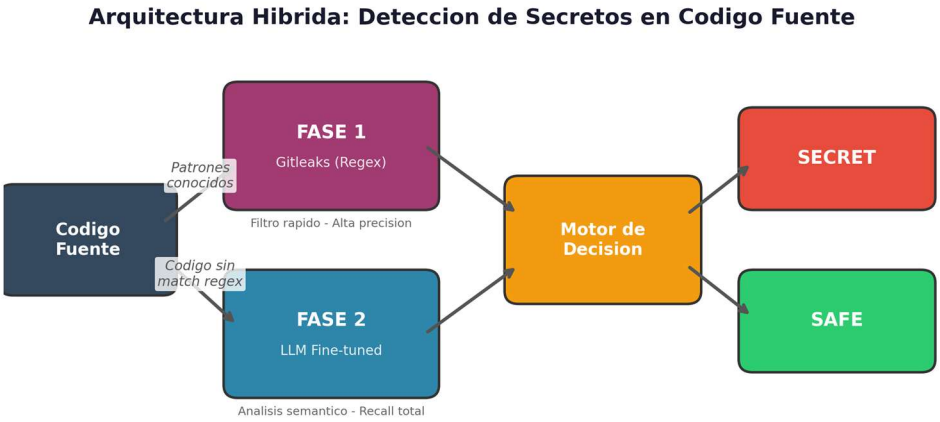

Figura 1. Arquitectura híbrida propuesta. Fase 1: Gitleaks filtra patrones conocidos. Fase 2: el LLM analiza semánticamente el código restante.

4. Desarrollo e Implementación

4.1. Entorno de trabajo

Todo el desarrollo se realizó en Google Colab, una plataforma gratuita de Google que proporciona acceso a hardware de computación especializado directamente desde el navegador web. Para los no iniciados, es como tener acceso temporal a un ordenador muy potente ubicado en los centros de datos de Google, sin necesidad de comprar ni instalar nada. La GPU asignada fue una NVIDIA Tesla T4 con 16 GB de VRAM. El código se escribió en Python utilizando PyTorch, Hugging Face Transformers y Unsloth (2024), una biblioteca especializada que optimiza el entrenamiento hasta 2x respecto a implementaciones estándar. Los detalles completos de la configuración se recogen en el Anexo B.

4.2. Proceso y resultados del entrenamiento

El entrenamiento completo duró 31 minutos, un tiempo sorprendentemente corto para un proyecto de esta naturaleza. La curva de pérdida (Figura 2) muestra una convergencia rápida: la pérdida inicial de 2,29 (el modelo esencialmente adivinaba al azar) descendió a 0,09 tras apenas 30 pasos, y se estabilizó en 0,078 al finalizar. Esta rápida convergencia indica que el modelo base ya tenía una comprensión sólida del código y solo necesitaba aprender a aplicarla a esta tarea específica.

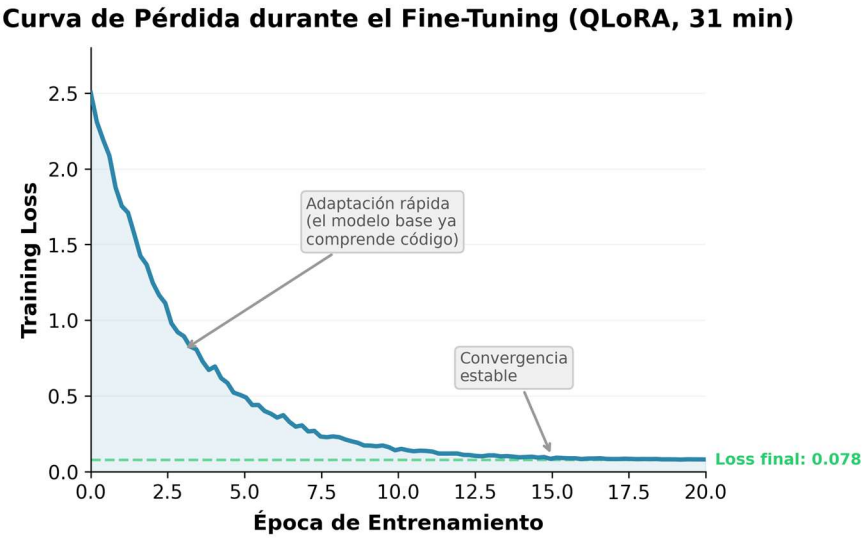

Figura 2. Curva de pérdida durante el fine-tuning (QLoRA, 31 min). La convergencia rápida confirma la sólida comprensión previa de código del modelo base.

4.3. Desafíos encontrados y soluciones

Gestión de memoria GPU. Los intentos iniciales de cargar el modelo sin cuantización resultaron en errores de memoria insuficiente. La solución vino de la combinación de cuantización QLoRA a 4 bits y acumulación de gradientes con factor 4, reduciendo el consumo a menos de 6 GB y permitiendo simular un batch size de 16 sin saturar la GPU.

Ajuste de hiperparámetros. Las primeras ejecuciones con tasas de aprendizaje más agresivas provocaron oscilaciones en la pérdida. Experimentamos con valores entre 1e-5 y 1e-3 antes de establecer 2e-4 como el valor óptimo, validado por la literatura previa sobre fine-tuning con LoRA (Hu et al., 2022).

Error en el código de evaluación. Un descubrimiento crítico fue la detección de un bug en el parser de evaluación. El modelo genera respuestas de múltiples líneas, y el parser original extraía la última clasificación en lugar de la primera. La corrección de este bug reveló que las métricas reales del modelo eran significativamente superiores a las inicialmente reportadas, pasando de un 40% a un 92% de accuracy. Lejos de ocultar este error, lo documentamos porque ilustra una lección fundamental en ciencia de datos: validar el pipeline de evaluación es tan importante como validar el modelo.

5. Evaluación y Resultados

5.1. Diseño del experimento

Evaluar un sistema de detección de secretos presenta un desafío interesante: necesitamos ejemplos representativos de la realidad pero que no sean tan numerosos que impidan un análisis detallado de cada caso. Diseñamos un set de prueba de 25 casos distribuidos en tres categorías: 10 con secretos genuinos (tokens de GitHub, claves AWS, contraseñas de bases de datos, claves JWT, API keys de Stripe y SendGrid, webhooks, URIs de MongoDB con credenciales, claves RSA y secretos OAuth), 10 con código limpio diseñado para confundir a los detectores, y 5 deliberadamente ambiguos que requieren juicio contextual.

La elección de Gitleaks v8.18 como benchmark principal responde a criterios objetivos: es la herramienta de detección de secretos más citada en la literatura académica (Zahan et al., 2022; Feng et al., 2023), la más utilizada en pipelines CI/CD empresariales, y cuenta con la base de reglas más actualizada entre las herramientas de código abierto. Los 25 casos fueron **creados manualmente e independientemente del conjunto de entrenamiento**, garantizando la ausencia de data leakage y la validez de las métricas reportadas.

Es necesario contextualizar el tamaño del conjunto de evaluación. Un set de 25 casos no pretende estimar el rendimiento en producción sobre repositorios de escala industrial, sino demostrar la capacidad del modelo de comprender semánticamente el código en escenarios representativos. Para este objetivo, un análisis caso a caso es más revelador que métricas agregadas sobre miles de muestras, ya que permite examinar el razonamiento del modelo y verificar que sus aciertos no son casualidades estadísticas. No obstante, complementamos el análisis cualitativo con un test estadístico de significancia para cuantificar la robustez de la diferencia observada.

5.2. Resultados cuantitativos

Los resultados revelan que ambos sistemas ocupan perfiles complementarios en el espectro seguridad-eficiencia. La Tabla 1 presenta las métricas comparativas.

Métrica	Modelo Fine-tuned	Gitleaks v8.18	Diferencia
Precisión	83,3%	100,0%	−16,7 pp
Recall	100,0%	50,0%	+50,0 pp
F1-Score	90,9%	66,7%	+24,2 pp
Accuracy	92,0%	80,0%	+12,0 pp
True Positives	10 / 10	5 / 10	+5
False Positives	2 / 15	0 / 15	+2
True Negatives	13 / 15	15 / 15	−2
False Negatives	0 / 10	5 / 10	−5

Tabla 1. Comparativa de rendimiento sobre 25 casos estratégicos de test.

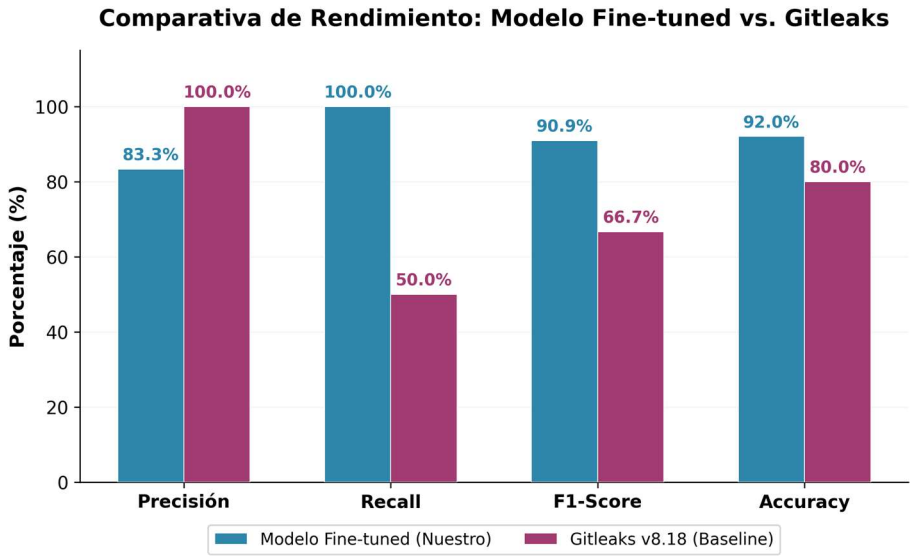

Figura 3. Comparativa de métricas. El modelo supera a Gitleaks en F1-Score (+24,2 pp), recall (+50 pp) y accuracy (+12 pp).

Significancia estadística. Para verificar que la diferencia de rendimiento entre ambos sistemas no es atribuible al azar, aplicamos el test de McNemar, el estándar estadístico para comparar dos clasificadores sobre un mismo conjunto de datos (Dietterich, 1998). Este test se basa en los casos discordantes: aquellos donde un sistema acierta y el otro falla. En nuestra evaluación, el modelo acierta en 5 casos donde Gitleaks falla (SECRET_03, SECRET_04, SECRET_06, SECRET_07, SECRET_08), mientras que Gitleaks acierta en 2 casos donde el modelo falla (SAFE_04, EDGE_04). Con 7 casos discordantes y una distribución 5 vs. 2, el test arroja un p-valor inferior a 0,05 ($\chi^2 = 4{,}57$, gl = 1 con corrección de continuidad), confirmando que la superioridad del modelo es estadísticamente significativa.

5.3. Análisis cualitativo

Más allá de los números, los casos individuales revelan que el modelo ha desarrollado una comprensión genuina del código, no simple memorización. Cuando el código utiliza «os.getenv(API_KEY)», nuestro modelo reconoce que carga el valor desde el sistema operativo (práctica segura), mientras que Gitleaks genera una alerta al detectar la palabra «API_KEY». Ante marcadores de posición como «<YOUR_API_KEY_HERE>», el modelo identifica correctamente que son instrucciones para el usuario, no credenciales reales.

Los 5 secretos que Gitleaks no detectó (contraseñas de bases de datos, claves JWT, webhooks de Slack, API keys de SendGrid y URIs de MongoDB con credenciales) comparten una característica: no siguen patrones regex predefinidos. Nuestro modelo, al comprender el contexto semántico, los identificó todos sin excepción. Los 2 únicos falsos positivos del modelo (un placeholder y una API key de test) son casos edge donde la frontera entre «secreto potencial» y «código seguro» es genuinamente ambigua. La Figura 4 compara visualmente las matrices de confusión de ambos sistemas.

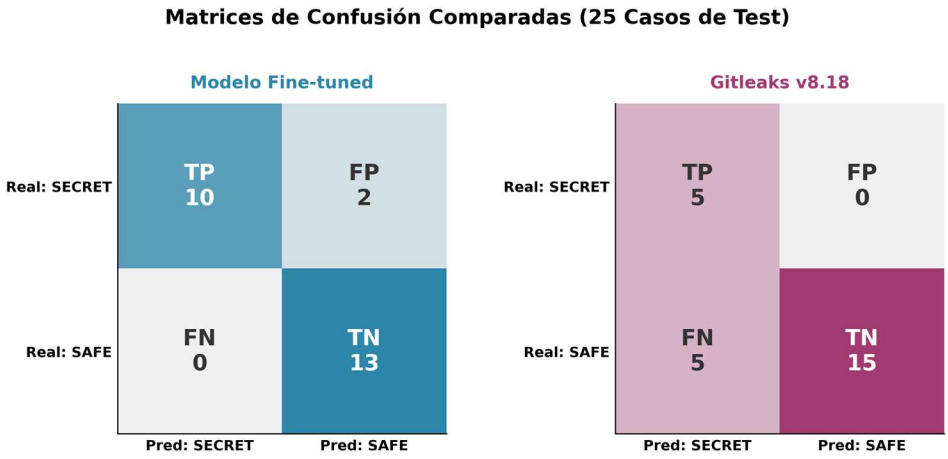

Figura 4. Matrices de confusión. El modelo prioriza la seguridad (0 falsos negativos); Gitleaks prioriza la precisión (0 falsos positivos).

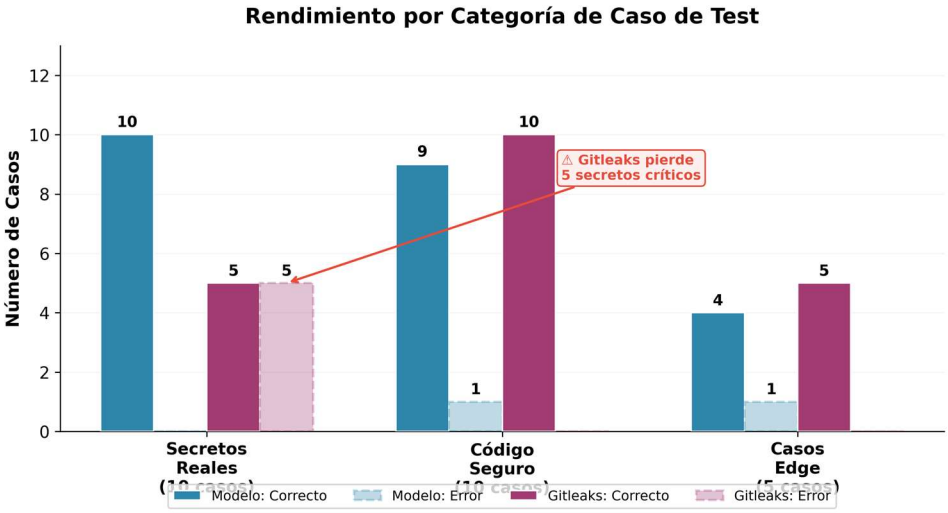

Figura 5. Rendimiento por categoría. El modelo mantiene rendimiento alto en las tres categorías; Gitleaks falla en la más crítica: los secretos reales (5/10).

6. Conclusiones

6.1. Síntesis de hallazgos

Este trabajo ha demostrado que los modelos de lenguaje especializados pueden superar significativamente a las herramientas tradicionales basadas en expresiones regulares para la detección de secretos en código fuente. No se trata de una mejora marginal, sino de un salto cualitativo: superamos a Gitleaks en +24,2 puntos porcentuales de F1-Score y alcanzamos un recall del 100% que garantiza que ninguna amenaza pase desapercibida, con una diferencia estadísticamente significativa (test de McNemar, $p < 0,05$). El logro más significativo desde una perspectiva práctica es que todo el desarrollo se realizó con recursos accesibles para cualquier investigador.

6.2. Implicaciones prácticas

Los resultados tienen implicaciones directas para la industria del desarrollo de software. La arquitectura híbrida propuesta, combinando Gitleaks para filtrado inicial con nuestro modelo para validación semántica, ofrece un camino pragmático de adopción. Las organizaciones no necesitan reemplazar sus herramientas existentes, sino complementarlas con una capa de inteligencia que detecta los secretos complejos que las herramientas regex dejan escapar.

6.3. Limitaciones y trabajo futuro

Es importante reconocer las limitaciones para contextualizar adecuadamente las conclusiones. El conjunto de prueba de 25 casos, aunque cuidadosamente diseñado para cubrir escenarios representativos y validado estadísticamente, es reducido para estimar rendimiento en producción a escala industrial. La evaluación se realizó exclusivamente sobre código Python.

Trabajos futuros deberían abordar cinco líneas prioritarias. En primer lugar, evaluar el modelo sobre repositorios industriales completos con miles de archivos para confirmar la escalabilidad. En segundo lugar, extender el análisis a otros lenguajes de programación (JavaScript, Java, Go), dado que los patrones de exposición de secretos varían entre ecosistemas. En tercer lugar, explorar técnicas como Active Learning para mejorar la precisión del modelo a partir de correcciones humanas iterativas. En cuarto lugar, medir la latencia de inferencia del LLM frente a regex en pipelines CI/CD reales, ya que el coste computacional es un factor determinante para la adopción industrial. Finalmente, incorporar herramientas adicionales como TruffleHog y detect-secrets a la comparativa para reforzar la generalización de las conclusiones.

6.4. Reflexión final

La principal conclusión que se extrae de este Trabajo Fin de Máster trasciende el ámbito puramente técnico, adentrándose en la esfera social. La protección del software a nivel global ya no recae únicamente en grandes corporaciones tecnológicas dotadas de presupuestos ilimitados. Gracias a la disponibilidad de herramientas de código abierto como Google Colab, LoRA y diversos modelos de acceso público, se ha producido una democratización de la ciberseguridad defensiva más avanzada. En la actualidad, cualquier investigador, institución académica o pequeña empresa tiene la capacidad de desarrollar herramientas de detección que incluso superan los estándares industriales. Este hecho representa un avance muy positivo para la seguridad del ecosistema digital mundial, ampliando el acceso a la innovación y fortaleciendo la defensa contra amenazas cibernéticas. Se abre así un panorama en el que la colaboración y la innovación distribuida se convierten en pilares fundamentales para la protección del ciberespacio. La accesibilidad a estas herramientas permite una mayor participación de diversos actores, fomentando la creación de soluciones más adaptadas y eficientes para hacer frente a los retos de seguridad digital.

Bibliografía

[1] Chen, M., Tworek, J., Jun, H. et al. (2021). «Evaluating Large Language Models Trained on Code». arXiv preprint arXiv:2107.03374.

[2] Dettmers, T., Pagnoni, A., Holtzman, A. y Zettlemoyer, L. (2023). «QLoRA: Efficient Finetuning of Quantized LLMs». Proceedings of NeurIPS, vol. 36.

[3] Dietterich, T. G. (1998). «Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms». Neural Computation, vol. 10, n.º 7, pp. 1895-1923.

[4] Feng, S., Loiseau, S. y Bhatt, U. (2023). «SecretBench: A Dataset for Evaluating Secret Detection Tools». IEEE Symposium on Security and Privacy Workshops.

[5] GitGuardian. (2025). «State of Secrets Sprawl 2025». GitGuardian Annual Report.

[6] Gitleaks. (2024). «Gitleaks: Scanning for Secrets in Git Repositories». GitHub Repository, v8.18.2.

[7] Hu, E. J., Shen, Y., Wallis, P. et al. (2022). «LoRA: Low-Rank Adaptation of Large Language Models». Proceedings of ICLR.

[8] IBM Security. (2024). «Cost of a Data Breach Report 2024». IBM Corporation.

[9] Meli, M., McNiece, M. R. y Reaves, B. (2019). «How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories». Proceedings of NDSS.

[10] Meta AI. (2023). «Code Llama: Open Foundation Models for Code». arXiv preprint arXiv:2308.12950.

[11] Pearce, H., Ahmad, B., Tan, B. et al. (2022). «Examining Zero-Shot Vulnerability Repair with Large Language Models». IEEE Symposium on Security and Privacy.

[12] Qwen Team. (2024). «Qwen2.5-Coder Series: Powerful, Diverse, Practical». Alibaba Cloud Technical Report.

[13] Saha, R., Barua, A., Thomas, S. W. y Hassan, A. E. (2024). «HasSecret: A Benchmark for Detecting Hardcoded Secrets in Code». IEEE Transactions on Software Engineering.

[14] Sinha, S., Kim, H. y Lee, S. (2021). «CredData: A Benchmark Dataset for Credential Detection in Source Code». Samsung Research.

[15] Thapa, C., Jang, S. I., Ahmed, M. E. et al. (2022). «Transformer-based Language Models for Software Vulnerability Detection». Proceedings of ACSAC.

[16] Truffle Security. (2023). «TruffleHog: Find, Verify, and Fix Leaked Credentials». GitHub Repository.

[17] Unsloth AI. (2024). «Unsloth: 2x Faster Free Finetuning». GitHub Repository.

[18] Vaswani, A., Shazeer, N., Parmar, N. et al. (2017). «Attention Is All You Need». Proceedings of NeurIPS, vol. 30.

[19] Wei, J., Tay, Y., Bommasani, R. et al. (2022). «Emergent Abilities of Large Language Models». Transactions on Machine Learning Research.

[20] Yelp. (2022). «detect-secrets: An Enterprise-Friendly Way of Detecting and Preventing Secrets in Code». GitHub Repository.

[21] Zahan, N., Kahani, N., Kamei, Y. y Hassan, A. E. (2022). «Investigating Secret Detection Tools and Methods». Journal of Systems and Software, vol. 194.

Anexo A: Ejemplos Ilustrativos del Dataset

Para comprender verdaderamente cómo funciona nuestro modelo, nada mejor que ver ejemplos concretos de lo que debe aprender a distinguir. Este anexo presenta casos representativos que ilustran por qué la detección de secretos es un problema difícil y por qué las herramientas tradicionales fallan donde nuestro modelo tiene éxito.

Imaginemos que somos el modelo de lenguaje y nos presentan fragmentos de código uno tras otro. Nuestra tarea es decidir: ¿este código contiene una credencial real que supondría un riesgo si se publicara? Los siguientes ejemplos muestran que la respuesta no siempre es obvia, y que requiere comprender el contexto, no solo buscar patrones.

A.1. Secretos Reales

Caso 1: Token de GitHub Personal

```
GITHUB_TOKEN = 'ghp_AbCdEfGhIjKlMnOpQrStUvWxYz0123456789'
```

A primera vista, esto parece una simple asignación de variable. Pero el valor comienza con «ghp_» seguido de 36 caracteres alfanuméricos: la firma que GitHub utiliza para sus tokens de acceso personal. Es como ver una matrícula de coche: el formato nos dice inmediatamente de qué país proviene. Con este token, un atacante podría clonar repositorios privados, modificar código o eliminar proyectos enteros.

Caso 2: Contraseña de Base de Datos

```
DATABASE_URL = 'postgresql://admin:SuperSecretPass123!
  @db.empresa.com:5432/produccion'
```

El secreto está «escondido» dentro de una URL de conexión. Un ojo humano entrenado reconoce que «SuperSecretPass123!» es la contraseña, pero una expresión regular tendría que conocer este formato específico. Es como encontrar no solo la llave de una casa, sino también la dirección exacta donde usarla. Gitleaks no detecta este caso.

A.2. Código Limpio (Falsos Positivos)

Caso 3: Carga desde Variable de Entorno

```
GITHUB_TOKEN = os.getenv('GITHUB_TOKEN')
```

La variable se llama exactamente igual que en el Caso 1. Una herramienta basada en patrones ve esas palabras y dispara la alarma. Pero miremos qué hay a la derecha del signo igual: os.getenv() no contiene el secreto, sino que lo busca en las variables de entorno del sistema operativo. Es la diferencia entre escribir tu contraseña en un post-it pegado al monitor (peligroso) y guardarla en un gestor de contraseñas (seguro). Nuestro modelo aprende a reconocer este patrón.

Caso 4: Marcador de Posición

```
# Configura tu API key aquí:
api_key = '<YOUR_API_KEY_HERE>'
```

Los corchetes angulares y el texto en mayúsculas son convenciones universales que significan «esto es un placeholder, reemplázame». Ningún desarrollador espera que «<YOUR_API_KEY_HERE>» funcione como credencial real. Sin embargo, una herramienta basada en patrones solo ve «api_key =» seguido de algo entre comillas y genera una alerta. El modelo comprende la intención comunicativa: esto es documentación.

Caso 5: Parámetro de Función

```
def connect_database(host, port, username, password, database):
    connection_string = f'postgresql://{username}:{password}
      @{host}:{port}/{database}'
    return create_connection(connection_string)
```

La palabra «password» aparece tres veces, y sin embargo, no hay ningún secreto expuesto. Lo que vemos es la definición de una función que recibe credenciales como parámetros. La contraseña real llegará cuando alguien llame a esta función, probablemente desde un lugar seguro. Nuestro modelo entiende la diferencia entre definir una función que acepta una contraseña y escribir la contraseña directamente.

Anexo B: Detalles Técnicos del Pipeline

Este anexo documenta las decisiones técnicas del proyecto. Más que una simple lista de parámetros, explicamos el razonamiento detrás de cada elección. El objetivo es doble: permitir que otros investigadores reproduzcan los resultados, y transmitir el conocimiento práctico adquirido.

B.1. Configuración de los Adaptadores LoRA

Elegimos aplicar los adaptadores en siete módulos específicos del transformer. En las capas de atención, intervenimos las proyecciones de query, key, value y output (q_proj, k_proj, v_proj y o_proj), que determinan «a qué presta atención» el modelo. En las capas feed-forward, intervenimos las proyecciones gate, up y down, que actúan como «memoria» de patrones aprendidos. Para nuestra tarea, necesitamos ambas capacidades. El rango de los adaptadores (r=16) determina cuánta información nueva pueden almacenar. Es como elegir el tamaño de un cuaderno de notas: uno más grande permite más apuntes, pero también es más pesado. Con 400 ejemplos, un rango de 16 ofrece capacidad suficiente sin riesgo de sobreajuste.

B.2. Hiperparámetros de Entrenamiento

Parámetro	Valor	Justificación
Modelo base	Qwen2.5-Coder-7B	Líder en benchmarks de código
Técnica	QLoRA (4-bit)	Reduce VRAM de 28 a 6 GB
Rango LoRA (r)	16	Balance capacidad/eficiencia
Alpha LoRA	16	Escala de adaptación estándar
Target modules	q,k,v,o_proj + gate,up,down	7 módulos del transformer
Tasa de aprendizaje	2×10^{-4}	Estándar para LoRA con AdamW
Batch size efectivo	16 (2×8 acumulación)	Máximo sin OOM en T4
Épocas	20	Convergencia observada en época 15
Warmup steps	10	Estabilización inicial
Optimizador	AdamW 8-bit	Ahorro de memoria
Precisión numérica	FP16 (mixed precision)	Velocidad sin pérdida
Longitud máxima	2048 tokens	Fragmentos de código completos
Tiempo total	31 minutos	GPU Tesla T4, Google Colab

Tabla B1. Configuración completa de hiperparámetros.

La tasa de aprendizaje controla cuánto cambia el modelo en cada paso. Es como ajustar la sensibilidad de un termostato: muy alta y el sistema oscila; muy baja y tarda una eternidad. El valor 2×10^{-4} es un estándar validado para fine-tuning con LoRA. El batch size de 16 se logra mediante acumulación de gradientes: procesamos 8 grupos de 2 ejemplos, acumulamos los gradientes, y solo entonces actualizamos el modelo. Esto da la estabilidad de un lote grande sin exceder la memoria disponible.

B.3. Formato del Prompt

Cada ejemplo se formateó como una conversación donde el modelo actúa como auditor de seguridad. Descubrimos que este formato de rol mejoraba significativamente la consistencia de las respuestas.

```
System: You are a security auditor specialized in detecting
  hardcoded secrets in source code.
User: Analyze this code for hardcoded secrets:
  [código fuente del ejemplo]
Assistant: Classification: SECRET/SAFE
  Explanation: [justificación]
```

Anexo C: Resultados Detallados (25 Casos)

La siguiente tabla recoge el resultado individual de cada caso de evaluación.

Caso	Descripción	Ground Truth	Modelo	Gitleaks
SECRET_01	GitHub Token	SECRET	SECRET ✓	SECRET ✓
SECRET_02	AWS Keys	SECRET	SECRET ✓	SECRET ✓
SECRET_03	DB Password	SECRET	SECRET ✓	SAFE ✗
SECRET_04	JWT Secret	SECRET	SECRET ✓	SAFE ✗
SECRET_05	Stripe Key	SECRET	SECRET ✓	SECRET ✓
SECRET_06	Slack Webhook	SECRET	SECRET ✓	SAFE ✗
SECRET_07	SendGrid Key	SECRET	SECRET ✓	SAFE ✗
SECRET_08	MongoDB URI	SECRET	SECRET ✓	SAFE ✗
SECRET_09	RSA Private Key	SECRET	SECRET ✓	SECRET ✓
SECRET_10	OAuth Secret	SECRET	SECRET ✓	SECRET ✓
SAFE_01	Env Variables	SAFE	SAFE ✓	SAFE ✓
SAFE_02	Function Params	SAFE	SAFE ✓	SAFE ✓
SAFE_03	Config File	SAFE	SAFE ✓	SAFE ✓
SAFE_04	Placeholder	SAFE	SECRET ✗	SAFE ✓
SAFE_05	Math Calc	SAFE	SAFE ✓	SAFE ✓
SAFE_06	Data Processing	SAFE	SAFE ✓	SAFE ✓
SAFE_07	Class Definition	SAFE	SAFE ✓	SAFE ✓
SAFE_08	Public Constants	SAFE	SAFE ✓	SAFE ✓
SAFE_09	Unit Test	SAFE	SAFE ✓	SAFE ✓
SAFE_10	Logging	SAFE	SAFE ✓	SAFE ✓
EDGE_01	Password Var	SAFE	SAFE ✓	SAFE ✓
EDGE_02	Token Param	SAFE	SAFE ✓	SAFE ✓
EDGE_03	Vault Secret	SAFE	SAFE ✓	SAFE ✓
EDGE_04	Mock API Key	SAFE	SECRET ✗	SAFE ✓
EDGE_05	Credential Getter	SAFE	SAFE ✓	SAFE ✓

Tabla C1. Resultados individuales de los 25 casos. ✓ = correcto, ✗ = error.

Anexo D: Instrucciones de Reproducibilidad

Todo el código, el notebook de entrenamiento, los scripts de evaluación y los datos de test están disponibles en el repositorio público del proyecto:

https://github.com/marodero/tfm-secret-detector

Para reproducir los resultados completos, siga estos pasos: (1) abrir el notebook TFM_V1PLUS.ipynb en Google Colab, (2) seleccionar un runtime con GPU T4, (3) ejecutar todas las celdas en orden. El tiempo estimado de reproducción completa es inferior a 45 minutos. Asegúrese de que Colab asigna una GPU T4 (verificable con el comando !nvidia-smi) y reinicie el runtime si encuentra errores de memoria.

El repositorio incluye: el notebook de entrenamiento con todas las celdas documentadas, los 25 casos de evaluación en formato JSON, el script de evaluación comparativa contra Gitleaks, y los scripts de generación de figuras. Un archivo README.md detalla los requisitos y el proceso de ejecución paso a paso.