Machine Translated by Google

Un estudio de las representaciones de código fuente para máquinas Tareas de ciberseguridad basadas en el aprendizaje

BEATRICE CASEY, Universidad de Notre Dame, Notre Dame, Estados Unidos JOANNA CS SANTOS, Universidad de Notre Dame, Notre Dame, Estados Unidos GEORGE PERRY, Universidad de Notre Dame, Notre Dame, Estados Unidos

Las técnicas de aprendizaje automático para tareas de ingeniería de software relacionadas con la ciberseguridad son cada vez más populares. La representación del código fuente es un aspecto clave de la técnica que puede influir en la capacidad del modelo para aprender las características del código fuente. Con el creciente desarrollo de estas técnicas, resulta valioso conocer el estado actual del campo para comprender mejor qué existe y qué aún no existe. Este artículo presenta un estudio de estos enfoques existentes basados en aprendizaje automático y demuestra qué tipo de representaciones se utilizaron para diferentes tareas de ciberseguridad y lenguajes de programación. Además, estudiamos qué tipos de modelos se utilizan con diferentes representaciones. Hemos descubierto que las representaciones basadas en grafos son la categoría de representación más popular, y los tokenizadores y los árboles de sintaxis abstracta (AST) son las dos representaciones más populares en general (por ejemplo, AST y tokenizadores son las representaciones con el mayor número de artículos, mientras que las representaciones basadas en grafos son la categoría con el mayor número de artículos). También descubrimos que la tarea de ciberseguridad más popular es la detección de vulnerabilidades, y el lenguaje que cubre la mayoría de las técnicas es C. Finalmente, descubrimos que los modelos basados en secuencias son la categoría de modelos más popular, y las máquinas de vectores de soporte son el modelo más popular en general.

Conceptos del CCS: • General y de referencia → Encuestas y descripciones generales;

Palabras y frases clave adicionales: Representación del código fuente, aprendizaje automático para la seguridad del software, revisión sistemática de la literatura.

Formato de referencia ACM:

Beatrice Casey, Joanna CS Santos y George Perry. 2025. Un estudio de representaciones de código fuente para tareas de ciberseguridad basadas en aprendizaje automático. ACM Comput. Surv. 57, 8, artículo 217 (abril de 2025), 41 páginas. https://doi.org/10.1145/3721977

1 Introducción

Las vulnerabilidades de software son defectos que afectan las propiedades de seguridad previstas de un sistema de software [1], lo que permite a los atacantes realizar acciones maliciosas. A medida que nuestras vidas se vinculan más con la tecnología , los proveedores de software se ven cada vez más presionados a diseñar sistemas de software seguros; es decir, a tomar medidas proactivas para prevenir y reparar vulnerabilidades antes de implementar los sistemas en producción. Existen varias prácticas para abordar las preocupaciones de seguridad antes del lanzamiento del software en cada fase del ciclo de vida del desarrollo de software. En la fase de requisitos, los casos de uso indebido/abuso [2, 3] y el modelado de amenazas [4] son útiles para comprender mejor los requisitos de seguric

Información de contacto de los autores: Beatrice Casey, Universidad de Notre Dame, Notre Dame, Indiana, Estados Unidos; correo electrónico: bcasey6@nd.edu; Joanna CS Santos, Universidad de Notre Dame, Notre Dame, Indiana, Estados Unidos; correo electrónico: joannacss@nd.edu; George Perry, Universidad de Notre Dame, Notre Dame, Indiana, Estados Unidos; correo electrónico: gperry@nd.edu.

Esta obra está bajo una Licencia Creative Commons Atribución 4.0 Internacional.

© 2025 Copyright perteneciente al propietario/autor(es).
ACM 0360-0300/2025/04-ART217 https://doi.org/10.1145/3721977

Machine Translated by Google

Identificar las amenazas potenciales al sistema y las posibles maneras de mitigarlas. Durante la fase de diseño, el análisis de riesgos arquitectónicos [5] puede utilizarse para evaluar la probabilidad de explotación de un activo, y las tácticas y patrones de seguridad [6] pueden aplicarse en el diseño del software como una solución probada que funciona en un contexto determinado. En la fase de implementación, se pueden realizar revisiones de código seguro para examinar sistemáticamente el código fuente con el objetivo de identificar y corregir vulnerabilidades [7]. En la fase de prueba, las pruebas de penetración buscan probar el software de diversas maneras para intentar descifrarlo y descubrir vulnerabilidades [8] , y se pueden utilizar herramientas de análisis estático/dinámico para identificar posibles vulnerabilidades en el código fuente [9, 10].

Aunque estas prácticas pueden ayudar a mejorar la seguridad de un sistema de software, su realización puede ser propensa a errores y consumir mucho tiempo para los ingenieros. Por ejemplo, encontrar vulnerabilidades en el código puede ser difícil para los ingenieros, especialmente cuando no saben qué buscar [7]. Con los avances del aprendizaje automático (ML), trabajos previos han aplicado técnicas de ML a varias de estas tareas de ciberseguridad, como la detección de vulnerabilidades [11–16], la detección de malware [17–26] y la detección de comportamiento malicioso [27]. Estas técnicas son valiosas porque pueden ayudar a mejorar la seguridad del código que se publica y agilizar tareas que de otro modo serían propensas a errores y consumirían mucho tiempo. Por ejemplo, un modelo que detecte vulnerabilidades antes de la implementación ahorraría tiempo y dinero, y aumentaría la seguridad del sistema en su conjunto, especialmente porque los desarrolladores a menudo desconocen la existencia de una vulnerabilidad en el código hasta que los analistas de seguridad la explotan o la encuentran [28

Los modelos de aprendizaje automático no pueden comprender el código fuente sin procesar, es decir, el código fuente sin procesar . Dado que los modelos de aprendizaje automático se basan en datos numéricos para ajustar las ponderaciones necesarias para el aprendizaje, el código fuente debe representarse de forma que capture eficazmente su información estructural y semántica. Esta transformación permite que el modelo comprenda y aprenda de los patrones subyacentes del código. Por lo tanto, la representación del código fuente es crucial durante el desarrollo de técnicas basadas en aprendizaje automático, ya que diferentes representaciones ofrecerán información distinta de la que el modelo aprende para realizar su tarea [29].

Hay muchas maneras en las que se puede representar el código fuente, por ejemplo, como un árbol de sintaxis abstracta (AST) [12, 16, 30–34], un gráfico de flujo de control (CFG) [12, 13, 35–37] o tokenizado [11, 15, 38–40], entre otros. Aunque trabajos anteriores [41–48] han introducido nuevas representaciones de código fuente para diferentes tareas de ciberseguridad, no existe un conocimiento actual de qué representaciones existen y se usan comúnmente, así como de las tareas de ciberseguridad para las que se usa el modelo. Además, muchas de estas técnicas basadas en ML solo se prueban o se crean para un lenguaje de programación en particular, y no existe un conocimiento actual de los lenguajes que cubren estas técnicas. Comprender las representaciones de código fuente disponibles y lo que ofrecen permitirá a los investigadores identificar qué representación pueden querer usar en función de la tarea que pretendan completar. Además, comprender la relación entre las tareas y las representaciones de ciberseguridad permitirá a los investigadores elegir una representación que se haya utilizado previamente para una tarea particular o probar si una representación diferente sería más adecuada.

En este artículo, realizamos una Revisión Sistemática de la Literatura (SLR) siguiendo las directrices de Kitchenham y Charters [49] para comprender el estado actual de la representación de código fuente para técnicas basadas en aprendizaje automático (ML) en tareas de ciberseguridad. Investigamos la popularidad de ciertas representaciones y tareas de ciberseguridad, los lenguajes de programación que abarcan las técnicas existentes y los tipos comunes de modelos de ML utilizados con diferentes representaciones. También investigamos las relaciones entre las representaciones y las tareas de ciberseguridad. El objetivo del estudio es permitir a los investigadores comprender las deficiencias en este ámbito, en particular si existen ciertas tareas de ciberseguridad o lenguajes que las técnicas existentes descuidan. Además, estudiamos y contrastamos las representaciones existentes.

Las contribuciones de este artículo son las siguientes: (1) un examen del estado del arte de las tareas de ciberseguridad basadas en ML; (2) una investigación de las representaciones del código fuente y sus relaciones.

Machine Translated by Google

a las tareas y modelos de ciberseguridad; (3) la identificación de los lenguajes de programación cubiertos /no cubiertos por las técnicas existentes basadas en aprendizaje automático; y (4) una comparación de las diferentes representaciones del código fuente. Los recursos del artículo, incluyendo conjuntos de datos, código y recursos adicionales, están disponibles en GitHub: https://github.com/s2e-lab/code-representation-slr.

El resto de este artículo se organiza de la siguiente manera. La Sección 2 proporciona las definiciones de términos relevantes para comprender nuestro trabajo. La Sección 3 describe trabajos relacionados. La Sección 4 explica la metodología de esta SLR. Las Secciones 5 a 9 comparten los resultados de este trabajo. La Sección 10 explica las amenazas a la validez, y la Sección 11 ofrece una discusión y conclusión de nuestros hallazgos.

2 Antecedentes En

esta sección se analiza la terminología fundamental para que el artículo pueda ser comprendido por un público más amplio.

2.1 ML para ingeniería de software segura Además de

desarrollar código nuevo de buena calidad, los ingenieros de software son responsables de descubrir y resolver errores, defectos, vulnerabilidades y cualquier otro problema que pueda surgir en el código fuente.
Estas tareas pueden ser difíciles, propensas a errores, lentas y tediosas [7]. Según la definición de Kemmerer [50], una tarea de ciberseguridad es aquella cuyo objetivo es frustrar a posibles intrusos. Por lo tanto, las tareas de ciberseguridad que realizan los ingenieros de software hoy en día implican la resolución e identificación de código que podría permitir a un atacante aprovecharse de un sistema.

Con los recientes avances del aprendizaje automático (ML), varios trabajos previos identificaron maneras en que el ML podría ayudar a los ingenieros de software en estas tareas de ciberseguridad [51]. En particular, dado el impacto negativo que las vulnerabilidades y otros problemas relacionados con la seguridad tienen en las empresas, los investigadores comenzaron a estudiar cómo el ML puede ayudar a mitigar estos problemas y, en general, mejorar la calidad del código fuente que se publica .

2.2 Representaciones de código fuente e incrustaciones de código. Una

representación de código fuente captura la sintaxis y la semántica del código fuente, de modo que el modelo pueda aprender las características clave. Hay muchas maneras de representar un fragmento de código fuente.
Por ejemplo, dada la naturaleza estructurada del código fuente, trabajos previos capturaron esta estructura representándola como un grafo [52]. Otros trabajos [38, 53, 54] emplearon técnicas de Procesamiento del Lenguaje Natural (PLN) en código fuente para aprovechar la tecnología y el conocimiento ya existentes. Estas representaciones ofrecen información diversa sobre el código fuente y, por lo tanto, influyen en lo que el modelo puede aprender sobre él. Por ejemplo, las técnicas de PLN no ofrecen información estructural, sino información semántica. Por lo tanto, el modelo aprenderá las relaciones semánticas, pero no las estructurales, en el código.

Los modelos de ML aprenden de incrustaciones vectoriales, que es una forma de baja dimensión de representar datos de alta dimensión. En el caso del aprendizaje de código fuente, las incrustaciones se crean a partir de la representación del código fuente. La representación del código fuente es el primer nivel de abstracción para el código fuente original. Esto es lo que se considera como la fase de extracción de características. En esta fase, los datos originales de alta dimensión se transforman en datos de menor dimensión, que representan las características clave de los datos. La extracción de características tiene como objetivo preservar la mayor cantidad posible de información sobre los datos originales [55]. Las incrustaciones vectoriales son el segundo nivel de abstracción y son lo que nos permite realizar técnicas de ML en las representaciones al transformar la representación a la forma numérica que las máquinas pueden entender [56]. En esta revisión de la literatura, nos centramos solo en el primer nivel de abstracción (es decir, la representación del código fuente).

Existen numerosas técnicas para crear estas incrustaciones. En el ámbito del procesamiento del lenguaje natural (PLN), las incrustaciones suelen crearse mediante el modelo word2vec [57], que toma tokens y

Machine Translated by Google

Los convierte en vectores numéricos. Inspirados por esta metodología, los investigadores han encontrado una manera de tomar un grafo y crear estas incrustaciones vectoriales (graph2vec [58]). Esta técnica es la que se utiliza habitualmente para generar incrustaciones a partir de las estructuras de árbol o grafo generadas por estas representaciones de código fuente. Graph2vec implementa ideas de doc2vec y word2vec, y trata un grafo completo como un documento y los subgrafos como palabras [58]. Además, otros trabajos han analizado cómo crear incrustaciones directamente desde el código fuente y han creado técnicas como code2vec [59] y GraphCode2Vec [60].

3 trabajos relacionados

Hasta donde sabemos, este es el primer SLR de su tipo que se centra en las representaciones de código fuente utilizadas en técnicas basadas en ML para tareas de ciberseguridad. Existen varios artículos [61–69] que realizan un estudio de mapeo sistemático o una revisión de la literatura sobre ML para ingeniería de software, y algunos se centran en la detección, el análisis o la evaluación de vulnerabilidades. Todos estos artículos se centran principalmente en los modelos de ML utilizados, pero pocos mencionan o ofrecen descripciones detalladas de las representaciones empleadas en estos esfuerzos. Además, algunos artículos se centran únicamente en técnicas de aprendizaje profundo [64–66]. A diferencia de estos trabajos previos, nuestro artículo de revisión se centra principalmente en el tema de las representaciones utilizadas para ML en tareas relacionadas con la seguridad.

Ghaffarian y Shahriari [70] analizaron las técnicas utilizadas para el análisis de vulnerabilidades y describieron cuatro categorías principales en las que se encuadran los enfoques de estas técnicas: métricas de software, detección de anomalías, reconocimiento de patrones de código vulnerable y miscelánea. De forma similar, Nazim et al. [63] analizaron modelos de aprendizaje profundo para la detección de código vulnerable. Su estudio examinó los diferentes tipos de conjuntos de datos utilizados para entrenar modelos de aprendizaje profundo (sintéticos, semisintéticos, datos reales, etc.), las métricas de evaluación utilizadas para evaluar el rendimiento, así como las diferentes representaciones de código fuente utilizadas. A diferencia de estos trabajos previos, miramos más allá del análisis y la detección de vulnerabilidades y, en su lugar, analizamos todas las tareas de ingeniería de software relacionadas con la seguridad. También tenemos un alcance más amplio porque analizamos todos los tipos de modelos de aprendizaje automático, no solo los modelos de aprendizaje profundo.

Wu [71] realizó una revisión bibliográfica sobre técnicas de PLN para la detección de vulnerabilidades. Si bien este artículo ofrece una breve descripción general de los diferentes tipos de representaciones, se centra en las técnicas de PLN, en particular en los modelos de PLN enfocados en la inteligencia de código, como CodeBERT y CodeXGlue, para la detección de vulnerabilidades. Chen y Monperrus [72] realizan una revisión bibliográfica similar , investigando técnicas de incrustación de palabras en programas. En este estudio, los autores exploran diferentes granularidades de incrustaciones de diferentes artículos y muestran visualizaciones de las mismas . Nos centramos en una amplia gama de tareas relacionadas con la seguridad, así como en numerosas representaciones y en cómo pueden afectar la capacidad del modelo para aprender vulnerabilidades.

Kotti et al. [73] realizaron un estudio terciario sobre el aprendizaje automático (ML) para la ingeniería de software. Este artículo evaluó 83 revisiones o encuestas sobre el campo del ML para la ingeniería de software. Si bien nuestro artículo se centra en las tareas de ingeniería de software relacionadas con la seguridad, Kotti et al. [73] investigaron todas las tareas de ingeniería de software basadas en ML . Además, si bien este artículo ofrece un análisis más amplio de las tareas de ingeniería de software que abarca el ML, nuestro artículo se centra principalmente en las representaciones del código fuente utilizadas para realizar una tarea de seguridad.

Dos artículos crean una taxonomía para las tareas de ingeniería de software y el aprendizaje automático (ML). Uno se centra ampliamente en los desafíos de la ingeniería de software para los sistemas de ML [74], y el otro en la detección de vulnerabilidades de software y los enfoques de ML [75]. Ninguno de estos artículos profundiza en los esfuerzos de representación del código fuente ni en la información que ofrecen. Sin embargo, Hanif et al. [75] sí mencionan la importancia de la representación en el flujo de trabajo del ML. A diferencia de estos estudios, nuestro artículo se centra en las representaciones utilizadas, en lugar de solo en los modelos, y en una amplia gama de tareas de seguridad.

Usman et al. [76] realizaron una encuesta sobre las iniciativas de aprendizaje de representación en ciberseguridad. Sin embargo, este artículo no se centra en la representación del código fuente, sino en diferentes algoritmos de aprendizaje automático.

Machine Translated by Google

Se utilizan para abordar problemas de ciberseguridad, así como para analizar los conjuntos de datos y cómo la industria utiliza estas diferentes iniciativas para mejorar su ciberseguridad. De igual manera, Macas et al. [77] elaboraron una encuesta sobre técnicas de aprendizaje profundo para la ciberseguridad. Este artículo se centra en estas técnicas para analizar el tráfico de internet. Proporciona información y futuras direcciones en esta área del aprendizaje profundo para analizar el tráfico de internet con fines de ciberseguridad.

4 Metodología Seguimos

las directrices descritas por Kitchenham y Charters [49] para llevar a cabo nuestra SLR, que implica tres actividades principales: planificar, llevar a cabo e informar la revisión. Durante la fase de planificación, definimos las preguntas de investigación de este estudio y la consulta de búsqueda utilizada para encontrar los artículos. Durante la fase de realización, buscamos tres fuentes de biblioteca y descargamos todos los artículos que encontramos en archivos CSV. Luego aplicamos nuestros criterios de inclusión y exclusión en tres fases para eliminar artículos hasta que llegamos al grupo final de artículos que se incluyen en este estudio. Dos revisores leyeron de forma independiente cada artículo y realizaron un análisis, extrayendo la información relevante para las preguntas de investigación que desarrollamos en la fase de planificación. Revisamos y resolvimos las discrepancias para obtener los análisis finales. Calculamos la kappa de Cohen para evaluar la fiabilidad de nuestra evaluación. Nuestra puntuación de 0,97 indica que tuvimos un acuerdo casi perfecto en nuestro análisis.

Finalmente, durante la fase de informe, analizamos nuestros datos y los organizamos para poder responder a las preguntas de investigación que planteamos.

4.1 Preguntas de investigación A

través de este SLR, pretendemos responder cinco preguntas de investigación.

RQ1: ¿Cuáles son las representaciones de código fuente más utilizadas?

En esta primera pregunta, investigamos las representaciones de código fuente utilizadas para resolver problemas de seguridad. Nuestro objetivo es comprender cuáles son las más populares y comparar sus ventajas y desventajas.

RQ2: ¿Ciertas tareas de ciberseguridad utilizan únicamente o mayoritariamente un tipo de representación de código fuente?

En esta pregunta, investigamos qué representaciones de código fuente se utilizan para cada tarea de ciberseguridad y si ciertas representaciones son las preferidas para tareas específicas. Además , queremos investigar por qué se prefiere una representación específica para una tarea.

RQ3: ¿Qué tareas de ciberseguridad están cubiertas por las técnicas que se han creado?

Investigamos cómo estas tareas se integran en el ciclo de vida del desarrollo de software para identificar cómo se utilizarían estas técnicas durante el desarrollo. Además, describimos y localizamos cada tarea relacionada con la ciberseguridad para ofrecer una visión más clara de cada tarea y su importancia en el ámbito de la seguridad del software.

RQ4: ¿Qué lenguajes de programación son el objetivo predominante de las técnicas basadas en ML para tareas de ciberseguridad?

Dada la gran cantidad de lenguajes de programación utilizados en la práctica, investigamos en qué lenguajes está escrito el código fuente analizado mediante estas técnicas. Esta pregunta ayuda a identificar posibles lagunas en la cobertura de los lenguajes de programación que utilizan estas técnicas.

Machine Translated by Google

RQ5: ¿Qué modelos se utilizan comúnmente con diferentes representaciones?

En esta última pregunta, estudiamos qué representaciones de código fuente se utilizan para diferentes tipos de modelos. Al examinar la frecuencia de coocurrencia entre modelos y representaciones, buscamos comprender las tendencias y preferencias comunes en el diseño de técnicas basadas en aprendizaje automático. Este análisis ayuda a identificar posibles deficiencias y oportunidades de mejora en la combinación de modelos con representaciones específicas.

4.2 Método de búsqueda

Utilizamos la siguiente cadena de búsqueda para encontrar todos los estudios primarios relacionados con la(s) representación(es) del código fuente para tareas de ciberseguridad basadas en aprendizaje automático: ("aprendizaje automático" O "aprendizaje profundo" O "inteligencia artificial") Y ("seguridad" O "vulnerabilidad") Y ("código"). Si bien se trata de una cadena de búsqueda muy general, que arrojó un total de 67 512 artículos, decidimos que, en lugar de tener una cadena específica que pudiera omitir una categoría de tareas o representaciones de seguridad de software, crearíamos una cadena general y eliminaríamos manualmente cualquier artículo que no cumpliera con nuestros criterios de inclusión o exclusión.

Buscamos en tres bases de datos para encontrar artículos relevantes: la Biblioteca Digital ACM,[1] IEEE Xplore,[2] y Springer Link.[3] También buscamos nueve conferencias de ML y NLP que se consideran A* conferencias por el ranker de conferencias CORE. Estas nueve conferencias son Conferencia Nacional de la Asociación Americana para la Inteligencia Artificial (AAAI), Conferencia sobre Teoría del Aprendizaje (COLT), Conferencia Internacional sobre Representaciones del Aprendizaje (ICLR), Conferencia Internacional sobre Aprendizaje Automático (ICML), Conferencia Conjunta Internacional sobre Inteligencia Artificial (IJCAI), Avances en Sistemas de Procesamiento de Información Neural (NeurIPS, anteriormente conocido como NIPS), Asociación para la Lingüística Computacional (ACL),[4] Métodos Empíricos en el Procesamiento del Lenguaje Natural (EMNLP) y Conferencia Internacional sobre los Principios de la Representación y el Razonamiento del Conocimiento (KR). No buscamos artículos publicados en la Conferencia Conjunta Internacional sobre Agentes Autónomos y Sistemas Multiagente (anteriormente Conferencia Internacional sobre Sistemas Multiagente, ICMAS, modificada en 2000), la Conferencia Internacional ACM sobre Descubrimiento de Conocimiento y Minería de Datos, o la Conferencia Internacional IEEE sobre Minería de Datos porque las actas de estas conferencias estaban incluidas en las bibliotecas ACM e IEEE.

4.3 Criterios de inclusión y exclusión

La Tabla 1 enumera los criterios de inclusión/exclusión aplicados a los artículos en múltiples etapas para eliminar los artículos irrelevantes para este estudio. Limitamos nuestra búsqueda a artículos publicados entre enero de 2012 y mayo de 2023. Nuestros criterios de inclusión se centraron en técnicas basadas en ML para ciberseguridad que implicaban la representación del código fuente. Eliminamos estudios duplicados, trabajos que no estaban en inglés y cualquier artículo que no fuera un artículo completo, como libros, artículos cortos (es decir, artículos con menos de cinco páginas completas de texto, sin incluir referencias) y tutoriales. También descartamos cualquier artículo que no representara el código fuente en sí (por ejemplo, artículos que trataran con archivos binarios, extracción del manifiesto de Android), ya que solo nos interesa comprender la representación del código fuente sin procesar.

1https://dl.acm.org
2https://ieeexplore.ieee.org
3https://link.springer.com
4Cuando obtuvimos los datos para ACL, eso incluye Transacciones de la Asociación de Lingüística Computacional y la Antología de ACL.

Machine Translated by Google

Tabla 1. Criterios de inclusión y exclusión

Criterios de inclusión	Criterios de exclusión
I1 Escrito entre 2012 y mayo de 2023 I2 Un artículo completo I3 Enfocado en ML para tareas de ciberseguridad I4 Contiene información sobre la representación del código fuente.	E1 Estudios duplicados E2 Libros, entradas de trabajos de referencia, obras de referencia E3 Documentos de posición, artículos breves, documentos de demostración de herramientas, notas clave, revisiones, tutoriales y mesas redondas E4 Estudios que no están en inglés E5 Estudios comparativos/de encuesta.

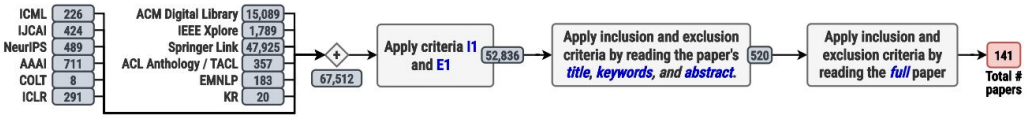

Fig. 1. Resumen de las tres etapas de nuestro proceso de búsqueda

4.4 Selección de Artículos.

La Figura 1 muestra el número de artículos que superaron cada etapa del proceso de selección. Comenzamos con un total de 67,512 estudios primarios. Primero excluimos los estudios duplicados y los estudios que estaban fuera del rango de años de 2012 a mayo de 2023 (criterio I1), así como los artículos no completos (criterio E1). Esto resultó en 52,836 artículos. Posteriormente, inspeccionamos el título, las palabras clave y el resumen de cada artículo para incluirlos o excluirlos según si cumplían con nuestros criterios. Después de esta búsqueda, nos quedaron 520 artículos. Luego aplicamos los mismos criterios a estos 520 artículos, esta vez leyendo el artículo completo. Esto nos dejó con los 141 artículos que se incluyen en esta encuesta.

4.5 Extracción de datos

A medida que revisamos los artículos, extrajimos la información clave que buscábamos para responder a nuestras preguntas de investigación: la representación utilizada, la tarea de ciberseguridad que estaba completando, los lenguajes de programación para los que se diseñó o probó la técnica y el tipo de modelo utilizado en el trabajo. Al determinar la representación utilizada, empleamos un enfoque de un artículo a múltiples representaciones, basado en las representaciones que el artículo afirma utilizar. En otras palabras, consideramos cada representación que un artículo utiliza explícitamente para entrenar o perfeccionar sus modelos como una representación independiente y válida. Sin embargo, si un artículo emplea una o más representaciones únicamente como pasos internos para derivar una representación final, consideramos únicamente la representación final como la representación utilizada. Por ejemplo, si un artículo refina un CFG inicial mediante múltiples transformaciones para generar una expresión regular que represente el código, capturamos únicamente la expresión regular como la representación utilizada por el artículo.

Además de capturar los metadatos anteriores para responder directamente a nuestras cinco preguntas de investigación, realizamos un análisis exhaustivo del rendimiento de las representaciones del código fuente. En concreto, revisamos los resultados reportados en cada artículo y extrajimos las métricas proporcionadas. Esto nos permitió agregar los datos y comparar el rendimiento de diferentes representaciones de código fuente en diversas tareas de ciberseguridad.

5 Resultados de RQ1: ¿Cuáles son las representaciones de código fuente más utilizadas?

La Tabla 2 resume las representaciones de código fuente utilizadas/descritas en los artículos analizados. Al igual que en un trabajo previo [64], organizamos estas representaciones en cuatro categorías: basadas en grafos, basadas en árboles, léxicas y misceláneas. Encontramos que las tres representaciones de código fuente más utilizadas

Machine Translated by Google

Tabla 2. Representaciones del código fuente, sus categorías y frecuencia de uso en los artículos encuestados

	Representación	Papeles	#	Representación	Papeles	#
Basado en gráficos	Gráfico de flujo de control (CFG)	[12, 13, 35–37, 78–86]	14	Gráfico de comportamiento de componentes (CBG)	[27]	1
	Gráfico de dependencia del programa (PDG)	[12, 87–93]	8	Gráfico de dependencia de componentes (CDG)	[27]	1
	Gráfico de flujo de datos (DFG)	[13, 37, 78, 80, 84, 94, 95]	7	ICFG contextual (CICFG)	[26]	1
	Gráfico de llamadas	[20, 22, 25, 35, 79, 96]	6	Dependencia de permisos contextuales Gráfico (CPDG)	[26]	1
	Gráfico de propiedades de código (CPG)	[12, 97–100]	5	Gráfico de dependencia contextual de origen y destino (CSSDG)	[26]	1
	Diagrama de flujo de control interprocedimental (ICFG)	[18, 19, 26]	3	Gráfico de flujo de datos cruciales (CDFG)	[101]	1
	Gráfico de dependencia de API contextual (CADG)	[19, 26]	2	Gráfico del programa	[102]	1
	Gráfico de contrato	[46, 103]	2	Cadena de propagación	[104]	1
	Rebanadas de programa	[14, 105]	2	Gráfico de propiedades	[45]	1
	Gráfico de propiedades de código simplificado (SCPG)	[106, 107]	2	Gráfico semántico	[108]	1
	Gráfico de dependencia del sistema (GDS)	[109, 110]	2	Gráfico de propiedades de corte (SPG)	[111]	1
	Gráfico de tokens	[112, 113]	2	Gráfico de flujo de valor (GV)	[44]	1
	Gráfico de agregación de código (CAG)	[114]	1			
Léxico	Tokenizador	[11, 15, 17, 38–40, 43, 53, 54, 82, 86–90, 104, 115–145]	47	Métricas de código	[96, 116, 132, 133, 146–157]	16
	Código intermedio y candidato a vulnerabilidad basado en semántica (iSeVC)	[48, 158, 159]	3	Gadgets de código	[14, 41, 110, 160–163]	7
	Código fuente y vulnerabilidad basada en sintaxis Candidato (sSyVC)	[48, 158]	2	Imagen	[21, 87, 92]	3
	Fragmento de contrato	[47]	1	Secuencias de códigos de operación	[164]	1
Basado	Árbol de sintaxis abstracta (AST)	[12, 16, 30–34, 37, 78, 80, 85, 88–90, 106, 142, 144, 157, 163, 165–177]	32	Información de la aplicación	[23]	1
	AST+	[51, 178]	2	Llamadas API	[24]	1
	Árbol de análisis sintáctico	[179]	1	Expresión regular	[180]	1

son un tokenizador, un AST y métricas de código. También encontramos dos artículos [51, 178] que utilizaron un
Versión AST modificada para representar el código fuente (las denotamos como AST+ en la Tabla 2). Aunque
Más raramente, algunos artículos [21, 87, 92] representaron el código como un análisis de imagen y realizaron un análisis de imagen.
Análisis del código fuente para identificar patrones asociados con vulnerabilidades o malware.
Las siguientes subsecciones dan una explicación detallada de qué es cada representación y la información
Se lleva a cabo hasta las incrustaciones.

5.1 Representaciones de código fuente basadas en árboles

Las representaciones basadas en árboles son aquellas que demuestran la naturaleza jerárquica del código fuente [64].

5.1.1 Árbol de sintaxis abstracta. Un AST es una representación de árbol del código fuente que proporciona información sobre los elementos del código (por ejemplo, variables) y su relación estructural [63, 181]. Fue el
La representación más popular (utilizada en 32 artículos). Aunque un artículo [168] utilizó code2vec [182] como forma de generar incrustaciones de código fuente, la base de su modelo es un AST; el código fuente es representado como un AST antes de que se generen los vectores.

Utilizando la información de un AST, los modelos pueden capturar patrones generales de código estructural, ya que
Los AST abstraen los detalles de sintaxis de bajo nivel del lenguaje de programación subyacente del código [12]. Esto reduce el esfuerzo de aprendizaje y permite que los AST se utilicen para múltiples tareas [59].
Las incrustaciones para AST se pueden generar de diferentes maneras. Normalmente, el nodo y la ruta son lo que...
Formar la incrustación de modo que la relación entre dos nodos pueda ser capturada de manera efectiva por el incrustación [59].

5.1.2 Árbol de análisis. Un árbol de análisis representa la jerarquía de tokens, es decir, la terminal del programa. y símbolos no terminales. Esta estructura de datos la genera el analizador sintáctico del lenguaje [183]. Por lo tanto,

Machine Translated by Google

Los nodos representan la derivación de la gramática que genera las cadenas de entrada. Esta representación
Ceccato et al. [179] lo han utilizado para representar consultas SQL para entrenar un modelo que detecta SQL.
Vulnerabilidades de inyección. Aunque tanto los árboles de análisis como los AST representan el código fuente en un árbol
estructura, su diferencia clave es que los AST son mucho más simples que los árboles de análisis, ya que abstraen
nodos relacionados con la gramática, mientras que los árboles de análisis conservan estos tokens y sus significados con respecto a
su gramática. En el Apéndice A.1, proporcionamos un ejemplo de un árbol de análisis y un AST para el mismo
Código fuente para demostrar estas diferencias.

5.1.3 AST+. Un artículo [178] utilizó una representación que es una versión mejorada de un AST (que
que denotamos en nuestro artículo como AST+). Ese trabajo utiliza una convención [184] que describe los nodos AST
en tres tipos: marcadores de posición, API y nodos de sintaxis. Los AST se serializan y recorren mediante
Recorrido en profundidad, y cada nodo y elemento se asigna a un vector. Xia et al. [51] no especifican
las modificaciones realizadas al AST; sin embargo, el documento afirma que se agregan bordes adicionales
al AST para capturar más información semántica y de flujo. Esta representación se utilizó para
detección de vulnerabilidades [51, 178].

5.2 Representaciones basadas en gráficos

Las representaciones basadas en gráficos son aquellas que transforman el código fuente en una forma de gráfico, con nodos
y bordes que representan ciertas características y relaciones, respectivamente, entre cada código
Las representaciones basadas en gráficos se pueden integrar utilizando Graph2vec [58], ya que es un elemento optimizado .
método para transformar los gráficos en vectores numéricos de baja dimensión que los modelos
aprenderá de. Como se muestra en la Tabla 2, encontramos 24 representaciones gráficas diferentes, cuyas
Se proporcionan descripciones en las siguientes secciones.

5.2.1 Gráfico de flujo de control. Un CFG [185] fue la representación gráfica más popular utilizada en 14
artículos [12, 13, 35–37, 78–86]. Un CFG es un grafo dirigido = (,) con nodos y aristas
, donde × . El conjunto de nodos representa los bloques básicos del procedimiento de un programa.
(es decir, una función/método), mientras que el conjunto de bordes representa el flujo de control entre los elementos básicos.
bloques. Un bloque básico es un grupo de instrucciones que se ejecutan en orden, una tras otra. Un
La ventaja de CFG = −→ denota que la ejecución del programa puede fluir desde a .

5.2.2 Diagrama de flujo de control interprocedimental. Un diagrama de flujo de control interprocedimental (ICFG)
es una variación del CFG que describe no sólo los flujos intraprocedimentales entre los bloques básicos sino
También interprocedimentales [19]. Un ICFG conecta CFG individuales en los sitios de llamada para representar
flujos de control entre procedimientos. Por lo tanto, el ICFG permite que el modelo comprenda el flujo de control de
todo el programa, mientras que el CFG permite que el modelo comprenda el flujo de control de un programa específico.
procedimiento [186].

5.2.3 Gráfico de flujo de datos. Siete artículos utilizaron un gráfico de flujo de datos (DFG), que es un gráfico
= (,), donde los nodos son declaraciones en el código fuente y el conjunto de aristas
Representa las dependencias de datos entre los nodos. En otras palabras, una arista = indica −→
que el nodo utiliza datos definidos por .

5.2.4 Gráfico de dependencia del programa. Los gráficos de dependencia del programa (PDG) fueron el segundo...
La representación basada en grafos más popular, utilizada en ocho artículos. Un PDG [187] es una representación dirigida.
gráfico = (,) que muestra las dependencias de datos y control para cada declaración en un programa
procedimiento. El conjunto de nodos en un PDG se divide en dos tipos: nodos de declaración y
nodos de expresión de predicado Un nodo de declaración representa declaraciones simples
En un programa, que son acciones que debe realizar un programa (p. ej., x = 2;). Un nodo de predicado
denota declaraciones que evalúan como verdaderas o falsas (p. ej., x != 2). El conjunto de aristas en
Un PDG tiene dos particiones: bordes de dependencia de control y bordes de dependencia de datos. Un control

Machine Translated by Google

El borde de dependencia = −→ indica que se evalúa como solo se ejecuta si la expresión de predicado en verdadero. Un borde de dependencia de datos = ha sido −→ denota que utiliza datos que tienen definido por .

5.2.5 Gráfico de llamadas. Un gráfico de llamadas es un gráfico dirigido = (,) en el que los nodos son los funciones/métodos en un programa, mientras que los bordes representan relaciones entre el llamador y el llamado entre Procedimientos del programa [188]. Una arista = −→ invoca . Estos gráficos denota que Pueden ser de dos tipos: gráficos de llamadas estáticos y dinámicos. Los gráficos de llamadas dinámicos brindan información sobre Las llamadas a procedimientos de un programa durante su ejecución. Muestra la secuencia de funciones y métodos. Llamadas y los parámetros que se pasan a cada procedimiento en la secuencia. Solo gráficos de llamadas estáticas. Proporcionar información sobre las posibles rutas de ejecución que puede tener un programa en función de la información Disponible en tiempo de compilación. Por lo tanto, un gráfico de llamadas estático no refleja con tanta precisión las llamadas reales. en un programa, particularmente si el programa es complejo [188]. Un total de seis artículos incluidos en este La encuesta utilizó gráficos de llamadas estáticos [20, 22, 25, 35, 79, 96].

En el Apéndice A.2, proporcionamos un ejemplo de un gráfico CFG, ICFG, DFG, PDG y de llamadas para mostrar Cada representación proporciona un tipo diferente de información para el mismo fragmento de código.

5.2.6 Gráfico de dependencia del sistema. Dos artículos representaron el código fuente utilizando gráficos de dependencia del sistema (SDG) [109, 110]. Un SDG es un gráfico con múltiples PDG conectados a través de Relación llamada-llamado dada por un grafo de llamadas. Los SDG extienden los PDG al describir la interprocedimentalidad. relaciones entre los puntos de entrada del programa5 y los procedimientos que llaman [109]. Para conectar Los PDG, hay nodos y bordes adicionales que dictan los parámetros de entrada reales y Valores de salida reales de un procedimiento. Cada argumento pasado tiene un nodo de entrada real y un nodo de entrada formal. nodos conectados por la arista de entrada de parámetros −→ . Cada parámetro modificado y El valor devuelto tiene un nodo de salida real y un nodo de salida formal que están conectados por el Borde de salida de parámetro −→ . Los nodos de entrada y salida formales dependen del control del nodo de entrada. y los nodos de entrada y salida reales dependen del control del nodo de llamada. Este paso de parámetros El modelo garantiza que los eventos interprocedimentales de un procedimiento se propaguen por los sitios de llamada.

5.2.7 Porciones de programa. Una porción de programa [189] es un subgrafo de un PDG o SDG que incluye solo Los nodos relevantes para un cálculo en un punto específico del programa. Este subgrafo se calcula utilizando un criterio de corte , que denota una variable de interés en un programa punto. Estos cortes se pueden calcular de forma inversa o progresiva. Un corte inverso incluye todos los nodos que pueden afectar el valor de en el punto del programa. Un corte hacia adelante incluye todos Nodos afectados por la variable en el punto del programa. Los segmentos del programa fueron utilizados por Dos artículos [14, 105] para detectar vulnerabilidades. El criterio de segmentación se determina mediante declaraciones en El código que se considera vulnerable. Las declaraciones también podrían ser puntos donde se encuentran los valores. cambiado, lo que podría provocar que una llamada API sea vulnerable. Cheng et al. [105] utilizan un PDG y Realizar cortes hacia adelante y hacia atrás desde el nodo de interés. No se especifica en el trabajo de Chen et al. [14] si se utiliza un PDG o un SDG, pero el mismo criterio para hacia atrás y hacia adelante Se utilizan cortes (es decir, los cortes hacia adelante incluyen declaraciones que se ven afectadas por el nodo de interés y Los cortes hacia atrás incluyen declaraciones que afectan al nodo de interés).

5.2.8 Gráfico de flujo de datos cruciales. Un gráfico de flujo de datos cruciales (CDFG), introducido por Wu et al. [101], es un subgrafo de un DFG que contiene solo la información crucial del DFG que podría desencadenar una vulnerabilidad de reentrada en contratos inteligentes. Los nodos cruciales son variables que contienen información sensible o crítica, y que tienen un flujo de datos directo a otro nodo crucial. Un CDFG se define como = (,), donde son los nodos cruciales y los bordes representan los

5Los puntos de entrada son las funciones/métodos que inician la ejecución del programa (por ejemplo, la función main() en C).

Machine Translated by Google

Relación de flujo de datos. Por ejemplo, = nodos, −→ indica que y Ambos son cruciales
y que existe un flujo de datos entre las dos variables.

En el Apéndice A.2, proporcionamos un ejemplo de un SDG, una porción de programa y un CDFG como demostración de qué información se puede esperar de cada representación.

5.2.9 Gráfico de programa. Wang et al. [102] introdujeron el concepto de gráfico de programa, que es un gráfico dirigido = (,) en el que los nodos pueden ser declaraciones, identificadores (por ejemplo, función declaraciones o variables) o valores. Este gráfico tiene ocho tipos de aristas: aristas de flujo de control, aristas de datos bordes de flujo, protegidos por bordes, bordes de salto, bordes ComputedFrom, bordes NextToken, bordes LastUse, y los bordes LastLexicalUse. Un borde de flujo de control indica que se puede ejecutar después

· Un borde de flujo de datos indica que se utiliza una variable que ha sido definida por

· Un borde protegido = −→ indica que solo se ejecuta si la expresión en
se evalúa como verdadero (lo cual es útil para identificar operaciones que pueden estar en el orden incorrecto). Un salto borde = −→ indica que tiene una dependencia de control de . Un borde ComputedFrom
= −→ Indica que es o contiene una variable utilizada en una expresión en A .
La arista NextToken = −→ indica que es un sucesor de (es decir, sigue) donde indica ,
y son nodos terminales o tokens del AST. Una arista LastUse indica que es una sentencia = −→
if . utiliza la misma variable que se utiliza en . = −→
utiliza la misma variable que se utiliza en si

5.2.10 Gráfico de propiedades de código. Utilizado en cinco artículos [12, 97–100], un gráfico de propiedades de código (GPC) Es una combinación de AST, CFG y PDG [52]. Fue introducido por primera vez por Yamaguchi et al. [52]. específicamente como una forma de detectar vulnerabilidades en programas C/C++ mediante análisis estático. La forma en que un El CPG se genera tomando el AST, CFG y PDG de un programa y modelándolos como propiedades. gráficos, y luego estos modelos se combinan conjuntamente conectando nodos de declaración y predicado. Un CPG se define formalmente como = (, , ,), que es un multigrafo dirigido, etiquetado y atribuido, con nodos , aristas y una función de mapeo de propiedades . Es conjunto
El conjunto de nodos de un CPG son los nodos de un AST. El conjunto de aristas de un CPG tiene tres particiones: control bordes de flujo, bordes de dependencia del programa y bordes AST. × Una dependencia de flujo de .

del programa. Una arista de dependencia del programa = indica que tiene un programa
borde de dependencia de . Un borde AST indica que es sintácticamente
relacionado con . La función de etiquetado de bordes: −→ Σ asigna una etiqueta del alfabeto Σ a cada borde en . La función : ( ) × −→ aplica propiedades a nodos y bordes, donde es el conjunto de claves de propiedad y es el conjunto de valores de propiedad. Dado que un CPG es una combinación de tantos Representaciones, proporciona una comprensión muy sólida del código. Otras implementaciones de un CPG mejorarlo añadiendo información de un DFG [63].

5.2.11 Gráfico de propiedades de código simplificado. Si bien los CPG pueden capturar información semántica y La información sintáctica también es muy compleja de crear. Generar un PDG por sí solo tiene una complejidad. de O(n ^{2}). El tamaño de los gráficos también es bastante grande, un ejemplo tiene 52 millones de nodos y 87 millones de aristas [52]. Para resolver este problema, dos artículos implementaron una propiedad de código simplificada Gráfico (SCPG) [106, 107]. Un SCPG solo utiliza bordes de un AST y un CFG, como dependencia de datos. se puede aproximar a partir de estos dos gráficos. Los nodos en el SCPG tienen dos valores: los tokens de código y el tipo de nodo. Eliminar la necesidad de generar un PDG reduce considerablemente el costo de generar esta representación, ya que solo sería necesario generar el AST y el CFG.

5.2.12 Grafo de propiedades. Un grafo de propiedades [45] es una variante de un CPG, definido como (, , = Λ, Σ, , ,). Aquí, las aristas y los nodos son los mismos que en el CPG. La función de adyacencia
: → × asigna cualquier arista a un par ordenado de sus vértices de origen y destino. La función

Machine Translated by Google

: → Λ asigna cualquier vértice dado a sus respectivos atributos Λ, y : → Σ es el atributo
función para los bordes, que, al igual que , asigna cualquier borde dado a sus respectivos atributos Σ.

5.2.13 Gráfico de Agregación de Código. Un Gráfico de Agregación de Código (CAG) se construye a partir de una combinación de un AST, CFG, PDG, árbol dominador y árbol postdominador. Un CAG se define formalmente como = (,), que es un multigrafo dirigido, etiquetado y atribuido, con nodos y aristas donde × . El conjunto de nodos en un CAG son los nodos de un AST. El conjunto de aristas en un CAG tiene cinco particiones: bordes de flujo de control × , bordes de dependencia de programa ×, bordes de árbol dominador , AST y bordes de árbol post-dominador × × × Una dependencia de flujo de control = −→ indica que puede fluir a en el siguiente .

del programa. Una arista de dependencia del programa = −→ indica que tiene un programa
borde de dependencia de . Un borde AST indica que es sintácticamente
relacionado con Un árbol dominador borde = −→ indica que la operación está dominada
por y todos los dominadores (es decir, todas las rutas desde el nodo de entrada hasta el primer paso).
Un borde de árbol post-dominador indica que la operación es post-dominada
por , lo que significa que todas las rutas desde hasta el nodo final deben pasar por . Usando un dominador
El árbol y el árbol postdominador permiten que esta representación capture mejor la información semántica en código fuente. Esto, a su vez, permite que los modelos tengan un mejor rendimiento en la detección de vulnerabilidades. Nguyen et al. [114] señalan cierta información que un CFG y un AST en particular no logran capturar, y cómo un árbol dominador y un árbol post-dominador pueden describir mejor estos atributos.

5.2.14 Gráfico de flujo de valor. Un gráfico de flujo de valor (GV) es similar a un GDP en que muestra La dependencia interprocedimental del programa. Las aristas, al igual que en un PDG, describen el flujo de control. y dependencia de los datos del programa [190]. Un VFG = (,) es un gráfico etiquetado dirigido, con nodos y aristas . El conjunto de nodos es un par (1, 2) en el que hay un nodo del 1 grafo acíclico predirigido y es un nodo del grafo acíclico posdirigido. Ambos representan el mismo valor. El conjunto de aristas en un VFG es un par (, el grafo de es el y 1 2 predecesor de (′), el grafo de flujo de nodos de la arista de conexión [191]. ′) tal que ( ), el flujo del nodo , ′, y los valores se mantienen a lo largo

El artículo [44] que utilizó un VFG utiliza un proceso especial que selecciona y preserva rutas de flujo de valor factibles para reducir la cantidad de datos necesarios para entrenar modelos para vulnerabilidad basada en rutas. detección. Esto hace que su método sea más liviano que un VFG típico.

En el Apéndice A.2, proporcionamos un ejemplo de un CPG, un CAG y un VFG para el mismo código fuente. para resaltar las diferencias en el tipo de información proporcionada por cada representación.

5.2.15 Gráfico de dependencia de componentes. Un gráfico de dependencia de componentes (CDG) [27] representa las relaciones entre los diferentes componentes de un gráfico y se creó para capturar Lógica del programa de la aplicación Android. El CDG se define formalmente como = (,), que es un código etiquetado dirigido. Grafo con nodos y aristas . El conjunto de nodos en un CDG representa los componentes de la aplicación Android (es decir, Actividad, Servicio o Receptor de transmisión). El conjunto de bordes en un CDG representa la relación de activación entre los componentes. Una arista = −→ indica que el componente Podría activar el inicio del ciclo de vida del componente.

5.2.16 Gráfico de comportamiento de componentes. Un gráfico de comportamiento de componentes (CBG) [27] representa La lógica de flujo de control o duración de las funciones API relacionadas con permisos en Java o Android. programa, así como las funciones realizadas en un recurso específico para cada componente. Esto es la segunda mitad del CDG, donde ambas representaciones se unen para describir completamente el Aplicación de Android. Hay cuatro tipos de nodos, cada uno indicando el tipo de componente en esa parte. del gráfico. Los bordes que conectan el CBG demuestran la lógica del flujo de control entre la API funciones y recursos sensibles.

Machine Translated by Google

El CBG = (,) es un grafo etiquetado dirigido con nodos y aristas . El
El conjunto de nodos en un CBG se divide en cuatro tipos: nodo raíz, nodos de función del ciclo de vida
, nodos de funciones API relacionados con permisos y nodos de recursos sensibles. Un nodo de inicio
representa el componente en sí. Un nodo de función de ciclo de vida representa el
Lógica de programación en tiempo de ejecución. Cada nodo de funciones de API relacionadas con permisos denota
una función de API relacionada con permisos, por ejemplo, la API sendTextMessage() de Android. Una función sensible
El nodo de recursos indica datos confidenciales a los que accede un componente. El conjunto de aristas en un
CBG representa la lógica del flujo de control de las funciones API del marco y los recursos sensibles.
Los bordes CDG están en el mismo bloque de flujo de control, indica que si y
luego se ejecuta inmediatamente después sin ejecuciones intermedias, o si y están en dos
bloques de flujo de control continuo (el nombre es la última función), respectivamente), entonces
El nodo en es el primer nodo en En el Apéndice .
A.2, proporcionamos un ejemplo de un CDG y un CBG para demostrar cómo funcionan
juntos.

5.2.17 Gráfico de flujo de control interprocedimental contextual. El gráfico de flujo de control interprocedimental contextual (CICFG) es una extensión del ICFG y describe el flujo de control completo.
en todas las instrucciones, incluido el contexto [19]. Un contexto define la información necesaria para una
operación que se produzca. El CICFG se define formalmente como = (, ,). Los nodos son básicos
bloques y los bordes son flujos de control intraprocedimentales o relaciones de llamada desde un
nodo. Por último, es un conjunto de contextos a través de los cuales se puede llegar a cada nodo [19].
La principal diferencia entre el CICFG y el ICFG es que el CICFG proporciona una visión más detallada
análisis de un programa porque el contexto permite diferenciar entre diferentes instancias o
Rutas por las que se puede llamar a una función. Dos ejemplos de contexto son conscientes del usuario y no conscientes del usuario.
que indica si el usuario es consciente de qué operaciones o recursos utiliza una aplicación o pieza
del código está usando. Por ejemplo, si una aplicación usa la ubicación del usuario, en un contexto que lo reconoce,
El usuario sabe que la aplicación está usando su ubicación, mientras que en un contexto en el que el usuario no lo sabe,
no saben que la aplicación utiliza la ubicación del usuario [26].

5.2.18 Gráfico de dependencia de API contextual. El Gráfico de dependencia de API contextual (CADG)
se construye a partir de un CICFG. No todos los nodos del CICFG están relacionados con la seguridad ni invocan una vulnerabilidad sensible.
API. El CADG es una abstracción del CICFG que se centra únicamente en la API sensible a la seguridad.
invocaciones [19]. Un CADG = (, ,) es un gráfico etiquetado dirigido, con nodos , aristas
, y función de etiquetado. El conjunto de nodos en un CADG representa los bloques básicos del
programa. El conjunto de aristas en un CADG representa el flujo de datos entre los bloques básicos. Un CADG
El borde indica que la función de etiquetado: utiliza datos que han sido definidos por el bloque básico .
→ Σ asocia nodos con las etiquetas de la API contextual correspondiente
operaciones. Cada etiqueta consta del prototipo de API, el punto de entrada y el parámetro constante [192].

5.2.19 Gráfico de contrato/semántico. Un gráfico de contrato [103] (o un gráfico semántico [108]) es una
representación creada específicamente para la detección de vulnerabilidades en contratos inteligentes. El conjunto de nodos en un
El gráfico de contrato/semántico se divide en tres tipos: nodo central, nodos normales y , ,
nodos de reserva. Un nodo central representa las invocaciones y variables clave que intervienen
un papel crucial en la detección de vulnerabilidades. Un nodo normal representa invocaciones y
variables que pueden ayudar a detectar vulnerabilidades, aunque no tienen la misma importancia
como nodos centrales. Un nodo de reserva simula la función de reserva que se incurre en un contrato.
ataque.

El conjunto de aristas en un gráfico de contrato/semántico tiene tres particiones: aristas de flujo de
control (×) (×), bordes del flujo de datos (×) (

Machine Translated by Google

Las aristas de reserva indican que el flujo se dirige al siguiente paso del programa. Una arista de flujo de datos poder
= indica que recibe datos de .

Un borde de respaldo indica que es el nodo de respaldo
y es la invocación call.value, que es la invocación en un contrato inteligente que puede causar una
vulnerabilidad de reentrada, si el código es vulnerable [108]. También puede significar que es la función
bajo prueba, y es el nodo de respaldo. Generalmente, este borde indica interacciones con el respaldo.
función [46].

En el Apéndice A.2, proporcionamos un ejemplo de un CADG y un gráfico de contrato/semántico para un fragmento
del código fuente para demostrar cómo se abstrae el código en estas formas gráficas.

5.2.20 Gráfico de dependencia de permisos contextuales. El gráfico de dependencia de permisos contextuales
El gráfico (CPDG) [26] también se construye a partir de un CICFG. El CPDG es una abstracción del CICFG que solo
se centra en la funcionalidad relacionada con los permisos de Android [26]. Un CPDG = (,) es un dirigido
Gráfico etiquetado, con nodos y aristas . Los nodos en un CPDG representan la estructura básica del programa.
bloques cuya funcionalidad se relaciona con el uso de permisos de Android. Los bordes en un CPDG representan
Flujo de datos entre los bloques básicos. Una arista CPDG = −→ indica que hay una ruta
de a en el CICFG, y que ambos nodos están en la misma función. es el conjunto de etiquetas
representando los permisos en cuestión. es un conjunto de contextos a través de los cuales cada nodo es el
Se podría alcanzar el CPDG [26].

5.2.21 Gráfico de dependencia contextual de origen y sumidero. El gráfico contextual de origen y sumidero .
El gráfico de dependencia (CSSDG) [26] también se construye a partir de un CICFG. El CSSDG es una abstracción de
el CICFG que considera únicamente los nodos cuya funcionalidad está relacionada con el uso de fuentes y sumideros.
[26]. Las fuentes son donde los datos confidenciales ingresan a un programa, y los receptores son donde se realizan las tareas de seguridad.
Operaciones críticas. Este flujo de datos sensibles podría ser un punto de vulnerabilidad si los datos no están...
manejado correctamente [96]. Por lo tanto, un CSSDG = (, , ,) es un grafo etiquetado dirigido, con nodos y
aristas . Los nodos en un CSSDG representan los bloques básicos del programa cuya funcionalidad
está relacionado con el uso de fuentes y receptores, mientras que los bordes representan el flujo de datos entre los elementos básicos.
bloques. es el conjunto de etiquetas que representan las fuentes y los sumideros en cuestión. es un conjunto de contextos
a través del cual se puede llegar a cada nodo del CSSDG [26].

5.2.22 Gráfico de propiedades de corte. Los gráficos de propiedades de corte fueron propuestos por Zheng et al. [111] y
El objetivo es preservar la información semántica y estructural relevante para las vulnerabilidades. También
Su objetivo es eliminar información irrelevante para reducir la complejidad de los gráficos. El gráfico utiliza
SyVC (Candidatos de vulnerabilidad basados en sintaxis) como criterio de segmentación para extraer los nodos de segmentación que
son relevantes para las vulnerabilidades. Luego, los bordes del CPG se utilizan como bordes entre los nodos en
El SPG.

5.2.23 Grafo de Tokens. Los grafos de tokens [112] se construyen a partir de tokens, conectándolos mediante una
construcción centrada en índices. Un grafo de tokens = (,) es un grafo dirigido, con nodos y aristas.
donde × . El conjunto de nodos en un grafo de tokens representa tokens individuales de
El código fuente. Por ejemplo, un conjunto de nodos puede ser , , ==, . El conjunto de aristas en un grafo de tokens
Define una relación de coocurrencia entre tokens. Las coocurrencias describen las relaciones.
entre tokens que aparecen dentro de una ventana deslizante de tamaño fijo [193].

5.2.24 Cadena de propagación. Una cadena de propagación [104] existe cuando hay una secuencia de código
entre varios fragmentos de código específicos. La secuencia tiene datos y control directos o indirectos.
dependencias entre fragmentos de código adyacentes. El conjunto de la cadena de propagación (,) denota el conjunto de
cadenas de propagación entre dos fragmentos de código a y b. Cada fragmento de programa tendrá propagación
cadenas que lo afectan y las cadenas de propagación que se ven afectadas por él. En términos de detección de vulnerabilidades,

Machine Translated by Google

Una cadena de propagación vulnerable o defectuosa denota una secuencia de código desde el código vulnerable hasta la salida vulnerable del programa. El conjunto de la cadena de propagación de defectos, denominado conjunto de la cadena de propagación de defectos, se denota como (,) desde (,) hasta el código de la cadena de fallo de propagación . Las cadenas de propagación pueden construirse mediante relaciones de flujo de datos o de flujo de control. Zhang et al. [104] utilizan relaciones de flujo de datos para crear las cadenas de propagación para contratos inteligentes. En este caso, el grafo de flujo de datos se define como un conjunto de nodos y aristas = (,), donde los nodos representan variables en un contrato inteligente y el conjunto de aristas denota las relaciones de dependencia entre ellas. Por ejemplo, = tiene una relación o dependencia de datos con denota que

.

5.3 Representaciones léxicas. Las

representaciones léxicas describen representaciones centradas en palabras y vocabulario. Estas representaciones no muestran relaciones entre nodos, como ocurre en las representaciones gráficas. Las representaciones léxicas también se basan principalmente en el trabajo de PLN.

5.3.1 Tokenizador. Un tokenizador, también conocido como representación léxica, toma el código fuente y crea tokens individuales para cada palabra o símbolo [63]. Esto se basa principalmente en técnicas de PLN existentes. Si bien existen diversos algoritmos de tokenización (p. ej., SentencePiece), la mayoría de los artículos estudiados no especifican qué algoritmo se utilizó, más allá de simplemente indicar que el código fue tokenizado. Algunos artículos, en cambio, especifican la biblioteca utilizada para tokenizar el código; por ejemplo, el tokenizador de Python [117], el tokenizador de javalang [40], ANTLR [15], js-tokens [120], Clang [137] y phply [86].

En el caso de los artículos que indicaron explícitamente su algoritmo de tokenización, encontramos artículos que utilizaban tokenización de subpalabras Byte Pair Encoding (BPE) [119, 122], SentencePiece [54], tokenizadores a nivel de carácter [39, 54], WordPiece [129, 138], tokenizadores de espacios en blanco [133] y tokenizadores a nivel de oración [87].
La tokenización de subpalabras BPE es un método que divide palabras completas en partes más pequeñas para comprimir los datos tokenizados. Las palabras frecuentes se representan como tokens individuales, mientras que las poco frecuentes se dividen en múltiples tokens de subpalabra. Por ejemplo, si el par de tokens "a" y "b" se repite con frecuencia, se combinarán y formarán el token único "ab" [194]. SentencePiece es otro método de tokenización de subpalabras que emplea tokenización sin pérdidas, donde toda la información necesaria para reproducir el texto normalizado se conserva en la salida del codificador (es decir, trata el texto de entrada como una secuencia de caracteres Unicode [195]). WordPiece tokeniza una palabra mediante MaxMatch, un proceso que consiste en seleccionar iterativamente el prefijo más largo del texto restante que coincida con un token de vocabulario hasta que se segmenta toda la palabra [196]. En el Apéndice A.3, proporcionamos un ejemplo de un tokenizador de subpalabras BPE, un tokenizador estándar, SentencePiece y WordPiece para el mismo código fuente, para demostrar las diferencias entre los algoritmos.

Los tokenizadores que utilizan algoritmos de incrustación robustos, como word2vec [57] , pueden capturar el significado semántico del código. Cuando las incrustaciones vectoriales se crean a partir del tokenizador, estos números se basan principalmente en la relación semántica con otra palabra. En otras palabras, si una palabra está semánticamente relacionada con otra, sus representaciones vectoriales serán similares [57]. Estas técnicas pueden ser particularmente útiles cuando el modelo necesita aprender la semántica de un fragmento de código para completar la tarea en cuestión. También es bastante sencillo tokenizar el código fuente, con una complejidad de ( ). Una función integrada para prácticamente cualquier lenguaje simplemente tomará una línea de texto y la dividirá en tokens según un delimitador proporcionado. Modelos como word2vec y doc2vec [57, 197] están muy desarrollados y son una excelente manera de crear incrustaciones de palabras a partir de un vocabulario. Esta es una de las razones por las que esta representación también es tan popular. Sin embargo, los tokenizadores no capturan las propiedades estructurales del código fuente y, por lo tanto, esta representación carece de la capacidad de comprender la sintaxis de un programa.

Machine Translated by Google

5.3.2 iSeVC y sSyVC. Los sSyVC (candidatos a vulnerabilidad basados en código fuente y sintaxis) son características del código que presentan algunas características de sintaxis de vulnerabilidad. Un ejemplo serían las vulnerabilidades asociadas a punteros. Un sSyVC sería una línea de código que contiene un asterisco (*), ya que este símbolo es el que se utiliza al tratar con punteros [158]. Estas características se obtienen mediante AST. [158]. Los iSeVC (candidatos a vulnerabilidad basados en código intermedio y semántica) se derivan de sSyVC mediante segmentación de programa. Los sSyVC son los nodos de interés, y el PDG del programa permite realizar la segmentación hacia delante y hacia atrás, como se describe en la Sección 5.2.7. El conjunto resultante de sentencias ordenadas, todas conteniendo datos o dependencias de control entre ellas, son los iSeVC [158]. Los iSeVC contienen información sobre la dependencia de los datos y el control, de ahí su nombre que los relaciona con la semántica [48].

5.3.3 Fragmento de contrato. Un fragmento de contrato [47] contiene sentencias o líneas clave del programa de un contrato inteligente que podrían inducir una vulnerabilidad. Estos fragmentos de contrato buscan ser altamente expresivos para que se puedan extraer características más relevantes. Todos los fragmentos de contrato están semánticamente relacionados por la dependencia del flujo de control y todos resaltan un elemento clave (call.value) en la detección de reentrada (en la que se centra el artículo en el que se propone esta representación). Los fragmentos de contrato pueden generarse mediante análisis del flujo de control. Una vez creados, se tokenizan y se transforman en vectores de características.

5.4 Representaciones Varias Las representaciones varias son aquellas que no encajan en ninguna de las categorías anteriormente descritas.

5.4.1 Imagen. Trabajos previos utilizaron técnicas ya muy desarrolladas de análisis de imágenes para analizar aspectos del código y detectar vulnerabilidades [87, 92] y malware en aplicaciones Android [21].
La idea central de este método es aprovechar los patrones visuales del software para detectar anomalías o similitudes. Esta técnica permite a los investigadores aprovechar las técnicas desarrolladas para la detección de elementos en imágenes regulares.

5.4.2 Métricas de Código. Las métricas de código son una medida cuantitativa que relaciona ciertas características con un valor numérico, concretamente el número de veces que aparecen [198]. Las métricas de código pueden definirse de forma diferente para distintas tareas. Algunas métricas comunes incluyen líneas de código, rotación de código (es decir, la frecuencia con la que se modifica el código), entre otras. Trabajos previos también introducen nuevas métricas para un propósito específico, como la inyección SQL [156]. En lugar de líneas de código u otras métricas clásicas que no serían útiles para la detección de inyecciones SQL, métricas como el número de puntos y comas, la presencia de condiciones siempre verdaderas y el número de comandos por sentencia proporcionan información más relevante que permite predecir mejor la inyección SQL. Estas métricas pueden estar relacionadas con un factor de riesgo que determina el grado de impacto que la métrica podría tener en el código para crear un problema de seguridad [198].

5.4.3 Gadgets de Código. Los gadgets de código son esencialmente un método para describir o representar una porción de programa. Constan de varias sentencias o líneas de código ordenadas, semánticamente relacionadas entre sí por dependencia de datos o controles [41]. Los gadgets de código se crearon precisamente para la detección de vulnerabilidades [41], lo que explica su popularidad en las tareas de seguridad.

5.4.4 Secuencias de códigos de operación. Un código de operación especifica la operación que debe completarse para una instrucción [199]. En particular, especifican la operación de nivel más bajo que debe completarse, como PUSH, MSTORE y CALLVALUE [164]. Estas características permiten comprender a bajo nivel la función del código. Los códigos de operación se aprenden como vectores, y Liao et al. [164] utilizan n-gramas y

Machine Translated by Google

word2vec[57] para aprenderlos como incrustaciones. Dado que los códigos de operación ya dictan operaciones en las computadoras, esta representación es fácil de generar.

5.4.5 Expresión regular. Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda, a menudo utilizado para la coincidencia de cadenas o la manipulación de texto dentro de ellas según reglas específicas. Un artículo representó el código fuente como expresiones regulares, que codifican información sobre las llamadas a la API en el código. Específicamente, el trabajo descrito por Liu et al. [180] primero toma una aplicación Android y crea un CFG a partir de las devoluciones de llamada. El CFG se transforma en un ICFG, se reduce a un grafo de API y luego se utiliza un algoritmo de autómatas a expresiones regulares para generar las expresiones regulares. Esta solución se utiliza para la clasificación de malware multifamiliar y aborda los problemas de reconocimiento de patrones de comportamiento de familias de malware, ofuscación de código y variantes polimórficas que los atacantes suelen usar para evadir la detección. Las expresiones regulares describen los patrones de comportamiento de las familias de malware. Si bien este método puede ser computacionalmente costoso, ya que se deben crear tres grafos antes de transformarlo en una expresión regular, permite capturar las diferencias entre las familias de malware.

5.4.6 Información de la aplicación. Si bien esta representación puede clasificarse según las métricas de código, las características extraídas por Koli [23] se clasifican con mayor precisión como información de la aplicación. En esta representación, se aplica ingeniería inversa a una aplicación Android para extraer los archivos Java originales y el XML de Android. De estos archivos, se extraen las llamadas a la API realizadas y los permisos utilizados. También se extraen otras características, como el código criptográfico o la base de datos que especifica ciertas características del código que podrían estar asociadas con malware [23] . Esta representación es un método de extracción de características. Una vez extraídas, estas características se transforman en vectores de características y se les asigna una etiqueta como benignas o maliciosas. Posteriormente, estos datos pueden utilizarse en un clasificador de aprendizaje automático.

5.4.7 Llamadas API. Wang et al. [24] utilizan llamadas API extraídas del código fuente de una aplicación Android, junto con permisos extraídos del archivo de manifiesto de Android. Este artículo utiliza la herramienta DroidAPIMiner para extraer las 20 llamadas API más comunes que realizan las aplicaciones maliciosas. El uso de estas llamadas API y permisos como características para entrenar un modelo de aprendizaje profundo permite [24] detectar malware en aplicaciones Android. De nuevo, esta representación es un método de extracción de características, que se convierten en vectores que posteriormente se pasan a un modelo.

Resultados de RQ1:

- Hay 39 representaciones que ofrecen una variedad de información sobre el código fuente, aunque un objetivo común es capturar la información semántica y sintáctica en el código.
- Hay 24 representaciones únicas basadas en gráficos, la categoría de representación de código fuente más popular, ya que muestra las relaciones entre diferentes nodos (por ejemplo, líneas o declaraciones) y cómo interactúan.
- Aunque existe una mayor variedad de representaciones basadas en gráficos, 47 artículos utilizaron tokenizadores y 32 artículos utilizaron AST como sus representaciones.

Siete artículos [43–48, 101] proponen una representación específica para su aplicación (p. ej., un grafo de contratos [46] para detectar vulnerabilidades en contratos inteligentes). Estas representaciones específicas de la tarea o lenguaje no son generalizables a otros lenguajes ni propósitos.

6 RQ2: ¿Algunas tareas utilizan solo o mayoritariamente un tipo de representación?

La Figura 2 muestra las relaciones entre las representaciones y las tareas de ciberseguridad. Los AST y los tokenizadores fueron las dos representaciones más utilizadas para la detección de vulnerabilidades. Dada la popularidad de los LLM y el PLN en los últimos años, se deduce que un tokenizador sería una representación popular, ya que este es el método utilizado para las técnicas de PLN. La amplia disponibilidad de modelos preentrenados...

Machine Translated by Google

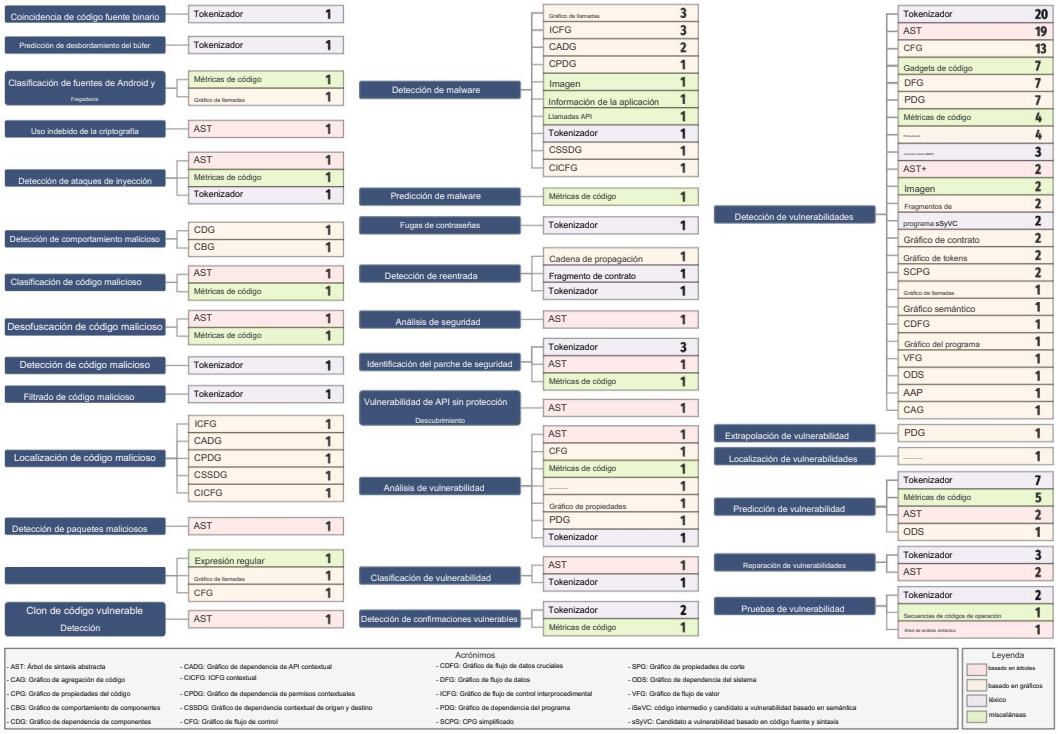

Fig. 2. Relación entre representaciones y tareas

Los modelos de PNL brindan a los desarrolladores el poder de ajustar fácilmente estos modelos en cualquier tarea que necesiten. deseo, sin un gran coste [200]. Los AST también son una representación relativamente rentable, ya que los compiladores Utilizar AST para representar la estructura del código fuente. Esto también permite a los desarrolladores generarlo. representación con relativa facilidad y al mismo tiempo proporciona detalles estructurales cruciales sobre el código fuente que pueden exponer vulnerabilidades [175].

Aunque AST y los tokenizadores fueron las representaciones más utilizadas, también encontramos que las técnicas de detección de vulnerabilidades utilizaban principalmente representaciones basadas en grafos. Las representaciones basadas en grafos fueron También es popular para otras tareas de ciberseguridad, a saber, localización de código malicioso, detección de malware, Localización de vulnerabilidades, análisis de vulnerabilidades, clasificación de malware y extrapolación de vulnerabilidades . Entre las representaciones basadas en gráficos, CFG es la más popular. Esto podría deberse a... Detalla el flujo de ejecución del programa, lo que puede ser útil para detectar si existe una vulnerabilidad. Esto podría ocurrir debido a la estructura del programa. Un desarrollador podría elegir un CFG en lugar de otro. representación, como un AST, porque proporciona información más detallada sobre la fuente Código. Por ejemplo, comparado con un gráfico de llamadas de un programa del conjunto de datos utilizado en Mester y Bodó [35], el CFG tiene 150.000 vértices/nodos, mientras que el grafo de llamadas tiene 10.000 nodos. Mientras que computacionalmente más costoso que otros métodos (por ejemplo, tokenizador o AST), la información La información proporcionada por el gráfico permite una comprensión más sólida del código.

DFG es otra representación popular, especialmente para la detección de vulnerabilidades. Al igual que los CFG, Los DFG detallan el flujo del programa, aunque también detallan el flujo de datos. Esto también puede ser útil en detección de vulnerabilidades porque si una entrada de datos dañinos llega a un punto de programa sensible a la seguridad, Entonces es probable que se produzca una vulnerabilidad. Por lo tanto, se puede optar por un DFG en lugar de un CFG para detectar una Un tipo específico de vulnerabilidad relacionada con los flujos de datos. Las representaciones basadas en gráficos son útiles para estos casos. tipos de tareas porque detallan la estructura general y el flujo del código. Para encontrar receptores, o puntos en

Machine Translated by Google

En el código donde se invoca una función peligrosa, es útil comprender la estructura, ya que se puede identificar con precisión qué función inicia o invoca la vulnerabilidad. Además, según el tipo de información compartida dentro del grafo, se pueden identificar mejor las áreas de código que contienen una vulnerabilidad.

Los gadgets de código, iSeVC y sSyVC, fueron representaciones diseñadas específicamente para técnicas de detección de vulnerabilidades. Estas representaciones se centran especialmente en candidatos a vulnerabilidades o fragmentos de código potencialmente vulnerables. Esta vista ampliada permite un enfoque más específico para la detección de vulnerabilidades. Se puede optar por esta representación para un enfoque más sencillo del problema en comparación con un método que consume más recursos, como un CPG.

Los ICFG, los gráficos de llamadas y los CADG son las representaciones preferidas para la detección de malware. Los ICFG describen el flujo de control completo de un programa, y los CADG se derivan de una versión de los ICFG que incluye contexto. Ambas representaciones proporcionan información sobre posibles invocaciones relacionadas con la seguridad, que podrían permitir que el malware afecte al sistema. Comprender qué llamadas se realizan y cómo fluyen las instrucciones en un programa puede brindar información sobre cualquier operación irregular o insegura realizada por el código. Los ICFG son conocidos por su robustez frente a los métodos de evasión y ofuscación utilizados por el malware [18]. Esto podría deberse a que el ICFG ofrece una vista de todo el programa, y el malware suele interactuar con todo el sistema, más allá de un solo procedimiento Más bien, el malware puede llamar a funciones fuera de un procedimiento, y el ICFG brindaría información sobre estas llamadas, así como sobre cualquier actividad inusual o inesperada [26].

Al elegir una representación para una tarea específica, es importante considerar las necesidades y los recursos del desarrollador para la aplicación. Por ejemplo, un CPG es un grafo muy robusto y complejo de generar. Si el desarrollador necesita un grafo muy robusto para su tarea y cuenta con los recursos para desarrollarlo (para miles de muestras de código), un CPG sería una buena opción como representación. Sin embargo, si el desarrollador no dispone del tiempo ni los recursos para un grafo tan robusto, pero sabe que el problema que intenta detectar se relaciona con el flujo de datos a través de un programa (por ejemplo, la detección de ataques de inyección), podría optar por un DFG, ya que proporcionaría la información necesaria sin ser excesivamente complejo ni consumir muchos recursos.

Hallazgos de RQ2:

- Dado que la detección de vulnerabilidades es la tarea más popular, es la tarea que más representa. ciones se utilizan para.
- Se crearon ciertas representaciones (es decir, iSeVC, sSyVC y gadgets de código) para el propósito específico. tarea de detección de vulnerabilidades y, por lo tanto, solo se utilizan para esa tarea.
- Los gráficos de llamadas, ICFG y CADG son la representación preferida para la detección de malware.

7 RQ3: ¿Qué tareas de ciberseguridad están cubiertas por las técnicas basadas en ML?

Para comprender mejor los tipos de tareas que abarcan las técnicas descritas en este documento, clasificamos las diferentes tareas para que se ajusten a las nueve disciplinas del ciclo del Proceso Unificado Racional (RUP) [201]. El ciclo RUP es un marco de proceso de desarrollo de software que permite a los desarrolladores de software organizar y planificar mejor el proceso de desarrollo. En esta pregunta, clasificamos las tareas únicas de ciberseguridad encontradas en nuestra búsqueda en los nueve flujos de trabajo principales del ciclo RUP: modelado de negocio, requisitos, análisis y diseño, implementación, pruebas, despliegue, configuración y gestión de cambios, gestión de proyectos y entorno. Observamos que las tareas de ciberseguridad solo encajan en cinco de las nueve categorías: análisis y diseño, configuración y gestión de cambios, entorno, implementación y pruebas.

La figura 3 muestra cómo las diferentes tareas de ciberseguridad encajan en estas cinco disciplinas del ciclo RUP.

Machine Translated by Google

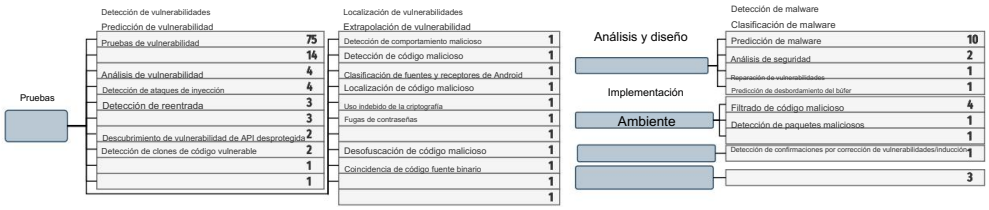

Figura 3. Tareas de ciberseguridad en el ciclo RUP

7.1 Análisis y diseño

El análisis y el diseño implican traducir los requisitos en un modelo formal del software, lo que da como resultado en una descripción del sistema que guía la implementación. Categorizamos estas tareas bajo análisis y diseño, ya que estas tareas analizan el código fuente sin necesariamente implementar un nuevo diseño.

Las tareas de esta categoría garantizan que, una vez que los desarrolladores puedan comenzar a implementar el sistema, No hay problemas de seguridad. La detección, predicción y clasificación de malware son tareas importantes. particularmente para aplicaciones Android [17–26, 35, 150, 180], pero también para cualquier sistema vulnerable a Malware. El malware permite a los atacantes aprovechar las fallas de seguridad en los sistemas. Comprender Los tipos de malware y ransomware, así como dónde ocurren, permiten tomar medidas para evitarlos. Estos problemas antes de implementar un sistema. El análisis de seguridad es una tarea que se utilizó particularmente para contratos inteligentes [175], pero se puede generalizar a cualquier sistema que tenga el potencial de ser comprometida.

7.2 Implementación

La implementación es la codificación real del modelo de software. Es aquí cuando el diseño del... La fase anterior cobra vida. La reparación de vulnerabilidades es una tarea que corrige vulnerabilidades en el código. Esta Encaja en la implementación porque en realidad está implementando la solución al código mientras el desarrollador está funcionando [30, 43, 115]. Además, hay algunas tareas involucradas en inferir o predecir Elementos de código, como la predicción de desbordamiento del búfer y la predicción de vulnerabilidades. es un tipo de vulnerabilidad, y Choi et al. [128] centran su enfoque de predicción en este tipo de vulnerabilidad en lugar de muchas otras. Estas técnicas de predicción permiten a los desarrolladores saber donde estos problemas pueden ocurrir en el código mientras lo implementan. Consideramos estas tareas como la fase de implementación porque la reparación de vulnerabilidad implica implementar una solución al problema. Sin embargo, las tareas de predicción también se consideran como implementación porque implican ser proactivo contra estas vulnerabilidades o problemas y encontrarlos antes de que puedan ser explotados.

7.3 Pruebas

Las pruebas implican ejercitar el software para detectar posibles fallas o inconsistencias de diseño, lo que ayuda a prevenir Problemas de seguridad costosos durante la producción. Las tareas de detección indican que las vulnerabilidades ya... existen en el sistema implementado. Por lo tanto, los sistemas se están probando para detectar estos problemas existentes. Se pueden resolver. Muchas técnicas de detección entran en esta categoría, como la criptografía. Mal uso, detección de código malicioso, filtrado, clasificación, desofuscación y localización; malicioso Detección de comportamiento, fugas de contraseñas, detección de ataques de inyección, detección de reentrada, vulnerabilidad análisis, detección, clasificación, localización, extrapolación y prueba; clonación de código vulnerable Detección y descubrimiento de vulnerabilidades en APIs desprotegidas. También se utiliza la comparación de código fuente binario. para tareas como la detección de malware y la evaluación de vulnerabilidades [202].

7.4 Configuración y gestión de cambios

La gestión de configuración y cambios rastrea y mantiene un proyecto a medida que evoluciona. tiempo. Garantiza que el código creado durante la implementación aún sea utilizable y pueda reutilizarse.

Machine Translated by Google

En otras partes del proyecto, si es necesario [201]. Los flujos de trabajo de desarrollo de software modernos suelen utilizar repositorios remotos para rastrear el código creado y sus cambios. Por lo tanto, las confirmaciones suelen ser el origen de una vulnerabilidad o un problema, y su solución. Zhou et al. [131] y Nguyen-Truong et al. [130] crearon técnicas para ayudar en la clasificación de las confirmaciones de seguridad, así como para identificar las confirmaciones que inducen o corrigen vulnerabilidades, incluyendo la detección de parches de seguridad. Debido a la naturaleza de estas tareas (p. ej., clasificar, identificar o detectar específicamente cambios relacionados con la seguridad), las clasificamos bajo la categoría de configuración y gestión de cambios, ya que están directamente relacionadas con cómo se modifica y reutiliza el código a lo largo de un proyecto.

7.5 Medio ambiente

El entorno se centra en el entorno de desarrollo de software necesario para que los ingenieros desarrollen el sistema. Esto incluye las técnicas y los procesos requeridos por los desarrolladores [201]. Los paquetes son una parte esencial del proceso de desarrollo. Proporcionan técnicas útiles que pueden simplificar enormemente la implementación de un sistema. Sin embargo, algunos paquetes pueden contener algún tipo de vulnerabilidad o malware que pueda comprometer la integridad del sistema. Por lo tanto, la detección de paquetes maliciosos es una tarea importante que puede proteger los sistemas de dicho software malicios
Clasificamos esta tarea en el entorno porque los paquetes son lo que los desarrolladores usan en su entorno como parte de su proceso de desarrollo. Si bien los paquetes maliciosos no necesariamente afectan directamente a su entorno, sí afectan el espacio en el que trabajan los desarrolladores.

A medida que las criptomonedas se vuelven un tema más popular y prevalente, la investigación también comienza a centrarse en la creación de técnicas basadas en aprendizaje automático (ML) para abordar problemas relacionados con las criptomonedas y los contratos inteligentes [46, 47, 79, 101, 103, 108, 164, 166, 175, 176]. Todos estos artículos se centran en la detección o prueba de vulnerabilidades y el análisis de seguridad de los contratos inteligentes. Dos artículos [47, 104] se centran en una vulnerabilidad específica denominada ataques de reentrada, específica de los contratos inteligentes. Esta vulnerabilidad permite que un atacante pueda retirar fondos de un contrato inteligente repetidamente y transferirlos [47].

Si bien la mayoría de las tareas son distintas, algunas están estrechamente relacionadas entre sí. Por ejemplo, la detección de comportamiento malicioso busca actividades y comportamientos sospechosos que puedan indicar un ataque, independientemente de si el ataque utiliza malware, mientras que la detección de malware suele buscar características o firmas específicas que se sabe que están asociadas con el malware [17]. Las tareas de detección se centran en la identificación general de una característica determinada (p. ej., la detección de vulnerabilidades se centra en encontrar ampliamente un segmento de código vulnerable), mientras que las tareas de localización se centran en señalar exactamente dónde, en una base de código más amplia, se produce la característica (p. ej., la localización de vulnerabilidades se centra en señalar la línea específica en la que se produce la vulnerabilidad). De forma similar, las técnicas de predicción se interesan en encontrar problemas antes de que sean evidentes en un sistema (p. ej., la predicción de vulnerabilidades se centra en encontrar vulnerabilidades antes de que se lancen oficialmente en un sistema, para evitar que un atacante tenga la oportunidad de descubrirlas cuando se lance), mientras que las técnicas de detección se interesan en identificar problemas que ya existen en un sistema en funcionamiento (p. ej., la detección de vulnerabilidades se centra en encontrar vulnerabilidades que ya están presentes en un sistema lanzado). Las pruebas de vulnerabilidad, sin embargo, se centran en evaluar un sistema para determinar si contiene fallas que podrían ser explotadas por actores maliciosos. El uso indebido de criptografía describe una tarea de detección, donde se descubren usos indebidos de las API de criptografía. Las API de criptografía tienden a ser difíciles de usar, por lo que esta tarea alerta a los desarrolladores sobre cuándo podrían haber cometido un error en su implementación [33]. La extrapolación de vulnerabilidades describe una tarea que busca encontrar vulnerabilidades potenciales basándose en vulnerabilidades existentes. Esto permite a los desarrolladores encontrar vulnerabilidades que podrían no existir o ser aún desconocidas basándose en vulnerabilidades pasadas y su estructura. Esta tarea, en particular, se centra en cómo las vulnerabilidades pueden propagarse o aparecer en contextos similares según las características de la vulnerabilidad [91]. El descubrimiento de vulnerabilid

Machine Translated by Google

Tabla 3. Idiomas admitidos por las técnicas existentes

Lang.	#Papeles	Lang.	#Papeles	Lang.	#Papeles	Lang.	#Papeles	Lang.	#Papeles	Lang.	#Papeles
C	81 (57,4%)	JS	C+ +	12 (8,5%)	Python	6 (4,3%)	C#	2 (1,4%)	Gecko	1 (0,7%)	Powershell 1 (0,7%)
SM	1 (0,7%)		50 (35,5%)	Solidity	12 (8,5%)	CSS	3 (2,1%)	SQL	2 (1,4%)	Go	1 (0,7%)
Ruby	Java	36 (25,5%)		1 (0,7%)	XML	1 (0,7%)	PHP	8 (5,7%)	Rust	3 (2,1%)	
TS	2 (1,4%)	HTML	1 (0,7%)	Smali		1 (0,7%)	XUL	1 (0,7%)			

JS = JavaScript; TS = TypeScript; SM = SpiderMonkey.

instancias en las que puede ocurrir una vulnerabilidad cuando se llama o utiliza una API no protegida en un sistema [174].

Hallazgos de RQ3:

— La detección de vulnerabilidades es, con diferencia, la tarea más popular, con 75 artículos centrados en ella. tarea.

— Algunos artículos se centran únicamente en detectar un tipo de vulnerabilidad, como el desbordamiento del búfer. predicción, detección de reentrada y detección de ataques de inyección.

—La mayoría de estas tareas encajan en la categoría de pruebas del ciclo RUP, lo que significa que estas técnicas están destinadas a evaluar la seguridad del código ya escrito antes de implementarlo .

8 RQ4: ¿Qué lenguajes de programación son el objetivo predominante de las técnicas basadas en ML para tareas de ciberseguridad?

La Tabla 3 muestra los lenguajes de programación que abarcan las técnicas existentes. Dos trabajos [35, 125] no especificaron el/los lenguaje(s) utilizado(s). Los artículos sin detalles del lenguaje o del conjunto de datos se etiquetan como «no especificado».

Observamos que C es el lenguaje más popular; 81 artículos (57,4%) desarrollaron técnicas para programas en C. C++ es el segundo lenguaje más común, cubierto por 50 artículos (35,5%), todos los cuales también admitían C. La popularidad de los lenguajes C y C++ podría atribuirse a dos factores principales: la disponibilidad de conjuntos de datos para tareas de ciberseguridad [203–206] y el mayor riesgo de vulnerabilidades relacionadas con la memoria en estos lenguajes [207].

También observamos que varias técnicas se centran en tareas de seguridad para aplicaciones Android [12, 12, 17–27, 35, 96, 150, 180]. Si bien las aplicaciones Android pueden escribirse con Kotlin y Java, los artículos estudiados en este estudio se centraron únicamente en aplicaciones escritas en Java. Además, con la creciente popularidad de los contratos inteligentes, varios artículos desarrollaron técnicas para Solidity, el lenguaje en el que se escriben.

También observamos que solo existen 6 técnicas para Python y 12 para JavaScript, lo cual es sorprendente dada su creciente popularidad entre los desarrolladores.⁶ Si se utilizan tanto en la práctica, cabría esperar que existiera un número asociado de técnicas que las cubran, especialmente para la asistencia relacionada con la seguridad. También es posible que no existan suficientes conjuntos de datos para estos lenguajes y, por lo tanto, no existan muchas técnicas para ellos, ya que los investigadores no pueden disponer de los datos necesarios para entrenar y probar los modelos. No obstante, esta es una deficiencia que debería abordarse en trabajos futuros.

Ninguna técnica era independiente del lenguaje, lo que significa que todos los sistemas creados se diseñaron solo para uno o posiblemente varios lenguajes. Desarrollar una herramienta independiente del lenguaje es difícil debido a la variedad de paradigmas de programación existentes y a que no todos los lenguajes siguen un paradigma en particular. Sin embargo, esto implica que los investigadores deben ser diligentes en el desarrollo de técnicas compatibles con lenguajes populares y de uso común.

6https://spectrum.ieee.org/los-mejores-lenguajes-de-programación-2023

Machine Translated by Google

Tabla 4. Diferentes modelos utilizados por los artículos encuestados

Tipo de modelo	Modelos
Transformador basado en secuencias	[43, 44, 54, 90, 99, 104, 115, 119, 122, 144], Transformada de Fourier discreta [51], Transformada neuronal convolucional Red [11, 21, 36, 38, 87, 92, 126 , 132, 139, 151, 161, 178], Red neuronal convolucional de pirámide profunda [145], CNN de texto [39], CNN temporal [14], red neuronal recurrente [38, 40, 90, 137, 158], RNN bidireccional [127, 173], codificadores automáticos [30], seq2seq [84, 144], BERT [135, 162], codeBERT [129, 138], JavaBERT [138], jerárquico Red de atención [89, 95, 172], doc2vec [141], word2vec [120, 141], Redes de autoatención [95], Red recurrente cerrada Unidad [174], Unidad Recurrente Bidireccional Cerrada [86, 160, 171, 178], Aprendizaje en Línea [19], Memoria a Largo Plazo y Corto Plazo [11, 117, 123, 134 , 135 , 152 , 159, 163, 165], LSTM bidireccional [13, 16, 41, 47, 95, 110, 135, 160, 169, 178], Párrafo Memoria distribuida vectorial [166], Unidad recurrente controlada convolucional basada en atención de aprendizaje profundo [124], RNN bidireccional para detección y localización de vulnerabilidades [48], clasificador pasivo-agresivo [18], Máquina de aprendizaje [150], codificador-decodificador [142], GPT [143]
Basado en gráficos	Red neuronal gráfica [12, 45, 46, 78, 80, 93, 98, 102, 105, 107, 108, 112–114, 118], red de creencias profundas [24], Red convolucional de grafos [35, 81, 86, 95, 113, 177], Red de atención de grafos [97], Red de atención de grafos de características Red convolucional [88], red convolucional de grafo recurrente [111], red neuronal de grafo cerrado [85, 99], Red neuronal de gráficos bidireccional [37], GraphCodeBERT [101]
Basado en árboles	Boosting [155, 156], Boosting de gradiente extremo [125], Máquina de boosting de gradiente ligero [125], adaboost[91], Árbol de decisión con aumento de gradiente [91], Bagging [156], Random Forest [20, 22, 23, 25, 27 , 53 , 82 , 83, 91, 109, 125 , 146, 148, 149, 155, 163, 164, 175], coForest [109], Árbol de decisión [23, 27, 83, 91 , 146, 148 , 153, 155, 163, 164, 175], code2vec [168], Naive Bayes [17, 23, 27, 53, 83, 125, 148, 153, 163], Árbol Naive Bayes aumentado [83], Resumen Red neuronal de árboles sintácticos [34], clasificador de árboles adicionales [36]
Redes neuronales	Perceptrón multicapa [22, 79, 82, 91 , 94, 107, 114, 136, 152, 170], Red neuronal [103, 163, 175, 180], Deep Red neuronal [11, 106, 146, 157, 167], Red neuronal profunda compleja [146], Árbol de sintaxis abstracta Red [34], Red neuronal de atención [100], Red neuronal de memoria [128], Red neuronal aleatoria [132]
Basado en características	Agrupamiento [91], agrupamiento de K-medias [141], agrupamiento de K-medianas [179], Bayes ingenuo [17, 23, 27, 53, 83, 125, 148, 153, 163], Bayes ingenuo gaussiano, Bayes ingenuo aumentado con árbol [83], regresión logística [17, 82, 83, 91, 94, 109, 125, 146, 148, 153, 154, 164], aprendizaje automático extremo [150], vecino más cercano [20], K-vecino más cercano [22, 91, 146, 153, 163, 164], regresión lineal [146], máquina de vectores de soporte [20, 23, 26, 27, 33, 91, 96, 116, 125 , 133, 140, 141, 146, 147, 154–156, 163, 164, 175], Variante de clasificación de vectores de soporte C de la máquina de vectores de soporte [146], Lineal Análisis discriminante [91, 156, 163], agrupamiento espacial basado en densidad de aplicaciones con ruido [141]

Resultados de RQ4:

— C es el lenguaje más común al que se dirigen las técnicas de ciberseguridad basadas en ML.
— A pesar de su popularidad, no existen muchas técnicas basadas en ML para Python y Javascript.
—Una gran parte de los artículos están destinados a resolver problemas de seguridad en las aplicaciones de Android. Por tanto, Java también es un lenguaje popular, con 36 técnicas orientadas a resolver estos problemas en Java.
— Dada la creciente popularidad de los contratos inteligentes, hay una serie de técnicas (12) que se crearon para Solidity, el lenguaje utilizado para crear contratos inteligentes.

9 RQ5: ¿Qué modelos se utilizan comúnmente con diferentes representaciones?

Para clasificar los diferentes tipos de modelos utilizados en los artículos, nos inspiramos en Siow et al. [208] y los clasificamos en cinco categorías: modelos basados en secuencias, modelos basados en características, modelos basados en árboles, modelos basados en grafos y redes neuronales. Si bien existe una superposición entre nuestras categorías de redes neuronales y otras, nuestra clasificación se basa en el tipo de entradas que aceptan los modelos. Por ejemplo, una red neuronal de grafos puede tener una arquitectura de red neuronal; sin embargo, está diseñada para manejar entradas basadas en grafos. La Tabla 4 muestra los modelos utilizados en los artículos analizados y cómo se clasifican en cada categoría. Por lejos, la mayoría de estos modelos están basados en secuencias, en particular las CNN, los Transformers y los LSTM. Esto probablemente se deba a su popularidad general, pero también a que estos modelos son muy potentes, pueden superar el problema del gradiente evanescente y pueden manejar dependencias a largo plazo [209].

Sin embargo, las máquinas de vectores de soporte (SVM) fueron, en general, el modelo más popular. Esto podría deberse a que, al determinar si el código presenta algún tipo de problema de ciberseguridad, como una vulnerabilidad, lo más útil para aprender o en lo que centrarse son las características del código. Por ejemplo, si un modelo puede

Machine Translated by Google

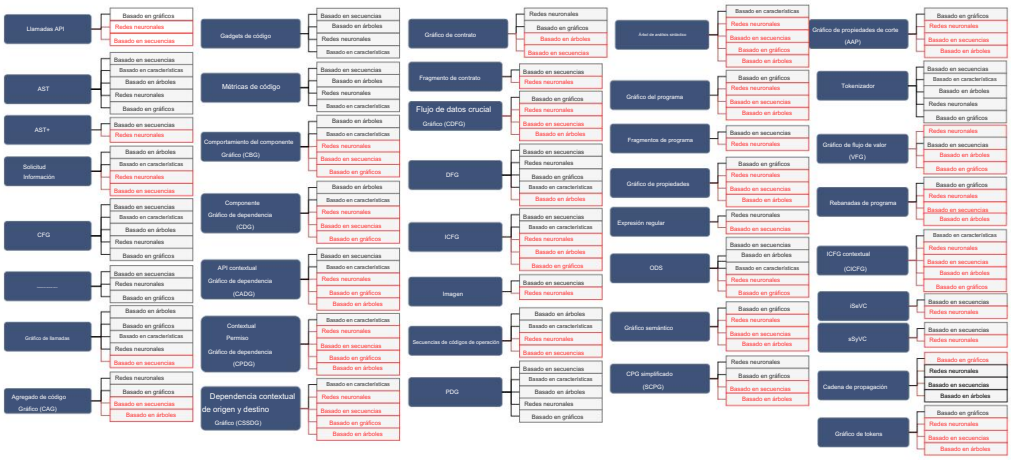

Fig. 4. Relaciones entre modelos y representaciones

Al comprender qué características hacen que un fragmento de código sea vulnerable o no, el modelo podrá detectar vulnerabilidades con éxito. Las máquinas de máquinas de seguridad (SVM) son eficaces para aprender las características que diferencian las clases, o en este caso, el código. Por lo tanto, son útiles para aprender las características del código que lo harían vulnerable, malicioso o benigno.

La Figura 4 muestra los tipos de modelos utilizados para cada tipo de representación. Además, los recuadros rojos indican los tipos de modelos que podrían ser utilizados por estas representaciones, aunque no lo observamos explícitamente en nuestros datos. Consideramos que una representación es potencialmente utilizable por un modelo si no se requieren transformaciones adicionales para obtener los datos en la forma que el modelo necesita. En otras palabras, esta categorización se basa en las representaciones que los modelos aceptan fácilmente. Muchos artículos también utilizaron múltiples modelos diferentes para comparar el rendimiento de su método. Naturalmente, existe una restricción en el tipo de representación y el tipo de modelo utilizado Por ejemplo, solo se pueden utilizar representaciones gráficas con un modelo basado en gráficos.

Resultados de RQ5:

- Los modelos basados en secuencias son la categoría de modelos más común y popular.
- Las SVM en general son el modelo más popular utilizado para diferentes tareas debido a su capacidad para discriminar características que agruparían el código en un grupo u otro.
- Los artículos estudiados se centran más en ajustar los modelos que en investigar la representación del código fuente en su conjunto.

10 amenazas a la validez En

esta sección, discutiremos las amenazas a la validez de esta encuesta en torno a las amenazas de validez de constructo, interna y externa, como lo describen Runeson y Höst [210].

La validez de constructo se refiere a la precisión con la que las medidas operativas utilizadas representan las preguntas de investigación estudiadas. En nuestro estudio, estas medidas se centran principalmente en el recuento de tareas y representaciones, así como en la relación entre ambos, y entre las representaciones y los modelos. Por lo tanto, nuestro análisis se basa en la precisión de los revisores al categorizar cada artículo. Además, al buscar los artículos, nos basamos en la capacidad de los motores de búsqueda que utilizamos para mostrar todos los artículos relacionados con nuestra consulta. Para mitigar estos problemas, solicitamos a los revisores que analizaran cada artículo por separado.

Machine Translated by Google

y posteriormente discutir y resolver cualquier discrepancia en los análisis. También creamos una consulta exhaustiva y lo suficientemente amplia como para garantizar que los resultados obtenidos de nuestra búsqueda abarcaran todos los artículos relacionados con este estudio.

La validez interna describe la eficacia con la que un estudio mitiga el sesgo y el error sistemático, de modo que se pueda extraer una conclusión causal. Debido a la posible amenaza de categorizar incorrectamente un artículo, solicitamos a dos autores que revisaran los artículos individualmente y se reunieran para discutir cualquier discrepancia entre las categorizaciones. Los desacuerdos se resolvieron mediante discusión. Para evaluar la fiabilidad de nuestra evaluación, utilizamos el índice kappa de Cohen. Nuestra puntuación calculada es de 0,97, lo que significa que obtuvimos una concordancia casi perfecta en nuestro análisis [211].

La validez externa evalúa la generalización del estudio. La principal amenaza para este trabajo es que solo nos centramos en los últimos 10 años y medio (2012-mayo de 2023), por lo que podríamos haber omitido artículos fuera de este rango. Tampoco incluimos preprints y, por lo tanto, podríamos haber omitido artículos más recientes. Además, nuestras palabras clave se basan en aprendizaje automático, por lo que es posible que se hayan omitido los artículos que no las incluyen . Finalmente, solo analizamos tres fuentes específicas: ACM, IEEE y Springer Link. Si bien puede haber habido documentos fuera de estas tres fuentes que fueran relevantes para nuestro trabajo, estaban fuera del alcance de la búsqueda para este trabajo.

11 Consideraciones finales

En esta sección, discutimos nuestros hallazgos, compartimos recomendaciones para trabajos futuros y concluimos el trabajo.

11.1 Discusión

Además de nuestras cinco preguntas de respuesta en este SLR, también identificamos hallazgos clave de nuestro análisis exhaustivo relacionado con la representación del código fuente:

- La complejidad del gráfico proporciona más información: muchos artículos encontraron que usar una representación gráfica permite capturar más información sobre el código fuente, lo que resulta en un mejor rendimiento [37, 41, 44, 78, 81, 86, 89 , 93, 98, 105, 106, 110, 113, 114, 135, 158, 161]. Capturar completamente las relaciones y dependencias semánticas y sintácticas demostró ser crucial para ciertas tareas, particularmente la detección de vulnerabilidades [41, 44, 78, 81, 86, 89, 93, 98, 105, 106, 110, 113, 114, 135, 158, 161]. Además, Sarbakysh y Wang [113] descubrieron que el uso de una representación gráfica permite revelar conexiones ocultas dentro del código fuente, lo que resultó ser útil para la detección de vulnerabilidades y abordó las limitaciones de los métodos tradicionales.

- Capturar la información estructural/sintáctica es importante: Otra tendencia común es que la estructura y la sintaxis del código fuente es un aspecto importante que afecta la capacidad de un modelo para aprender completamente el código fuente [19, 44, 78 , 85 , 86 , 93, 98, 105, 106, 114, 136, 152, 158, 161, 164, 175].
  Un estudio reveló que, en general, las representaciones gráficas ofrecen un mejor rendimiento, especialmente para la detección de vulnerabilidades, ya que la información significativa relativa a la estructura, la sintaxis y la semántica se conserva mediante la representación gráfica [98]. En general, un estudio reveló que las vulnerabilidades están relacionadas con la estructura y los patrones del código [175]. Además, otros estudios demostraron que incluir información sobre la estructura del código sobre una representación que no considera la estructura contribuía a mejorar el rendimiento general [85, 86].

- Las características extraídas impactan el rendimiento: Aunque parezca obvio, las características extraídas del código fuente impactan el rendimiento del modelo, ya que determinan lo que este puede aprender. En el caso de la detección de vulnerabilidades, las características extraídas pueden no estar asociadas con algunas vulnerabilidades, lo que las hace, a falta de un término mejor, inútiles para el modelo [15, 105, 132, 154, 164, 175]. De igual manera, ciertas representaciones

Machine Translated by Google

ofrecen más información sobre el código fuente que otros (es decir, un CPG versus un AST), lo que mejora la expresividad de la semántica, dando como resultado características extraídas más informativas, lo que lleva a que el modelo funcione mejor [31, 37, 41, 45, 110, 119, 158].

- Considerar el código como texto no captura bien los comportamientos: Muchos estudios coinciden en que tratar el código como texto plano no proporciona suficiente información para que un modelo aprenda sobre el código [43, 44, 81, 89, 105, 119, 132 , 135, 136, 158, 164]. El código tiene más estructuras semánticas que difieren del lenguaje natural [89], y representar el código como texto plano causa una pérdida en esta información semántica, como la lógica de ejecución y las relaciones entre el código, lo cual es crucial para tareas como la detección de vulnerabilidades [44, 81, 105, 132]. Un artículo encontró que usar la tokenización a nivel de subpalabra de BPE es más útil que la tokenización a nivel de palabra [119], como resultado de la tendencia del código a contener palabras clave poco comunes en lugar de lenguaje natural. Usar la tokenización a nivel de palabra, como se hace en NLP, da como resultado representaciones vectoriales que no son particularmente significativas, lo que resulta en un rendimiento deficiente [43, 119].
- Capturar el contexto mejora el rendimiento: Incluir información contextual ayuda a los modelos a distinguir mejor las características del código, lo que se traduce en un mejor rendimiento [19, 26, 100, 172]. En particular, esta información contextual puede reducir el número de falsos positivos. En el ejemplo de la detección de vulnerabilidades, añadir contexto puede ayudar a distinguir con precisión las zonas maliciosas de las benignas [19].
- En ciertos casos, los tokenizadores pueden ser útiles: si bien la mayoría de los artículos afirman que los tokenizadores no son óptimos para las tareas de ingeniería de software relacionadas con la ciberseguridad, dos artículos señalan que hay casos en los que los tokenizadores podrían ser una representación beneficiosa [15, 122]. La tokenización resulta útil para facilitar el acceso a cierta información sintáctica y semántica a arquitecturas neuronales simples, que la utilizan para el ajuste fino en tareas posteriores [122]. Además, un artículo descubrió que mejorar el diccionario incrementando su tamaño y complejidad durante la tokenización también mejoraba el rendimiento del método [15]. Como se mencionó anteriormente, se ha demostrado que la tokenización a nivel de subpalabra de BPE mejora el rendimiento [43, 119], lo que sugiere que, si bien un tokenizador estándar (p. ej., uno utilizado en un modelo de PLN) puede no ser la representación más útil, existe margen para mejorar los métodos de tokenización de modo que se transmita información más pertinente al modelo.

11.2 Recomendaciones

De los 141 artículos que hemos estudiado, observamos que, en lugar de buscar una forma nueva o más completa de representar el código fuente, los trabajos previos se centran en probar modelos diferentes o nuevos con arquitecturas más avanzadas para que el modelo pueda aprender más de la representación, sin modificarla. Dadas las mejoras en potencia y capacidad de los modelos de aprendizaje automático en los últimos años, es comprensible que los investigadores adopten este enfoque. Sin embargo, es importante recordar que la forma en que un modelo aprende se ve muy afectada por las representaciones de características, ya que estas le permiten aprender y aislar información crítica esencial para completar su tarea con éxito.

Con base en los resultados presentados en las Secciones 5 a 9, recomendamos que trabajos futuros exploren diferentes representaciones para una tarea específica, en lugar de simplemente ajustar el modelo. Dado que el modelo aprende características del código fuente a partir de la representación, se lograría una mayor mejora en el rendimiento si se prestara más atención a las representaciones. También es importante que los investigadores presten atención a la popularidad de los lenguajes, ya que se debe esforzarse por crear técnicas y herramientas que aborden los lenguajes más populares y de uso frecuente para garantizar que podamos evitar la mayor cantidad posible de riesgos de seguridad.

Al elegir una representación para una tarea específica, se pueden considerar las capacidades de las diferentes representaciones. La Tabla 5 destaca estas capacidades según sus características clave. «Ligero» se refiere a

Machine Translated by Google

Tabla 5. Habilidades de las diferentes representaciones en este estudio

	Estructura Semántica	Flujo de datos	Flujo de declaraciones	Ligero	Interprocedimental
CPG, PDG, Segmentos de programa, ODS, SPG, CAG					
VFG, Gráfico de contrato/semántico					
Gráfico de programa, Cadena de propagación CADG, Gráfico de propiedades, CPDG CICFG, Gráfico de llamadas SCPG, CSSDG AST, Árbol de análisis, Tokenizador de imágenes, Secuencias de código de operación iSeVC, Fragmento de contrato de gadgets de código, Información de la aplicación, Llamadas API, Métricas de código CFG DFG					
ICFG					
CDG					
CBG					
CDFG					
Gráfico de tokens					
sSyVC					
Expresión regular					

Las representaciones con las mismas capacidades se agrupan. BPE es tokenización de subpalabras BPE, Op. Seq. es código de operación. secuencias y la información de la aplicación es información de la aplicación.

a la complejidad de la representación. Por ejemplo, como se mencionó anteriormente, los AST son simples para generar. Debido a que su complejidad computacional es relativamente baja, pueden considerarse Ligero. Sin embargo, las porciones de programa tienen mucha sobrecarga y, por lo tanto, no se consideran ligeras.

Además, también se puede considerar el rendimiento de estas representaciones para tareas particulares en comparación entre sí. La Figura 5 muestra la comparación de la precisión promedio, memoria y puntuaciones F1 para ciertas tareas, según lo informado por los artículos estudiados. Por consideraciones de espacio, En este artículo solo demostramos la precisión, F1 y los puntajes de recuperación, y solo para tareas que tenían Representaciones múltiples. El resto de los datos métricos completos están disponibles en nuestro repositorio de GitHub. para el proyecto.

Podemos ver en la Figura 5 que para la detección de vulnerabilidades, el CPG tiene la mejor recuperación (99,0%). El programa slices tiene la mejor precisión (94,8%) y el programa graphic tiene la mejor puntuación F1 (94,5%). Curiosamente, todas estas representaciones se basan en gráficos, lo que demuestra la importancia de la información estructural a la hora de detectar vulnerabilidades. Como un CPG es un gráfico increíblemente robusto, lo que da Si se tiene conocimiento sobre muchos aspectos de un programa, se deduce que esta representación funcionaría increíblemente bien. Bueno. Una cosa a tener en cuenta es que, debido a que es una representación tan robusta, es complejo generar, lo que puede no ser ideal para alguien que no tiene los recursos necesarios para generar un gráfico de este tipo. En términos generales, las representaciones gráficas muestran resultados sólidos en una variedad de tareas, lo que habla de su capacidad para transmitir la información crucial relacionada con el código fuente. Sin embargo, en la predicción y análisis de vulnerabilidades, una representación tokenizada tuvo el mejor rendimiento. Rendimiento en comparación con otros métodos (80,1 % de recuperación y 96,8 %, respectivamente). Métricas de código También mostró resultados sólidos para la detección/identificación de parches de seguridad (puntuación F1 del 91,4 %), así como detección de confirmaciones de inducción/corrección de vulnerabilidades (92,0 % de precisión) y análisis de vulnerabilidades (96,8 % recordar). Las métricas de código son una representación muy ligera y su rendimiento sugiere que la Las características que presenta (por ejemplo, líneas de código o rotación de código) proporcionan suficiente información sobre la fuente. código para desempeñarse bien en tareas relacionadas con cambios de código relacionados con la seguridad y análisis de vulnerabilidades. Siow et al. [208] encontraron resultados similares a los nuestros en su estudio empírico sobre representaciones de código para código tareas de inteligencia como la clasificación de códigos. El artículo encontró que las representaciones basadas en grafos son El mejor en la representación de la semántica del programa, lo que resulta en el mejor rendimiento en comparación con

Machine Translated by Google

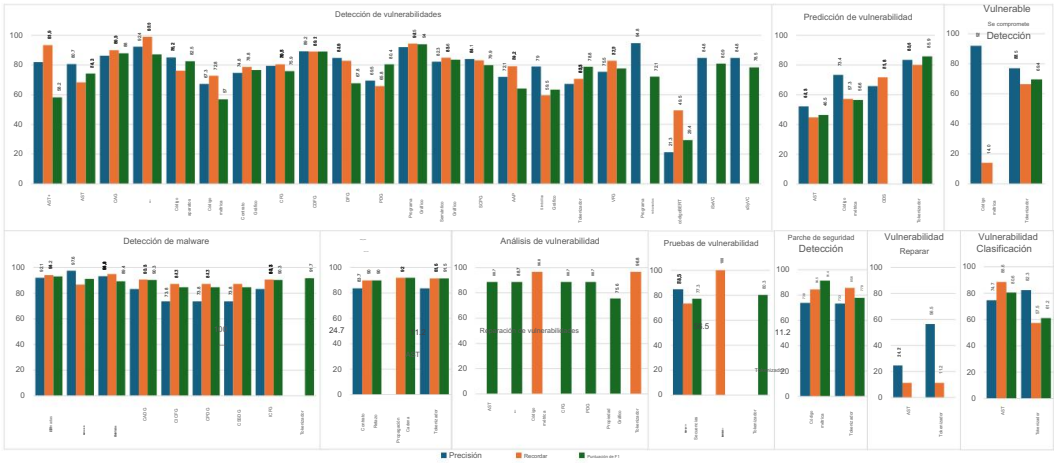

Fig. 5. Métricas agregadas para cada representación por tarea. Las abreviaturas son las siguientes: App Info = Información de la Aplicación; CADG = Gráfico de Dependencia de API Contextual; CICFG = ICFG Contextual; CPDG = Gráfico de Dependencia de Permisos Contextual; CSSDG = Gráfico de Dependencia de Origen y Destino Contextual; ICFG = Gráfico de Flujo de Control Interprocedimental; AST = Árbol de Sintaxis Abstracta; CPG = Gráfico de Propiedades de Código; CFG = Gráfico de Flujo de Control; PDG = Gráfico de Dependencia de Programa; CAG = Gráfico de Agregación de Código; CDFG = Gráfico de Flujo de Datos Cruciales; DFG = Gráfico de Flujo de Datos; SCPG = Gráfico de Propiedades de Código Simplificado; SPG = Gráfico de Propiedades de Segmento; VFG = Gráfico de Flujo de Valores; SDG = Gráfico de Dependencia del Sistema

Representaciones basadas en árboles, características y secuencias. Además, el artículo descubrió que diferentes tareas requieren una semántica específica para lograr el mejor rendimiento, pero en general, un grafo compuesto que represente la semántica del programa de forma exhaustiva producirá resultados sólidos.

11.3 Conclusión

En este artículo, analizamos 141 artículos para examinar el estado actual de las representaciones de código fuente en modelos de aprendizaje automático (ML) utilizados en tareas relacionadas con la ciberseguridad. Descubrimos que los AST y las representaciones tokenizadas son las formas más comunes de representar el código fuente. También descubrimos que la detección de vulnerabilidades, la detección de malware y la predicción de vulnerabilidades son las tareas más cubiertas por las técnicas existent Además, observamos que C fue el lenguaje cubierto por más técnicas, seguido de C++.

Referencias

[1] JCS Santos, K. Tarrit y M. Mirakhorli. 2017. Un catálogo de debilidades de la arquitectura de seguridad. En Actas de la Talleres de la Conferencia Internacional IEEE sobre Arquitectura de Software 2017 (ICSAW'17).

[2] I. Alexander. 2003. Casos de uso indebido: Casos de uso con intención hostil. IEEE Software 20, 1 (2003), 58–66. DOI: http://doi.org/10.1109/MS.2003.1159030

[3] J. McDermott y C. Fox. 1999. Uso de modelos de casos de abuso para el análisis de requisitos de seguridad. En Actas de la 15.ª Conferencia Anual de Aplicaciones de Seguridad Informática (ACSAC'99). DOI: http://doi.org/10.1109/CSAC.1999.816013

[4] A. Shostack. 2014. Modelado de amenazas: diseño para la seguridad. Wiley.

[5] ST Halkidis, N. Tsantalis, A. Chatzigeorgiou y G. Stephanides. 2008. Análisis de riesgos arquitectónicos de sistemas de software basado en patrones de seguridad. IEEE Transactions on Dependable and Secure Computing 5, 3 (2008), 129–142. DOI: http://doi.org/10.1109/TDSC.2007.70240

[6] J. Ryoo, P. Laplante y R. Kazman. 2010. Una metodología para extraer tácticas de seguridad de patrones de seguridad. En Actas de la 43.ª Conferencia Internacional de Hawái sobre Ciencias de Sistemas de 2010. DOI: http://doi.org/10.1109/HICSS.2010.18

[7] L. Braz, E. Fregnan, G. Çalikli y A. Bacchelli. 2021. ¿Por qué los desarrolladores no detectan una validación de entrada incorrecta?; Artículos sobre DROP TABLE. En Actas de la 43.ª Conferencia Internacional de Ingeniería de Software IEEE/ACM de 2021 (ICSE'21).

Machine Translated by Google

[8] B. Arkin, S. Stender y G. McGraw. 2005. Pruebas de penetración de software. IEEE Security & Privacy 3, 1 (2005), 84–87. DOI: http://doi.org/10.1109/MSP.2005.23

[9] B. Chess y G. McGraw. 2004. Análisis estático para seguridad. IEEE Security & Privacy 2, 6 (2004), 76–79. DOI: http://doi.org/10.1109/MSP.2004.111

[10] A. Russo y A. Sabelfeld. 2010. Análisis de seguridad dinámico vs. estático sensible al flujo. En Actas de la 23.ª edición de 2010. Simposio sobre Fundamentos de Seguridad Informática del IEEE. DOI: http://doi.org/10.1109/CSF.2010.20

[11] G. Lin, W. Xiao, J. Zhang y Y. Xiang. 2020. Detección de funciones vulnerables basada en aprendizaje profundo: Un punto de referencia. En Seguridad de la Información y las Comunicaciones. Apuntes de Informática, vol. 11999. Springer, 219–232.

[12] G. Renjith y S. Aji. 2021. Análisis y detección de vulnerabilidades mediante redes neuronales de grafos para el sistema operativo Android. En Seguridad de Sistemas de Información. Apuntes de Informática, vol. 13146. Springer, pp. 57–72.

[13] N. Guo, X. Li, H. Yin e Y. Gao. 2020. VulHunter: Un sistema automatizado de detección de vulnerabilidades basado en aprendizaje profundo y código de bytes. En Seguridad de la Información y las Comunicaciones. Apuntes de Informática, vol. 11999. Springer, 199–218.

[14] J. Chen, B. Liu, S. Cai, W. Wang y S. Wang. 2021. AIdetectorX: Un detector de vulnerabilidades basado en TCN y mecanismo de autoatención. En Ingeniería de Software Confiable: Teorías, Herramientas y Aplicaciones. Apuntes de Clase en Ciencias de la Computación , vol. 13071. Springer, 161–177.

[15] B. Mosolygó, N. Vándor, P. Hegedűs y R. Ferenc. 2022. Un enfoque de detección de vulnerabilidades explicable a nivel de línea para Java. En Ciencias de la Computación y sus Aplicaciones — Talleres ICCSA 2022. Apuntes de Clase en Ciencias de la Computación, vol. 13380. Springer, pp. 106–122.

[16] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel y P. Montague. 2018. Aprendizaje de representación de transferencia entre proyectos para el descubrimiento de funciones vulnerables. IEEE Transactions on Industrial Informatics 14, 7 (2018), 3289–3297. DOI: http://doi.org/10.1109/TII.2018.2821768

[17] L. Cen, CS Gates, L. Si y N. Li. 2015. Un modelo discriminativo probabilístico para la detección de malware en Android con código fuente descompilado. IEEE Transactions on Dependable and Secure Computing 12, 4 (2015), 400–412. DOI: http://doi.org/10.1109/TDSC.2014.2355839

[18] A. Narayanan, L. Yang, L. Chen y L. Jinliang. 2016. Detección adaptativa y escalable de malware para Android mediante aprendizaje en línea. En las Actas de la Conferencia Conjunta Internacional sobre Redes Neuronales de 2016 (IJCNN'16). DOI: http://doi.org/10.1109/IJCNN.2016.7727508

[19] A. Narayanan, M. Chandramohan, L. Chen y Y. Liu. 2017. Malware para Android sensible al contexto, adaptable y escalable. Detección mediante aprendizaje en línea (versión extendida). arXiv:cs.CR/1706.00947.

[20] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross y G. Stringhini. 2017. MaMaDroid: Detección Malware para Android mediante la construcción de cadenas de Markov de modelos de comportamiento. arXiv:cs.CR/1612.04433.

[21] P. Zegzhda, D. Zegzhda, E. Pavlenko y G. Ignatev. 2018. Aplicación de técnicas de aprendizaje profundo para la detección de malware en Android. En las Actas de la 11.ª Conferencia Internacional sobre Seguridad de la Información y las Redes (SIN'18). Artículo 7. DOI: http://doi.org/10.1145/3264437.3264476

[22] J. Allen, M. Landen, S. Chaba, Y. Ji, SPH Chung y W. Lee. 2018. Mejora de la precisión en la detección de malware en Android con conciencia contextual ligera. En las Actas de la 34.ª Conferencia Anual de Aplicaciones de Seguridad Informática (ACSAC'18). DOI: http://doi.org/10.1145/3274694.3274744 [23] JD Koli. 2018. RanDroid: Detección de malware en Android mediante clasificadores aleatorios de aprendizaje automático. En Actas del Congreso de Tecnologías para la Seguridad Energética y la Energía en Ciudades Inteligentes (ICSESP'18) de 2018. DOI: http://doi.org/10.1109/ICSESP.2018.8376705

[24] Z. Wang, J. Cai, S. Cheng y W. Li. 2016. DroidDeepLearner: Identificación de malware para Android mediante aprendizaje profundo. En las Actas del 37.º Simposio Sarnoff del IEEE de 2016. DOI: http://doi.org/10.1109/SARNOF.2016.7846747 [25] N. Xie, F. Zeng, X. Qin, Y. Zhang, M. Zhou y C. Lv. 2018. RepassDroid: Detección automática de malware para Android basada en permisos esenciales y características semánticas de API sensibles. En las Actas del Simposio Internacional sobre Aspectos Teóricos de la Ingeniería de Software (TASE'18) de 2018. DOI: http://doi.org/10.1109/TASE.2018.00015 [26] A. Narayanan, M. Chandramohan, L. Chen e Y. Liu. 2017. Un enfoque multivista contextual para la detección de malware y la localización de código malicioso en Android. Ingeniería de Software Empírica 23 (2017), 1222–1274. DOI: http://doi.org/10.1007/s10664-017-9539-8

[27] C. Yang, Z. Xu, G. Gu, V. Yegneswaran y P. Porras. 2014. DroidMiner: Minería automatizada y caracterización de comportamientos maliciosos de grano fino en aplicaciones Android. En Seguridad Informática — ESORICS 2014. Apuntes de Clases en Ciencias de la Computación, vol. 8712. Springer, 163–182.

[28] L. Braz, C. Aeberhard, G. Çalikli y A. Bacchelli. 2022. Menos es más: Apoyo a los desarrolladores en la detección de vulnerabilidades durante la revisión de código. En Actas de la 44.ª Conferencia Internacional de Ingeniería de Software (ICSE'22). DOI: http://doi.org/10.1145/3510003.3511560

[29] D. Vagavolu, KC Swarna y S. Chimalakonda. 2021. Un mocktail de representaciones de código fuente. En Actas de la 36.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASE'21) de 2021. DOI: http://doi.org/10.1109/ASE51524.2021.9678551

Machine Translated by Google

[30] J. Chi, Y. Qu, T. Liu, Q. Zheng y H. Yin. 2023. SeqTrans: Corrección automática de vulnerabilidades mediante aprendizaje secuencial . IEEE Transactions on Software Engineering 49 (2023), 564–585. DOI: http://doi.org/10.1109/TSE.2022.3156637 [31] F. Al Debeyan, T. Hall y D. Bowes. 2022. Mejora del rendimiento de la predicción de vulnerabilidades de código mediante información de árboles de sintaxis abstracta. En las Actas de la 18.ª Conferencia Internacional sobre Modelos Predictivos y Análisis de Datos en Ingeniería de Software (PROMISE'22). DOI: http://doi.org/10.1145/3558489.3559066

[32] A. Sejfia y M. Schäfer. 2022. Detección automatizada práctica de paquetes npm maliciosos. En Actas del 44.º Conferencia Internacional sobre Ingeniería de Software. DOI: http://doi.org/10.1145/3510003.3510104

[33] GE de P. Rodrigues, AM Braga y R. Dahab. 2020. Uso de incrustaciones de grafos y aprendizaje automático para detectar el uso indebido de criptografía en el código fuente. En Actas de la 19.ª Conferencia Internacional del IEEE sobre Aprendizaje Automático y Aplicaciones (ICMLA'20) de 2020. DOI: http://doi.org/10.1109/ICMLA51294.2020.00171

[34] G. Partenza, T. Amburgey, L. Deng, J. Dehlinger y S. Chakraborty. 2021. Identificación automática de código vulnerable: Investigaciones con una red neuronal basada en AST. En las Actas de la 45.ª Conferencia Anual de Computadoras, Software y Aplicaciones del IEEE de 2021 (COMPSAC'21). DOI: http://doi.org/10.1109/COMPSAC51774.2021.00219 [35] A. Mester y Z. Bodó. 2022. Clasificación de malware basada en redes neuronales convolucionales de grafos y características de grafos de llamadas estáticas. En Avances y Tendencias en Inteligencia Artificial. Teoría y Prácticas en Inteligencia Artificial. Apuntes de Clase en Ciencias de la Computación, vol. 13343. Springer, 528–539.

[36] JA Harer, LY Kim, RL Russell, O. Ozdemir, LR Kosta, A. Rangamani, LH Hamilton, GI Centeno, JR Key, P. M. Ellingwood, et al. 2018. Detección automatizada de vulnerabilidades de software con aprendizaje automático. arXiv:cs.SE/1803.04497.

[37] S. Cao, X. Sun, L. Bo, Y. Wei y B. Li. 2021. BGNN4VD: Construcción de una red neuronal de grafos bidireccional para la detección de vulnerabilidades. Tecnología de la Información y el Software 136 (2021), 106576. DOI: http://doi.org/10.1016/j.infsof.2021.106576

[38] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou y A. Ampatzoglou. 2022. Una evaluación empírica de la utilidad de las técnicas de incrustación de palabras en la predicción de vulnerabilidades basada en aprendizaje profundo. En Seguridad en Ciencias de la Computación e Información. Comunicaciones en Ciencias de la Computación e Información, vol. 1596. Springer, pp. 23-37.

[39] R. Feng, Z. Yan, S. Peng y Y. Zhang. 2022. Detección automatizada de fugas de contraseñas de repositorios públicos de GitHub. En las Actas de la 44.ª Conferencia Internacional sobre Ingeniería de Software IEEE/ACM de 2022 (ICSE'22). DOI: http:// //doi.org/10.1145/3510003.3510150 [40] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty y Y. Xiong. 2019. Proyecto Achilles: Una herramienta prototipo para la detección de vulnerabilidades a nivel de método estático en código fuente Java mediante una red neuronal recurrente. En las Actas del 34.º Taller de la Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASEW'19) de 2019. DOI: http:// //doi.org/10.1109/ASEW.2019.00040 [41] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng e Y. Zhong. 2018. VulDeePecker: Un sistema basado en aprendizaje profundo para la detección de vulnerabilidades. En las Actas del Simposio de Seguridad de Redes y Sistemas Distribuidos de 2018. DOI: http://doi.org/10.14722/ndss.2018.23158

[42] V.-A. Nguyen, D.Q. Nguyen, V. Nguyen, T. Le, Q.H. Tran y D. Phung. 2022. ReGVD: Revisión de las redes neuronales de grafos para la detección de vulnerabilidades. En Actas de la 44.ª Conferencia Internacional de Ingeniería de Software ACM/IEEE: Actas complementarias (ICSE'22). DOI: http://doi.org/10.1145/3510454.3516865

[43] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen y D. Phung. 2022. VulRepair: Reparación automatizada de vulnerabilidades de software basada en T5. En las Actas de la 30.ª Conferencia y Simposio Conjunto Europeo de Ingeniería de Software de la ACM sobre los Fundamentos de la Ingeniería de Software (ESEC/FSE'22). DOI: http://doi.org/10.1145/3540250.3549098 [44] X. Cheng, G. Zhang, H. Wang e Y. Sui. 2022. Incrustación de código sensible a rutas mediante aprendizaje contrastivo para la detección de vulnerabilidades de software. En Actas del 31.º Simposio Internacional ACM SIGSOFT sobre Pruebas y Análisis de Software (ISSTA'22). DOI: http://doi.org/10.1145/3533767.3534371

[45] SM Ghaffarian y HR Shahriari. 2021. Análisis de vulnerabilidades de software neuronal mediante representaciones de programas en grafos intermedios enriquecidos. Ciencias de la Información 553 (2021), 189–207. DOI: http://doi.org/10.1016/j.ins.2020.11.053 [46] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang y Q. He. 2020. Detección de vulnerabilidades en contratos inteligentes mediante redes neuronales de grafos. En las Actas de la 29.ª Conferencia Internacional Conjunta sobre Inteligencia Artificial (IJCAI'20). DOI: http://doi.org/10.24963/ijcai.2020/454

[47] P. Qian, Z. Liu, Q. He, R. Zimmermann y X. Wang. 2020. Hacia la detección automatizada de reentrada para contratos inteligentes basados en modelos secuenciales. IEEE Access 8 (2020), 19685–19695. DOI: http://doi.org/10.1109/ACCESS.2020.2969429 [48] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu y H. Jin. 2022. VulDeeLocator: Un detector de vulnerabilidades de grano fino basado en aprendizaje profundo . IEEE Transactions on Dependable and Secure Computing 19 (2022), 2821–2837. DOI: http://doi.org/10.1109/ TDSC.2021.3076142

[49] B. Kitchenham y S. Charters. 2007. Directrices para realizar revisiones sistemáticas de literatura en ingeniería de software. Informe técnico. EBSE.

[50] RA Kemmerer. 2003. Ciberseguridad. En Actas de la 25ª Conferencia Internacional sobre Ingeniería de Software de 2003. DOI: http://doi.org/10.1109/ICSE.2003.1201257

Machine Translated by Google

[51] X. Xia, Y. Wang y Y. Yang. 2021. Detección de vulnerabilidades de código fuente basada en SAR-GIN. En las Actas de la 2.ª Conferencia Internacional sobre Electrónica, Comunicaciones y Tecnologías de la Información (CECIT'21) de 2021. DOI: http://doi.org/10.1109/CECIT53797.2021.00202

[52] F. Yamaguchi, N. Golde, D. Arp y K. Rieck. 2014. Modelado y descubrimiento de vulnerabilidades con grafos de propiedades de código. En las Actas del Simposio IEEE sobre Seguridad y Privacidad de 2014. DOI: http://doi.org/10.1109/SP.2014.44 [53] R. Scandariato, J. Walden, A. Hovsepyan y W. Joosen. 2014. Predicción de componentes de software vulnerables mediante minería de texto. IEEE Transactions on Software Engineering 40, 10 (2014), 993–1006. DOI: http://doi.org/10.1109/TSE.2014. 2340398

[54] L. Buratti, S. Pujar, M. Bornea, S. McCarley, Y. Zheng, G. Rosselló, A. Morari, J. Laredo, V. Thost, Y. Zhuang, et al. 2020. Explorando la naturalidad del software a través de modelos de lenguaje neuronal. arXiv:cs.CL/2006.12641.

[55] A. Bernstein y A. Kuleshov. 2014. Representación de datos de baja dimensión en el análisis de datos. En Redes Neuronales Artificiales en el Reconocimiento de Patrones. Apuntes de Clase en Ciencias de la Computación, vol. 8774. Springer, pp. 47–58.

[56] M. Grohe. 2020. Word2vec, Node2vec, Graph2vec, X2vec: Hacia una teoría de incrustaciones vectoriales de datos estructurados. En las Actas del 39.º Simposio ACM SIGMOD-SIGACT-SIGAI sobre Principios de Sistemas de Bases de Datos (PODS'20). DOI: http://doi.org/10.1145/3375395.3387641

[57] T. Mikolov, K. Chen, G. Corrado y J. Dean. 2013. Estimación eficiente de representaciones de palabras en el espacio vectorial. arXiv:cs.CL/1301.3781.

[58] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu y S. Jaiswal. 2017. graph2vec: Aprendizaje de representaciones distribuidas de gráficos. arXiv:1707.05005 (cs). DOI: http://doi.org/10.48550/ARXIV.1707.05005 [59] U. Alon, M. Zilberstein, O. Levy y E. Yahav. 2019. Code2vec: Aprendizaje de representaciones distribuidas de código. Actas de la ACM sobre Lenguajes de Programación 3, POPL (2019), Artículo 40, 29 páginas. DOI: http://doi.org/10.1145/3290353 [60] W. Ma, M. Zhao, E. Soremekun, Q. Hu, J. M. Zhang, M. Papadakis, M. Cordy, X. Xie y YL Traon. 2022. Graph- Code2Vec: Incrustación de código genérico mediante análisis léxicos y de dependencia de programas. En las Actas de la 19.ª Conferencia Internacional sobre Minería de Repositorios de Software. DOI: http://doi.org/10.1145/3524842.3528456

[61] S. Khan y S. Parkinson. 2018. Análisis del estado del arte en la evaluación de vulnerabilidades mediante inteligencia artificial. En la Guía para el Análisis de Vulnerabilidades para Redes y Sistemas Informáticos. Springer, pp. 3-32. DOI: http://doi.org/10.1007/978-3- 319-92624-7_1

[62] VHS Durelli, RS Durelli, SS Borges, AT Endo, MM Eler, DRC Dias y MP Guimarães. 2019. Aprendizaje automático aplicado a las pruebas de software: Un estudio de mapeo sistemático. IEEE Transactions on Reliability 68, 3 (2019), 1189–1212. DOI: http://doi.org/10.1109/TR.2019.2892517 [63] MT Bin Nazim, MJH Faruk, H. Shahriar, MA Khan, M. Masum, N. Sakib y F. Wu. 2022. Análisis sistemático de un modelo de aprendizaje profundo para la detección de código vulnerable. En las Actas de la 46.ª Conferencia Anual de Computadoras, Software y Aplicaciones del IEEE de 2022 (COMPSAC'22). DOI: http://doi.org/10.1109/COMPSAC54236.2022.00281 [64] HP Samoaa, F. Bayram, P. Salza y P. Leitner. 2022. Un estudio sistemático de mapeo de la representación del código fuente para el aprendizaje profundo en ingeniería de software. IET Software 16, 4 (2022), 351–385. DOI: http://doi.org/10.1049/sfw2.12064 [65] Y. Yang, X. Xia, D. Lo y J. Grundy. 2020. Una encuesta sobre aprendizaje profundo para ingeniería de software. arXiv:cs.SE/2011.14597.

[66] F. Ferreira, LL Silva y MT Valente. 2020. La ingeniería de software se encuentra con el aprendizaje profundo: un estudio de mapeo. arXiv:cs.SE/1909.11436.

[67] G. Lin, S. Wen, Q.-L. Han, J. Zhang y Y. Xiang. 2020. Detección de vulnerabilidades de software mediante redes neuronales profundas: Un estudio. Actas del IEEE 108, 10 (2020), 1825–1848. DOI: http://doi.org/10.1109/JPROC.2020.2993293 [68] T. Sonnekalb, T. S Heinze y P. Mäder. 2022. Análisis de seguridad profundo del código del programa: una revisión sistemática de la literatura. Ingeniería de software empírica 27 (2022), 1–39. https://link.springer.com/content/pdf/10.1007/s10664-021-10029-x.pdf [69] AOA Semasaba, W. Zheng, X. Wu y SA Agyemang. 2020. Estudio bibliográfico sobre análisis de vulnerabilidades basado en aprendizaje profundo en código fuente. IET Software 14, 6 (2020), 654–664. DOI: http://doi.org/10.1049/iet-sen.2020.0084 [70] SM Ghaffarian y HR Shahriari. 2018. Análisis y descubrimiento de vulnerabilidades de software mediante técnicas de aprendizaje automático y minería de datos: Una encuesta. ACM Computing Surveys 50, 4 (2018), Artículo 56, 36 páginas. DOI: http://doi.org/10.1145/3092566

[71] J. Wu. 2021. Revisión de la literatura sobre detección de vulnerabilidades utilizando tecnología NLP. arXiv:cs.CR/2104.11230.

[72] Z. Chen y M. Monperrus. 2019. Un estudio de la literatura sobre incrustaciones en código fuente. arXiv:cs.LG/1904.03061.

[73] Z. Kotti, R. Galanopoulou y D. Spinellis. 2023. Aprendizaje automático para ingeniería de software: un estudio terciario. ACM Computing Surveys 55, 12 (2023), Artículo 256, 39 páginas. DOI: http://doi.org/10.1145/3572905

[74] LE Lwakatare, A. Raj, J. Bosch, HH Olsson e I. Crnkovic. 2019. Una taxonomía de los desafíos de la ingeniería de software para sistemas de aprendizaje automático: Una investigación empírica. En Procesos ágiles en ingeniería de software y programación extrema. Apuntes de clase en Business Information Processing, vol. 355. Springer, 227–243.

[75] H. Hanif, MH Nizam Md. Nasir, M. Faizal Ab Razak, A. Firdaus y NB Anuar. 2021. El auge de la vulnerabilidad del software: Taxonomía de la detección de vulnerabilidades de software y enfoques de aprendizaje automático. Journal of Network and Computer Applications 179 (2021), 103009. DOI: http://doi.org/10.1016/j.jnca.2021.103009

Machine Translated by Google

[76] M. Usman, MA Jan, X. He y J. Chen. 2019. Una encuesta sobre los esfuerzos de aprendizaje de representación en el dominio de la ciberseguridad. ACM Computing Surveys 52, 6 (2019), Artículo 111, 28 páginas. DOI: http://doi.org/10.1145/3331174

[77] M. Macas, C. Wu y W. Fuertes. 2022. Una encuesta sobre aprendizaje profundo para ciberseguridad: Avances, desafíos y oportunidades. Computer Networks 212 (2022), 109032. DOI: http://doi.org/10.1016/j.comnet.2022.109032 [78] Y. Zhou, S. Liu, J. Siow, X. Du y Y. Liu.

2019. Devign: Identificación eficaz de vulnerabilidades mediante el aprendizaje de la semántica integral de programas mediante redes neuronales de grafos. En las Actas de la 33.ª Conferencia Internacional sobre Sistemas de Procesamiento de Información Neural. https://proceedings.neurips.cc/paper_files/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f- Paper.pdf

[79] HH Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan y L. Jiang. 2022. MANDO: incorporaciones de gráficos heterogéneos de varios niveles para la detección detallada de vulnerabilidades de contratos inteligentes. arXiv:cs.SE/2208.13252.

[80] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari y J. Laredo. 2021. Detección de vulnerabilidad de software mediante aprendizaje profundo sobre la representación de gráficos de código desagregado. arXiv:cs.AI/2109.03341.

[81] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi e Y. Sui. 2019. Detección estática de vulnerabilidades relacionadas con el flujo de control mediante incrustación de grafos. En las Actas de la 24.ª Conferencia Internacional sobre Ingeniería de Sistemas Informáticos Complejos (ICECCS'19) de 2019. DOI: http://doi.org/10.1109/ICECCS.2019.00012

[82] G. Grieco, GL Grinblat, L. Uzal, S. Rawat, J. Feist y L. Mounier. 2016. Hacia el descubrimiento de vulnerabilidades a gran escala mediante aprendizaje automático. En las Actas de la 6.ª Conferencia de la ACM sobre Seguridad y Privacidad de Datos y Aplicaciones (CODASPY'16). DOI: http://doi.org/10.1145/2857705.2857720

[83] J. Kronjee, A. Hommersom y H. Vranken. 2018. Descubrimiento de vulnerabilidades de software mediante análisis de flujo de datos y aprendizaje automático. En Actas de la 13.ª Conferencia Internacional sobre Disponibilidad, Confiabilidad y Seguridad (ARES'18). Artículo 6. DOI: http://doi.org/10.1145/3230833.3230856 [84] K.

Cheng, G. Du, T. Wu, L. Chen y G. Shi. 2022. Mutación automatizada de códigos vulnerables mediante aprendizaje profundo para la detección de variabilidad. En las Actas de la Conferencia Conjunta Internacional sobre Redes Neuronales de 2022 (IJCNN'22). DOI: http://doi.org/10.1109/IJCNN55064.2022.9892444 [85] T. Wu, L. Chen, G. Du, C.

Zhu, N. Cui y G. Shi. 2022. Detección inductiva de vulnerabilidades mediante redes neuronales de grafos cerrados. En las Actas de la 25.ª Conferencia Internacional del IEEE sobre Trabajo Cooperativo en Diseño con Soporte Informático (CSCWD'22) de 2022. DOI: http://doi.org/10.1109/CSCWD54268.2022.9776051 [86] R. Rabheru, H. Hanif y S. Maffeis. 2022. Un enfoque de red

neuronal gráfica híbrida para detectar vulnerabilidades de PHP. En las Actas de la Conferencia IEEE de 2022 sobre Computación Confiable y Segura (DSC'22). DOI: http://doi.org/10.1109/ DSC54232.2022.9888816

[87] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu y H. Jin. 2022. VulCNN: Un sistema escalable de detección de vulnerabilidades basado en imágenes. En Actas de la 44.ª Conferencia Internacional de Ingeniería de Software (ICSE'22). DOI: http://doi.org/10. 1145/3510003.3510229

[88] Y. Li, S. Wang y TN Nguyen. 2021. Detección de vulnerabilidades con interpretaciones detalladas. En Actas de la 29.ª Reunión Conjunta de la ACM sobre la Conferencia Europea de Ingeniería de Software y el Simposio sobre los Fundamentos de la Ingeniería de Software (ESEC/FSE'21). DOI: http://doi.org/10.1145/3468264.3468597 [89] W. An, L. Chen, J. Wang, G. Du, G. Shi y D.

Meng. 2020. AVDHRAM: Detección automatizada de vulnerabilidades basada en representación jerárquica y mecanismo de atención. En Actas de la Conferencia Internacional IEEE de 2020 sobre Procesamiento Paralelo y Distribuido con Aplicaciones, Big Data y Computación en la Nube, Computación Sostenible y Comunicaciones, y Computación Social y Redes (ISPA/BDCloud/SocialCom/SustainCom'20). DOI: http:// //doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00068 [90] T. Wu, L. Chen, G. Du, C. Zhu y G. Shi. 2021. Detección automatizada de vulnerabilidades basada en autoatención con representación eficaz de datos. En Actas de la

Conferencia Internacional IEEE de 2021 sobre Procesamiento Paralelo y Distribuido con Aplicaciones, Big Data y Computación en la Nube, Computación y Comunicaciones Sostenibles, y Computación Social y Redes (ISPA/BDCloud/SocialCom/SustainCom'21). DOI: http://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00126

[91] J. Zeng, X. Nie, L. Chen, J. Li, G. Du y G. Shi. 2020. Una extrapolación eficiente de vulnerabilidades mediante la similitud del núcleo del grafo de PDG. En las Actas de la 19.ª Conferencia Internacional del IEEE sobre Confianza, Seguridad y Privacidad en Informática y Comunicaciones (TrustCom'20) de 2020. DOI: http://doi.org/10.1109/TrustCom50675.2020.00229 [92] A. Watson, E. Ufuktepe y K. Palaniappan.

2022. Detección de vulnerabilidades de código de software mediante redes neuronales convolucionales 2D con mapas de características de segmentación de programas. En las Actas del Taller de Reconocimiento de Patrones de Imágenes Aplicadas del IEEE de 2022 (AIPR'22). DOI: http://doi.org/10.1109/AIPR57179.2022.10092211 [93] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao

y H. Jin. 2022. mVulPreter: Un sistema de detección de vulnerabilidades multigranular con interpretaciones. Transacciones IEEE sobre Computación Confiable y Segura. Acceso anticipado. DOI: http://doi.org/10.1109/TDSC.2022.3199769 [94]

LK Shar, H. Beng Kuan Tan y LC Briand. 2013. Minería de vulnerabilidades de inyección SQL y scripts entre sitios mediante análisis de programas híbridos. En Actas de la 35.ª Conferencia Internacional de Ingeniería de Software (ICSE'13) de 2013. DOI: http://doi.org/10.1109/ICSE.2013.6606610

Machine Translated by Google

[95] G. Yan, S. Chen, Y. Bail y X. Li. 2022. ¿Pueden los modelos de aprendizaje profundo aprender los patrones vulnerables para la detección de vulnerabilidades? En Actas de la 46.ª Conferencia Anual de Computadoras, Software y Aplicaciones del IEEE de 2022 (COMPSAC'22). DOI: http://doi.org/10.1109/COMPSAC54236.2022.00142 [96] S.
Rasthofer, S. Arzt y E. Bodden. 2014. Un enfoque de aprendizaje automático para clasificar y categorizar fuentes y receptores de Android. En las Actas del Simposio de Seguridad de Redes y Sistemas Distribuidos (NDSS'14). DOI: http:// //doi.org/10.14722/ndss.2014.23039 [97] L. Zhou, M. Huang, Y. Li, Y. Nie, J. Li y Y. Liu.
2021. GraphEye: Una solución novedosa para detectar funciones vulnerables basada en la red de atención de grafos. En las Actas de la 6.ª Conferencia Internacional IEEE sobre Ciencia de Datos en el Ciberespacio (DSC'21) de 2021. DOI: http://doi.org/10.1109/DSC53577.2021.00060

[98] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo y A. Morari. 2020. Aprendiendo a mapear el código fuente a la vulnerabilidad del software. utilizando código como gráfico. arXiv:cs.SE/2006.08614.
[99] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser y B. Ray. 2022. VELVET: Un nuevo enfoque de aprendizaje conjunto para localizar automáticamente declaraciones vulnerables. En las Actas de la Conferencia Internacional IEEE de 2022 sobre Análisis, Evolución y Reingeniería de Software (SANER'22). DOI: http://doi.org/10.1109/SANER53432.2022.00114 [100] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo,
M. Yang e Y. Wu. 2019. VulSniper: Concentra tu atención en detectar vulnerabilidades de granularidad fina. En las Actas de la 28.ª Conferencia Internacional Conjunta sobre Inteligencia Artificial (IJCAI'19). DOI: http://doi.org/10.24963/ijcai.2019/648

[101] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang y X. Mao. 2021. Peculiar: Detección de vulnerabilidades en contratos inteligentes basada en gráficos de flujo de datos cruciales y técnicas de preentrenamiento. En las Actas del 32.º Simposio Internacional sobre Ingeniería de Confiabilidad del Software del IEEE de 2021 (ISSRE'21). DOI: http://doi.org/10.1109/ISSRE52982.2021.00047 [102] H. Wang, G.
Ye, Z. Tang, SH Tan, S. Huang, D. Fang, Y. Feng, L. Bian y Z. Wang. 2021. Combinación del aprendizaje basado en grafos con la recopilación automatizada de datos para la detección de vulnerabilidades de código. IEEE Transactions on Information Forensics and Security 16 (2021), 1943-1958. DOI: http://doi.org/10.1109/TIFS.2020.3044773
[103] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu y X. Wang. 2023. Combinación de redes neuronales de grafos con conocimiento experto para la detección de vulnerabilidades en contratos inteligentes. IEEE Transactions on Knowledge and Data Engineering 35, 2 (2023), 1296–1310. DOI: http://doi.org/10.1109/TKDE.2021.3095196
[104] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang y X. Mao. 2023. Detección y localización de vulnerabilidades de reentrada: Un enfoque de dos fases basado en aprendizaje profundo. En las Actas de la 37.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASE'22). Artículo 83. DOI: http://doi.org/10.1145/3551349.3560428 [105] X. Cheng, H.
Wang, J. Hua, G. Xu e Y. Sui. 2021. DeepWukong: Detección estadística de vulnerabilidades de software mediante redes neuronales de grafos profundos. ACM Transactions on Software Engineering and Methodology 30, 3 (2021), Artículo 38, 33 páginas. DOI: http://doi.org/10.1145/3436877
[106] J. Gear, Y. Xu, E. Foo, P. Gauravaram, Z. Jadidi y L. Simpson. 2022. SCEVD: Integración de código semánticamente mejorada para el descubrimiento de vulnerabilidades. En las Actas de la Conferencia Internacional IEEE de 2022 sobre Confianza, Seguridad y Privacidad en Informática y Comunicaciones (TrustCom'22). DOI: http://doi.org/10.1109/TrustCom56396.2022.00217 [107] Y. Wu, J. Lu, Y.
Zhang y S. Jin. 2021. Detección de vulnerabilidades en código fuente de C/C++ con aprendizaje de representación gráfica. En las Actas del 11.º Taller y Conferencia Anual de Computación y Comunicación del IEEE de 2021 (CCWC'21). DOI: http://doi.org/10.1109/CCWC51732.2021.9376145
[108] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He y S. Ji. 2021. Detección de vulnerabilidades de contratos inteligentes: de red neuronal pura a fusión de características de gráficos interpretables y patrones expertos. arXiv:cs.LG/2106.09282.
[109] LK Shar, LC Briand y HBK Tan. 2015. Predicción de vulnerabilidades en aplicaciones web mediante análisis híbrido de programas y aprendizaje automático. IEEE Transactions on Dependable and Secure Computing 12, 6 (2015), 688–707. DOI: http://doi.org/10.1109/TDSC.2014.2373377

[110] D. Zou, S. Wang, S. Xu, Z. Li y H. Jin. 2021. VulDeePecker: Un sistema basado en aprendizaje profundo para la detección de vulnerabilidades multiclase. IEEE Transactions on Dependable and Secure Computing 18 (2021), 2224–2236. DOI: http://doi.org/10.1109/ TDSC.2019.2942930

[111] W. Zheng, Y. Jiang y X. Su. 2021. Vu1SPG: Detección de vulnerabilidades basada en el aprendizaje de la representación de grafos de propiedades de corte. En Actas del 32.º Simposio Internacional sobre Ingeniería de Confiabilidad del Software del IEEE de 2021 (ISSRE'21). DOI: http://doi.org/10.1109/ISSRE52982.2021.00054 [112] Y.
Xue, J. Guo, L. Zhang y H. Song. 2022. Redes neuronales de grafos de paso de mensajes para la detección de vulnerabilidades de seguridad de software. En las Actas de la Conferencia Internacional sobre Redes Informáticas, Electrónica y Automatización de 2022 (ICCNEA'22). DOI: http://doi.org/10.1109/ICCNEA57056.2022.00041
[113] N. Sarbakysh y Z. Wang. 2023. A1BERT: Un modelo de red neuronal de grafos independiente del lenguaje para la detección de vulnerabilidades. En Actas de la 8.ª Conferencia Internacional sobre Ciencia de Datos en el Ciberespacio (DSC'23) de 2023. DOI: http://doi.org/10.1109/DSC59305.2023.00038
[114] HV Nguyen, J. Zheng, A. Inomata y T. Uehara. 2022. Gráfico de agregación de código: Representación eficaz para redes neuronales de grafos con el fin de detectar código vulnerable. IEEE Access 10 (2022), 123786–123800. DOI: http://doi.org/10.1109/ACCESS. 2022.3216395

Machine Translated by Google

[115] Z. Chen, S. Kommrusch y M. Monperrus. 2023. Aprendizaje por transferencia neuronal para la reparación de vulnerabilidades de seguridad en código C. IEEE Transactions on Software Engineering 49, 1 (2023), 147–165. DOI: http://doi.org/10.1109/
TSE.2022.3147265 [116] Z. Yu, C. Theisen, L. Williams y T. Menzies. 2021. Mejora de la eficiencia de la inspección de vulnerabilidades mediante el aprendizaje
IEEE Transactions on Software Engineering 47, 11 (2021), 2401–2420. DOI: http://doi.org/10.1109/TSE.2019.2949275 [117] A. Bagheri y P.
Hegedűs. 2021. Comparación de diferentes métodos de representación de código fuente para la predicción de vulnerabilidades en Python. En Calidad de la Tecnología de la Información y las Comunicaciones. Comunicaciones en Ciencias de la Computación e Información, vol. 1439. Springer, 267–281.

[118] D. Hin, A. Kan, H. Chen y MA Babar. 2022. LineVD: Detección de vulnerabilidades a nivel de declaración mediante redes neuronales de grafos. En las Actas de la 19.ª Conferencia Internacional sobre Minería de Repositorios de Software (MSR'22). DOI: http://doi.org/10.1145/3524842.3527949

[119] M. Fu y C. Tantithamthavorn. 2022. LineVul: Predicción de vulnerabilidad a nivel de línea basada en transformador. En Actas de la 19.ª Conferencia Internacional IEEE/ACM sobre Minería de Repositorios de Software (MSR'22) de 2022. DOI: http://doi.org/10.1145/3524842.3528452

[120] B. Mosolygó, N. Vándor, G. Antal, P. Hegedűs y R. Ferenc. 2021. Hacia un modelo explicable de predicción de vulnerabilidades de JavaScript basado en prototipos. En las Actas de la Conferencia Internacional sobre Calidad del Código de 2021 (ICCQ'21). DOI: http://doi.org/10.1109/ICCQ51190.2021.9392984 [121] A. Mazuera-Rozo, A. Mojica-
Hanke, M. Linares-Vásquez y G. Bavota. 2021. ¿Superficial o profundo? Un estudio empírico sobre la detección de vulnerabilidades mediante aprendizaje profundo. En las Actas de la 29.ª Conferencia Internacional sobre Comprensión de Programas IEEE/ACM de 2021 (ICPC'21). DOI: http://doi.org/10.1109/ICPC52881.2021.00034 [122] H. Hanif y S. Maffeis. 2022. VulBERTa: Preentrenamiento simplificado
de código fuente para la detección de vulnerabilidades. En Actas de la Conferencia Conjunta Internacional sobre Redes Neuronales de 2022 (IJCNN'22). DOI: http://doi.org/10.1109/IJCNN55064.2022.9892280

[123] HK Dam, T. Tran, T. Pham, SW Ng, J. Grundy y A. Ghose. 2021. Aprendizaje automático de características para predecir componentes de software vulnerables. IEEE Transactions on Software Engineering 47, 1 (2021), 67–85. DOI: http://doi.org/10.1109/TSE.2018.2881961

[124] THM Le, D. Hin, R. Croft y MA Babar. 2022. DeepCVA: Evaluación automatizada de vulnerabilidades a nivel de confirmación con aprendizaje profundo multitarea. En las Actas de la 36.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASE'21). DOI: http://doi.org/10.1109/ASE51524.2021.9678622 [125] THM Le, B. Sabir y MA Babar. 2019. Evaluación
automatizada de vulnerabilidad de software con deriva de concepto.
En las Actas de la 16.ª Conferencia Internacional IEEE/ACM sobre Minería de Repositorios de Software (MSR'19) de 2019. DOI: http://doi.org/10.1109/MSR.2019.00063

[126] X. Li, L. Wang, Y. Xin, Y. Yang y Y. Chen. 2020. Detección automatizada de vulnerabilidades en código fuente mediante aprendizaje de representación intermedia mínima. Applied Sciences 10, 5 (2020), 1692. DOI: http://doi.org/10.3390/app10051692 [127] V. Nguyen, T. Le, T. Le, K. Nguyen, O.
DeVel, P. Montague, L. Qu y D. Phung. 2019. Adaptación profunda del dominio para la identificación de funciones de código vulnerable. En las Actas de la Conferencia Conjunta Internacional sobre Redes Neuronales de 2019 (IJCNN'19). DOI: http://doi.org/10.1109/IJCNN.2019.8851923

[128] M.-J. Choi, S. Jeong, H. Oh y J. Choo. 2017. Predicción de extremo a extremo de desbordamientos de búfer a partir de código fuente sin procesar a través de redes de memoria neuronal. arXiv:cs.SE/1703.02458.

[129] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang y A. E. Hassan. 2021. Encontrar una aguja en un pajar: Minería automatizada de correcciones de vulnerabilidades silenciosas. En las Actas de la 36.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASE'21) de 2021. DOI: http://doi.org/10.1109/ASE51524.2021.9678720

[130] G. Nguyen-Truong, H. J. Kang, D. Lo, A. Sharma, A. E. Santosa, A. Sharma y M. Y. Ang. 2022. HERMES: Uso de la vinculación entre commits y issues para detectar commits de corrección de vulnerabilidades. En las Actas de la Conferencia Internacional IEEE de 2022 sobre Análisis, Evolución y Reingeniería de Software (SANER'22). DOI: http://doi.org/10.1109/SANER53432.2022.00018 [131] Y. Zhou, JK Siow, C. Wang, S. Liu
y Y. Liu. 2021. SPI: Identificación automatizada de parches de seguridad a través de confirmaciones.
ACM Transactions on Software Engineering and Methodology 31, 1 (2021), Artículo 13, 27 páginas. DOI: http://doi.org/10.1145/3468854

[132] K. Filus, M. Siavvas, J. Domańska y E. Gelenbe. 2021. La red neuronal aleatoria como modelo de enlace para la predicción de vulnerabilidades de software. En Modelado, análisis y simulación de sistemas informáticos y de telecomunicaciones. Apuntes de clase en informática, vol. 12527. Springer, 102–116.

[133] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl e Y. Acar. 2015. VCCFinder: Detección de posibles vulnerabilidades en proyectos de código abierto para facilitar las auditorías de código. En las Actas de la 22.ª Conferencia ACM SIGSAC sobre Seguridad Informática y de las Comunicaciones (CCS'15). DOI: http://doi.org/10.1145/2810103.2813604 [134] A. Xu, T. Dai, H. Chen, Z. Ming y W. Li.
2018. Detección de vulnerabilidades en código fuente mediante LSTM contextual. En las Actas de la 5.ª Conferencia Internacional de Sistemas e Informática (ICSAI'18) de 2018. DOI: http://doi.org/10.1109/ ICSAI.2018.8599360

Machine Translated by Google

[135] N. Ziems y S. Wu. 2021. Detección de vulnerabilidades de seguridad mediante el procesamiento del lenguaje natural con aprendizaje profundo. En las Actas de la Conferencia IEEE sobre Talleres de Comunicaciones Informáticas (INFOCOM WKSHPS'21). DOI: http:// // doi.org/10.1109/INFOCOMWKSHPS51825.2021.9484500

[136] D. Cao, J. Huang, X. Zhang y X. Liu. 2020. FTCLNet: LSTM convolucional con transformada de Fourier para la detección de vulnerabilidades . En Actas de la 19.ª Conferencia Internacional IEEE sobre Confianza, Seguridad y Privacidad en Informática y Comunicaciones (TrustCom'20) de 2020. DOI: http://doi.org/10.1109/TrustCom50675.2020.00078 [137] X.

Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun y F. Geck. 2021. PatchRNN: un sistema basado en aprendizaje profundo para la identificación de parches de seguridad. En actas de la Conferencia de Comunicaciones Militares del IEEE de 2021 (MILCOM'21). DOI: http://doi.org/10.1109/MILCOM52596.2021.9652940 [138] C. Mamede, E.

Pinconschi, R. Abreu y J. Campos. 2022. Exploración de transformadores para la clasificación multietiqueta de vulnerabilidades de Java. En las Actas de la 22.ª Conferencia Internacional del IEEE sobre Calidad, Confiabilidad y Seguridad del Software (QRS'22) de 2022. DOI: http://doi.org/10.1109/QRS57517.2022.00015

[139] FR Abdulhamza y RJS Al-Janabi. 2022. Detección de inyección SQL mediante redes neuronales convolucionales 2D (2D-CNN). En Actas de la Conferencia Internacional sobre Ciencia de Datos y Computación Inteligente de 2022 (ICDSIC'22). DOI: http://doi.org/10.1109/ICDSIC56987.2022.10075777 [140]

S. Ndichu, S. Ozawa, T. Misu y K. Okada. 2018. Un enfoque de aprendizaje automático para la detección de JavaScript malicioso mediante la representación vectorial de longitud fija. En las Actas de la Conferencia Conjunta Internacional sobre Redes Neuronales de 2018 (IJCNN'18). DOI: http://doi.org/10.1109/IJCNN.2018.8489414

[141] M. Mimura e Y. Suga. 2019. Filtrado de código JavaScript malicioso con Doc2Vec en un conjunto de datos desequilibrado. En Actas de la 14.ª Conferencia Conjunta Asiática sobre Seguridad de la Información (AsiaJCIS'19) de 2019. DOI: http://doi.org/ 10.1109/ AsiaJCIS.2019.000-9

[142] X. Zhou, Y. Chen, H. Guo, X. Chen y Y. Huang. 2023. Recomendaciones de código de seguridad para contratos inteligentes. En Actas de la Conferencia Internacional IEEE de 2023 sobre Análisis, Evolución y Reingeniería de Software (SANER'23). DOI: http://doi.org/10.1109/SANER56733.2023.00027

[143] Z. Liu, Q. Liao, W. Gu y C. Gao. 2023. Detección de vulnerabilidades de software con GPT y aprendizaje contextual. En Actas de la 8.ª Conferencia Internacional sobre Ciencia de Datos en el Ciberespacio (DSC'23) de 2023. DOI: http://doi.org/10. 1109/DSC59305.2023.00041

[144] Y. Wei, L. Bo, X. Wu, Y. Li, Z. Ye, X. Sun y B. Li. 2023. VulRep: Reparación de vulnerabilidades basada en la inducción y corrección de confirmaciones. Revista EURASIP sobre Comunicaciones Inalámbricas y Redes 2023, 1 (2023), Artículo 34. DOI: http://doi.org/10.1186/ s13638-023-02242-7

[145] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie y Shi Wu. 2020. CodeCMR: Recuperación intermodal para la correspondencia de código fuente binario a nivel de función. En las Actas de la 34.ª Conferencia sobre Sistemas de Procesamiento de Información Neural (NeurIPS'20). https://proceedings.neurips.cc/paper_files/paper/2020/file/285f89b802bcb2651801455c86d78f2a-Paper.pdf

[146] R. Ferenc, P. Hegedűs, P. Gyimesi, G. Antal, D. Bán y T. Gyimóthy. 2019. Desafiando algoritmos de aprendizaje automático en la predicción de funciones vulnerables de JavaScript. En Actas del 7.º Taller Internacional IEEE/ACM de 2019 sobre la Realización de Sinergias de Inteligencia Artificial en la Ingeniería de Software (RAISE'19). DOI: http://doi.org/10.1109/RAISE.2019.00010 [147]

AD Sawadogo, TF Bissyandé, N. Moha, K. Allix, J. Klein, L. Li y YL Traon. 2020. Aprendiendo a detectar parches de seguridad. arXiv:cs.SE/ 2001.09148.

[148] I. Chowdhury y M. Zulkernine. 2011. Uso de métricas de complejidad, acoplamiento y cohesión como indicadores tempranos de vulnerabilidades. Journal of Systems Architecture 57, 3 (2011), 294–313. DOI: http://doi.org/10.1016/j.sysarc.2010.06.003 [149] L.

Yang, X. Li y Y. Yu. 2017. VulDigger: Una herramienta oportuna y económica para la excavación de cambios que contribuyen a la vulnerabilidad. En Actas de la Conferencia de Comunicaciones Globales del IEEE de 2017 (GLOBECOM'17). DOI: http://doi.org/10. 1109/GLOCOM.2017.8254428

[150] L. Kumar, C. Hota, A. Mahindru y LBM Neti. 2019. Predicción de malware para Android mediante una máquina de aprendizaje extremo con diferentes funciones de kernel. En las Actas de la 15.ª Conferencia Asiática de Ingeniería de Internet (AINTEC'19). DOI: http:// doi.org/10.1145/3340422.3343639

[151] Z. Wang, J. Guo y H. Li. 2021. Modelo de extracción de características de vulnerabilidad para código fuente basado en aprendizaje profundo. En las Actas de la Conferencia Internacional sobre Redes Informáticas, Electrónica y Automatización de 2021 (ICCNEA'21). DOI: http://doi.org/10.1109/ICCNEA53019.2021.00016

[152] M. Zagane, MK Abdi y M. Alenezi. 2020. Aprendizaje profundo para la detección de vulnerabilidades de software utilizando métricas de código. IEEE Access 8 (2020), 74562–74570. DOI: http://doi.org/10.1109/ACCESS.2020.2988557 [153] S.

Ganesh, T. Ohlsson y F. Palma. 2021. Predicción de vulnerabilidades de seguridad mediante métricas de código fuente. En las Actas del Taller Sueco sobre Ciencia de Datos de 2021 (SweDS'21). DOI: http://doi.org/10.1109/SweDS53855.2021.9638301 [154] T.-Y.

Chong, V. Anu y KZ Sultana. 2019. Uso de métricas de software para predecir componentes de código vulnerables: Un estudio sobre proyectos de código abierto de Java y Python. En las Actas de la Conferencia Internacional IEEE sobre Ciencia e Ingeniería Computacional (CSE'19) de 2019 y la Conferencia Internacional IEEE sobre Computación Integrada y Ubicua (EUC'19). DOI: http://doi.org/10.1109/ CSE/EUC.2019.00028

Machine Translated by Google

[155] N. Medeiros, N. Ivaki, P. Costa y M. Vieira. 2020. Detección de código vulnerable mediante métricas de software y aprendizaje automático. IEEE Access 8 (2020), 219174–219198. DOI: http://doi.org/10.1109/ACCESS.2020.3041181

[156] M. Hasan, Z. Balbahaith y M. Tarique. 2019. Detección de ataques de inyección SQL: Un enfoque de aprendizaje automático. En Actas de la Conferencia Internacional sobre Tecnologías y Aplicaciones Eléctricas e Informáticas de 2019 (ICECTA'19). DOI: http://doi.org/10.1109/ICECTA48151.2019.8959617 [157] M.-H. Tsai,

C.-C. Lin, Z.-G. He, W.-C. Yang y C.-L. Lei. 2023. PowerDP: Desofuscación y perfilado de comandos maliciosos de PowerShell con clasificadores multietiqueta. IEEE Access 11 (2023), 256–270. DOI: http://doi.org/10.1109/ACCESS. 2022.3232505

[158] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu y Z. Chen. 2022. SySeVR: Un marco para usar el aprendizaje profundo para detectar vulnerabilidades de software. IEEE Transactions on Dependable and Secure Computing 19 (2022), 224–2258. DOI: http:// //doi.org/ 10.1109/TDSC.2021.3051525

[159] A. Mahyari. 2022. Una red neuronal profunda jerárquica para detectar líneas de código con vulnerabilidades. En Actas de la 22.ª Conferencia Internacional IEEE sobre Calidad, Confiabilidad y Seguridad del Software (QRS-C'22) de 2022. DOI: http://doi.org/10.1109/QRS-C57518.2022.00011

[160] C. Thapa, SI Jang, ME Ahmed, S. Camtepe, J. Pieprzyk y S. Nepal. 2022. Modelos de lenguaje basados en transformadores para la detección de vulnerabilidades de software. En las Actas de la 38.ª Conferencia Anual de Aplicaciones de Seguridad Informática (ACSAC'22). DOI: http://doi.org/10.1145/3564625.3567985

[161] Z. Tang, Q. Hu, Y. Hu, W. Kuang y J. Chen. 2022. SEVulDet: Un detector de vulnerabilidad aprendible y mejorado semánticamente. En Actas de la 52.a Conferencia Internacional Anual IEEE/IFIP de 2022 sobre Sistemas y Redes Confiables (DSN'22). DOI: http://doi.org/10.1109/DSN53405.2022.00026

[162] S. Kim, J. Choi, ME Ahmed, S. Nepal y H. Kim. 2022. VulDeBERT: Un sistema de detección de vulnerabilidades que utiliza BERT. En Actas de los talleres del Simposio Internacional IEEE 2022 sobre Ingeniería de Confiabilidad de Software (ISSREW'22). DOI: http://doi.org/10.1109/ISSREW55968.2022.00042

[163] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague e Y. Xiang. 2022. CD-VulD: Descubrimiento de vulnerabilidades entre dominios basado en la adaptación profunda de dominios. IEEE Transactions on Dependable and Secure Computing 19, 1 (2022), 438–451. DOI: http://doi.org/10.1109/TDSC.2020.2984505

[164] J.-W. Liao, T.-T. Tsai, C.-K. He y C.-W. Tien. 2019. SoliAudit: Evaluación de vulnerabilidades de contratos inteligentes basada en aprendizaje automático y pruebas fuzz. En las Actas de la VI Conferencia Internacional sobre Internet de las Cosas: Sistemas, Gestión y Seguridad (IOTSMS'19) de 2019. DOI: http://doi.org/10.1109/IOTSMS48152.2019.8939256 [165] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang y Y. Xiang. 2020.

DeepBalance: Aprendizaje profundo y sobremuestreo difuso para la detección de vulnerabilidades. IEEE Transactions on Fuzzy Systems 28, 7 (2020), 1329–1343. DOI: http://doi.org/10.1109/ TFUZZ.2019.2958558

[166] N. Ashizawa, N. Yanai, JP Cruz y S. Okamura. 2021. Eth2Vec: Aprendizaje de representaciones de código a nivel de contrato para la detección de vulnerabilidades en contratos inteligentes de Ethereum. En las Actas del 3.er Simposio Internacional de la ACM sobre Blockchain e Infraestructura Crítica Segura (BSCI'21). DOI: http://doi.org/10.1145/3457337.3457841

[167] R. Yan, X. Xiao, G. Hu, S. Peng y Y. Jiang. 2018. Nuevo método de aprendizaje profundo para detectar ataques de inyección de código en aplicaciones híbridas. Journal of Systems and Software 137 (2018), 67–77. DOI: http://doi.org/10.1016/j.jss.2017.11.001

[168] D. Coimbra, S. Reis, R. Abreu, C. Păsăreanu y H. Erdogmus. 2021. Sobre el uso de representaciones distribuidas de código fuente para la detección de vulnerabilidades de seguridad en C. arXiv:cs.CR/2106.01367.

[169] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague e Y. Xiang. 2021. Descubrimiento de vulnerabilidades de software mediante el aprendizaje de bases de conocimiento multidominio. IEEE Transactions on Dependable and Secure Computing 18, 5 (2021), 2469–2485. DOI: http://doi.org/10.1109/TDSC.2019.2954088

[170] Z. Bilgin, MA Ersoy, EU Soykan, E. Tomur, P. Çomak y L. Karaçay. 2020. Predicción de vulnerabilidades a partir del código fuente mediante aprendizaje automático. IEEE Access 8 (2020), 150672–150684. DOI: http://doi.org/10.1109/ACCESS.2020.3016774

[171] H. Feng, X. Fu, H. Sun, H. Wang y Y. Zhang. 2020. Detección eficiente de vulnerabilidades basada en árboles de sintaxis abstracta y aprendizaje profundo. En Actas de la Conferencia IEEE sobre Talleres de Comunicaciones Informáticas (INFOCOM WKSHPS'20). DOI: http://doi.org/10.1109/INFOCOMWKSHPS50562.2020.9163061

[172] M. Gu, H. Feng, H. Sun, P. Liu, Q. Yue, J. Hu, C. Cao e Y. Zhang. 2022. Red de atención jerárquica para la detección de vulnerabilidades interpretable y de grano fino. En las Actas de los Talleres de la Conferencia IEEE sobre Comunicaciones Informáticas (INFOCOM WKSHPS'22). DOI: http://doi.org/10.1109/INFOCOMWKSHPS54753.2022.9798297

[173] Y. Mao, Y. Li, J. Sun y Y. Chen. 2020. Detección explicable de vulnerabilidades de software basada en redes neuronales recurrentes bidireccionales basadas en la atención. En las Actas de la Conferencia Internacional IEEE sobre Big Data de 2020 (Big Data'20). DOI: http://doi.org/10.1109/BigData50022.2020.9377803

[174] Y. He, H. Sun y H. Feng. 2020. UA-Miner: Sistemas de aprendizaje profundo para exponer vulnerabilidades de API no protegidas en el código fuente. En Actas de la 12.ª Conferencia Internacional sobre Inteligencia Computacional Avanzada (ICACI'20) de 2020. DOI: http://doi.org/10.1109/ICACI49185.2020.9177528

Machine Translated by Google

[175] P. Momeni, Y. Wang y R. Samavi. 2019. Modelo de aprendizaje automático para el análisis de seguridad de contratos inteligentes. En Actas de la 17.ª Conferencia Internacional sobre Privacidad, Seguridad y Confianza (PST'19) de 2019. DOI: http://doi.org/10.1109/PST47121. 2019.8949045

[176] X. Yan, S. Wang y K. Gai. 2022. Un método basado en análisis semántico para la vulnerabilidad de contratos inteligentes. En las Actas de la 8.ª Conferencia Internacional del IEEE sobre Seguridad de Big Data en la Nube (BigDataSecurity'22) de 2022, la Conferencia Internacional del IEEE sobre Alto Rendimiento y Computación Inteligente (HPSC'22) y la Conferencia Internacional del IEEE sobre Datos Inteligentes y Seguridad (IDS'22). DOI: http://doi.org/10.1109/BigDataSecurityHPSCIDS54978.2022.00015 [177] H. Shi, R. Wang,
Y. Fu, Y. Jiang, J. Dong, K. Tang y J. Sun. 2019. Detección de clones de código vulnerables para sistemas operativos mediante aprendizaje inducido por correlación. IEEE Transactions on Industrial Informatics 15, 12 (2019), 6551–6559.
DOI: http://doi.org/10.1109/TII.2019.2929739
[178] W. Zheng, AO Abdallah Semasaba, X. Wu, SA Agyemang, T. Liu e Y. Ge. 2021. Representación vs. modelo: Lo más importante para la detección de vulnerabilidades en el código fuente. En las Actas de la Conferencia Internacional IEEE de 2021 sobre Análisis, Evolución y Reingeniería de Software (SANER'21). DOI: http://doi.org/10.1109/SANER50967.2021.00082 [179] M. Ceccato, CD Nguyen, D. Appelt
y LC Briand. 2016. SOFIA: Un oráculo de seguridad automatizado para pruebas de caja negra de vulnerabilidades de inyección SQL. En las Actas de la 31.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada (ASE'16). DOI: http://doi.org/
10.1145/2970276.2970343 [180] X. Liu, X. Du, Q. Lei y K. Liu. 2020. Clasificación
multifamiliar de malware para Android con una estrategia difusa para resistir variantes familiares polimórficas. IEEE Access 8 (2020), 156900–
156914. DOI: http://doi.org/10.1109/ACCESS.2020.3019282 [181] E. Spirin, E. Bogomolov, V. Kovalenko y T. Bryksin. 2021. PSIMiner:
Una herramienta para la minería de árboles de sintaxis abstracta enriquecidos a partir del código. En Actas de la 18.ª Conferencia Internacional IEEE/ACM sobre Minería de Repositorios de Software (MSR'21) de 2021.
DOI: http://doi.org/10.1109/MSR52588.2021.00014 [182]
U. Alon, M. Zilberstein, O. Levy y E. Yahav. 2018. code2vec: Aprendizaje de representaciones distribuidas de código.
arXiv:cs.LG/1803.09473.
[183] G. Buehrer, B.W. Weide y PAG. Sivilotti. 2005. Uso de la validación del árbol de análisis para prevenir ataques de inyección SQL. En las Actas del 5.º Taller Internacional sobre Ingeniería de Software y Middleware (SEM'05). DOI: http://doi.org/10. 1145/1108473.1108496

[184] G. Lin, J. Zhang, W. Luo, L. Pan y Y. Xiang. 2017. PÓSTER: Descubrimiento de vulnerabilidades con aprendizaje de representación de funciones a partir de proyectos sin etiquetar. En las Actas de la Conferencia ACM SIGSAC de 2017 sobre Seguridad Informática y de las Comunicaciones (CCS'17). DOI: http://doi.org/10.1145/3133956.3138840
[185] DG Fritz y RG Sargent. 1995. Una visión general de los modelos de grafos de flujo de control jerárquico. En Actas de la 27.ª Conferencia sobre Simulación Invernal (WSC'95). DOI: http://doi.org/10.1145/224401.224819 [186] S. Sinha. 2002.
Análisis estático y dinámico de programas que contienen flujo de control interprocedimental arbitrario. Ph.D.
Tesis. Instituto Tecnológico de Georgia. http://proxy.library.nd.edu/login?url=https://www.proquest.com/ disertaciones-tesis/análisis estático-dinámico-programas-que-contienen/docview/305602235/se-2 [187] J. Ferrante, K. J. Ottenstein y J. D. Warren.
1987. El grafo de dependencia del programa y su uso en optimización. ACM Transactions on Programming Languages and Systems 9, 3 (1987), 319–349. DOI: http://doi.org/10.1145/24039.24041 [188] D. Grove y C. Chambers. 2001. Un marco para algoritmos de construcción de
grafos de llamadas. ACM Transactions on Programming Languages and Systems 23, 6 (2001), 685–746. DOI: http://doi.org/10.1145/506315.506316
[189] M. Weiser. 1984. Segmentación de programas. IEEE Transactions on Software Engineering 4 (1984),
352–357.
[190] Y. Sui, X. Cheng, G. Zhang y H. Wang. 2020. Flow2Vec: Incrustación precisa de código basada en flujo de valor. Actas de la ACM sobre Lenguajes de Programación 4, OOPSLA (2020), Artículo 233, 27 páginas. DOI: http://doi.org/10.1145/3428301 [191] B. Steffen, J. Knoop
y O. Rüthing. 1990. El grafo de flujo de valor: Una representación de programa para transformaciones óptimas de programa. En ESOP'90. Apuntes de clase en Ciencias de la Computación, vol. 432. Springer, pp. 389–405. https://link.springer. com/content/pdf/10.1007/3-540-52592-0_76.pdf
[192] M. Zhang, Y. Duan, H. Yin y Z. Zhao. 2014.
Clasificación de malware para Android con enfoque semántico mediante gráficos de dependencia de API contextuales ponderados. En las Actas de la Conferencia ACM SIGSAC de 2014 sobre Seguridad Informática y de las Comunicaciones. DOI: http://doi.org/10.1145/2660267.2660359
[193] Y. Zhang, X. Yu, Z. Cui, S. Wu, Z. Wen y L. Wang. 2020.
Cada documento posee su propia estructura: texto inductivo.
Clasificación mediante redes neuronales gráficas. arXiv:cs.CL/2004.13826.
[194] T. Xu y P. Zhou. 2022. Extracción de características para la clasificación de carga útil: Un algoritmo de codificación de pares de bytes. En Actas de la 8.ª Conferencia Internacional sobre Informática y Comunicaciones del IEEE de 2022 (ICCC'22). DOI: http://doi.org/10.1109/ ICCC56324.2022.10065977

[195] T. Kudo y J. Richardson. 2018. SentencePiece: Un tokenizador y destokenizador de subpalabras simple e independiente del lenguaje.
para el procesamiento neuronal de texto. arXiv:cs.CL/1808.06226. https://arxiv.org/abs/1808.06226
[196] X. Song, A. Salcianu, Y. Song, D. Dopson y D. Zhou. 2021. Tokenización rápida de WordPiece. arXiv:cs.CL/2012.15524. https://arxiv.org/abs/
2012.15524
[197] QV Le y T. Mikolov. 2014. Representaciones distribuidas de oraciones y documentos. arXiv:cs.CL/1405.4053.

Machine Translated by Google

[198] Meiliana, S. Karim, HLHS Warnars, FL Gaol, E. Abdurachman y B. Soewito. 2017. Métricas de software para la predicción de fallos mediante enfoques de aprendizaje automático: Una revisión bibliográfica con el conjunto de datos del repositorio PROMISE. En las Actas de la Conferencia Internacional IEEE de 2017 sobre Cibernética e Inteligencia Computacional (CyberneticsCom'17). DOI: http://doi.org/10.1109/CYBERNETICSCOM.2017.8311708

[199] I. Santos, F. Brezo, X. Ugarte-Pedrero y PG Bringas. 2013. Secuencias de código de operación como representación de ejecutables para la detección de malware desconocido mediante minería de datos. Ciencias de la Información 231 (2013), 64–82. DOI: http://doi.org/10.1016/j. ins.2011.08.020

[200] B. Min, H. Ross, E. Sulem, APB Veyseh, TH Nguyen, O. Sainz, E. Agirre, I. Heinz y D. Roth. 2021. Avances recientes en el procesamiento del lenguaje natural a través de grandes modelos de lenguaje preentrenados: una encuesta. arXiv:cs.CL/2111.01243. https://arxiv.org/abs/2111.01243 [201] P. Kruchten. 2009. El proceso racional unificado: una introducción. Addison-Wesley.

[202] Y. Gui, Y. Wan, H. Zhang, H. Huang, Y. Sui, G. Xu, Z. Shao y H. Jin. 2022. Coincidencia de código fuente binario entre lenguajes con representaciones intermedias. arXiv:cs.SE/2201.07420. https://arxiv.org/abs/2201.07420 [203] Z. Chen, S. Kommrusch y M. Monperrus. 2023. Aprendizaje por transferencia neuronal para la reparación de vulnerabilidades de seguridad en código C. IEEE Transactions on Software Engineering 49, 1 (2023), 147–165. DOI: http://doi.org/10.1109/tse.2022.3147265 [204] Y. Chen, Z. Ding, L. Alowain, X. Chen y D. Wagner. 2023. DiverseVul: Un nuevo conjunto de datos de código fuente vulnerable para Detección de vulnerabilidades basada en aprendizaje profundo. arXiv:cs.CR/2304.00409.

[205] RL Russell, L. Kim, LH Hamilton, T. Lazovich, JA Harer, O. Ozdemir, PM Ellingwood y MW McConley. 2018. Detección automatizada de vulnerabilidades en código fuente mediante aprendizaje de representación profunda. arXiv:cs.LG/1807.04320.

[206] T. Boland y PE Black. 2012. Conjunto de pruebas Juliet 1.1 para C/C++ y Java. Computer 45, 10 (2012), 88–90. DOI: http:// //doi.org/10.1109/MC.2012.345 [207] L. Szekeres, M. Payer, T. Wei y D. Song. 2013. SoK: Guerra eterna en la memoria. En Actas del Simposio del IEEE de 2013. sobre seguridad y privacidad. DOI: http://doi.org/10.1109/SP.2013.13

[208] J. K. Siow, S. Liu, X. Xie, G. Meng e Y. Liu. 2022. Aprendizaje de la semántica de programas con representaciones de código: Un estudio empírico. En las Actas de la Conferencia Internacional IEEE de 2022 sobre Análisis, Evolución y Reingeniería de Software (SANER'22). DOI: http://doi.org/10.1109/SANER53432.2022.00073

[209] Y. Hu, A. Huber, J. Anumula y S. Liu. 2019. Superar el problema del gradiente evanescente en redes recurrentes simples. arXiv:cs.NE/1801.06105.

[210] P. Runeson y M. Höst. 2009. Directrices para la realización y presentación de informes de investigaciones de estudios de casos en ingeniería de software. Ingeniería de software empírica 14 (2009), 131–164. DOI: http://doi.org/10.1007/s10664-008-9102-8 [211] J. Cohen. 1960. Un coeficiente de concordancia para escalas nominales. Educational and Psychological Measurement 20, 1 (1960), 37–46. DOI: http://doi.org/10.1177/001316446002000104

Apéndice A

Ejemplos de Representaciones de Código Fuente de Uso Común A.1 Ejemplos de Representaciones de Código Fuente Basadas en Árboles

La Figura 6 muestra cómo los AST y los árboles de análisis sintáctico difieren para el mismo código Python. Observe cómo el árbol de análisis sintáctico retuvo todos los símbolos del código (p. ej., nuevas líneas y sangría), mientras que el AST los abstrajo. Dado que los árboles de análisis sintáctico son más detallados que los AST, el trabajo [179] que los utilizó incluyó un paso de preprocesamiento que eliminó del árbol los nodos irrelevantes para detectar ataques (p. ej., identificadores de usuario específicos).

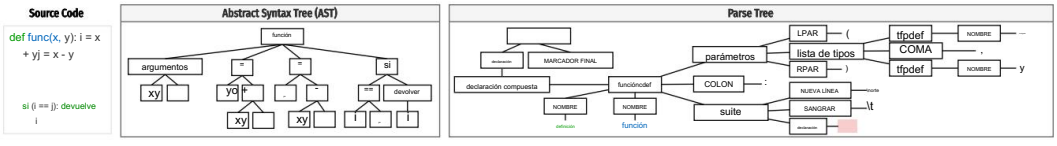

Fig. 6. Ejemplos de representaciones de árboles para el mismo código fuente de Python.

Machine Translated by Google

A.2 Ejemplos de representaciones basadas en grafos . La Figura 7 muestra el CFG de la función func(x,y). Cuenta con un bloque básico de entrada y uno de salida, que indican el inicio y el final de la ejecución de la función, respectivamente. El bloque de entrada está conectado a un bloque básico con tres instrucciones (i=x+y, j=xy y if i==j). Este bloque básico tiene dos aristas de salida: una representa el flujo cuando la condición if se evalúa como verdadera y la otra lo representa cuando se evalúa como falsa. Tras ejecutar una de las ramas, la ejecución fluye hacia el bloque de salida.

Además, la Figura 7 muestra el ICFG para todo el programa, incluyendo func(x,y) y main(). Tiene dos bloques de entrada: uno para main, donde se inicia el programa, y otro para func. También tiene dos bloques de salida que indican el final de la ejecución de las funciones main y func. El bloque de entrada para main está conectado al bloque básico que llama a func, que, a su vez, está conectado al bloque de entrada del receptor de la llamada. Para capturar el flujo desde la función invocada (es decir, func(x,y)) hasta su invocador (es decir, main()), esta representación incluye una arista desde el bloque de salida del receptor de la llamada hasta el bloque de retorno del receptor de la llamada. En la misma figura, también se muestra la DFG de una función de Python. Hay dos aristas de salida i=x+y hacia dos sentencias que usan la variable i. De forma similar, hay una arista de j=xy hacia si i==j porque la expresión usa la variable j.

La Figura 7 muestra un ejemplo de un PDG para una función de Python (func(x,y)), donde las líneas discontinuas y rectas representan las dependencias de control y de datos, respectivamente. El PDG comienza con un nodo de entrada para este procedimiento. Tiene dos parámetros en los nodos para representar los parámetros de la función (x e y). Para que se ejecuten las sentencias de las líneas 2 a 4, la entrada de la función debe haberse ejecutado. Por lo tanto, existe una dependencia de control desde la entrada hasta los nodos que representan estas líneas de código: i=x+y, j=xy y if i == j. Dado que las sentencias de las líneas 2 y 3 definen variables que se utilizan en la condición if , existe una dependencia de datos desde estas sentencias de asignación de variables hasta la condición if . También existe una dependencia de datos desde i =x+y hasta la sentencia return , ya que este nodo return utiliza la variable i. Dado que la sentencia return solo se ejecuta cuando la condición if se evalúa como verdadera, existe una dependencia de control entre el nodo if y el nodo return . Finalmente, la Figura 7 muestra el gráfico de llamadas para el fragmento de código Python proporcionado en ella.

En la Figura 8, se muestra el SDG para todo el programa. El SDG comienza con el nodo de entrada de main(), que es la función de punto de entrada en el grafo de llamadas. El primer nodo de entrada de main tiene una arista de flujo de control hacia func. Dos nodos de entrada actuales tienen aristas de flujo de datos, ya que los datos se inicializan desde main y fluyen hacia func. También hay dos nodos de entrada actuales que definen los parámetros que entran en el procedimiento func(x,y). El nodo de entrada en func tiene una arista de flujo de control hacia los nodos de instrucción que definen las variables i y j, y los dos nodos de entrada param tienen una arista de flujo de datos hacia esas mismas.

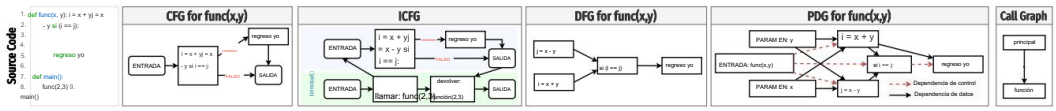

Fig. 7. Ejemplos de representaciones basadas en gráficos (CFG, ICFG, DFG, PDG y gráfico de llamadas) en el mismo código Python.

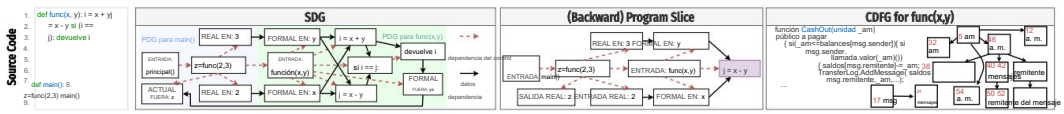

Fig. 8. Ejemplos de representaciones gráficas (SDG, Backward Program Slices y CDFG).

Machine Translated by Google

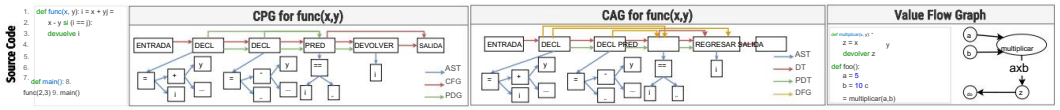

Fig. 9. Ejemplos de representaciones gráficas (CPG, CAG y VFG) en el mismo fragmento de código.

dos nodos; i=x+y y j=xy. Estos nodos de declaración tienen entonces un flujo de datos hacia el nodo de predicado si i == j. El nodo de entrada tiene una arista de dependencia de control hacia este nodo y los nodos in formales. Finalmente, este nodo de predicado tiene una arista de flujo de control hacia el bloque de retorno i , junto con el nodo i=x+y , excepto que este nodo tiene una arista de dependencia de datos. La línea discontinua en esta figura representa las dependencias de control, y la línea continua representa las dependencias de datos. La Figura 8 también muestra un ejemplo de una porción de programa hacia atrás sobre el SDG con el criterio de porción , 3 . Los nodos que tienen dependencias de control o datos hacia el nodo de interés (es decir, j = x - y) son parte de la porción de programa resultante. En la Figura 8, también vemos nuestros nodos cruciales para un CDFG: todos los nodos que gestionan mensajes, remitentes, saldos y _am. Estos nodos reciben o procesan datos y pueden ser responsables de una vulnerabilidad de reentrada.

La Figura 9 muestra la CPG para func(). El nodo de entrada se encuentra al inicio, mostrando la entrada a la función. Desde el nodo de entrada, se tiene una arista de flujo de control hacia el nodo de declaración, que se ramifica para mostrar los nodos y aristas AST de la línea i=x+y. Posteriormente, se tiene otra arista de flujo de control desde el nodo de declaración hacia otro nodo de declaración. Este nodo se ramifica hacia abajo para mostrar los nodos y aristas AST de la línea j=xy. Desde este nodo, se tiene una arista de flujo de control hacia un nodo de predicado. También se tienen aristas PDG desde el primer y segundo nodo de declaración hacia este nodo de predicado, ya que este último depende de los datos de ambos nodos de declaración. El nodo de predicado también tiene nodos y aristas AST para mostrar la línea i==j. Desde aquí, se tiene una arista de flujo de control hacia una sentencia de retorno. También se tiene una arista PDG hacia este nodo desde el nodo de predicado, ya que la sentencia de retorno depende del resultado de la arista de predicado. Finalmente, tenemos una arista de flujo de control hacia el nodo de salida desde el retorno. El nodo de salida también tiene una arista de flujo de control desde el nodo de predicado, ya que si el predicado se evalúa como falso, el programa termina. La misma figura también muestra un ejemplo de un CAG para la función func(). Comenzamos con el nodo de entrada, que tiene una arista de árbol dominador hacia el nodo DECL . El nodo DECL tiene nodos AST y aristas que representan la línea i=x+y. Desde el nodo DECL , tenemos una arista de árbol dominador y una arista de árbol postdominador hacia otro nodo DECL . Una vez más, el nodo DECL tiene nodos AST y aristas que representan la línea j=xy. A continuación, tenemos una arista de árbol dominador y una arista de árbol postdominador hacia el nodo PRED desde el nodo DECL . Además, tenemos una arista de dependencia de datos desde el nodo DECL que define i y el nodo DECL que define j nodos, ya que la sentencia if necesita los datos de ambos nodos para ejecutarse. El nodo PRED tiene una arista de árbol dominador y una arista de árbol postdominador hacia el nodo RETURN . El nodo RETURN tiene una arista de dependencia de datos del nodo DECL que define i, ya que la instrucción de retorno depende de los datos de ese nodo. Finalmente, tenemos una arista de árbol dominador hacia el nodo EXIT.

La Figura 9 muestra la función de flujo de datos (VFG) del fragmento de código proporcionado. Comenzamos con los nodos a y b. Estos dos nodos tienen una arista en el nodo multiply(x,y) , ya que los valores de a y b se pasan a esta función. Luego tenemos una arista en el nodo z, que representa la operación de multiplicación de a y b. Finalmente, z tiene una arista que lo conecta con c, que almacena el valor de la operación de multiplicación.

La Figura 10 muestra un ejemplo del CDG que comienza en el nodo del componente. Tenemos tres aristas: una para startActivity() que lleva a un nodo de página web, otra para triggerTasks() que lleva al nodo de tareas en segundo plano, y finalmente una arista para sendMessage() que lleva a un nodo de controlador de mensajes. Debajo del CDG, hay un ejemplo del CBG, como continuación del Gráfico de Comportamiento de la Aplicación Web más grande. El nodo de página web del CDG lleva a dos nodos diferentes en

Machine Translated by Google

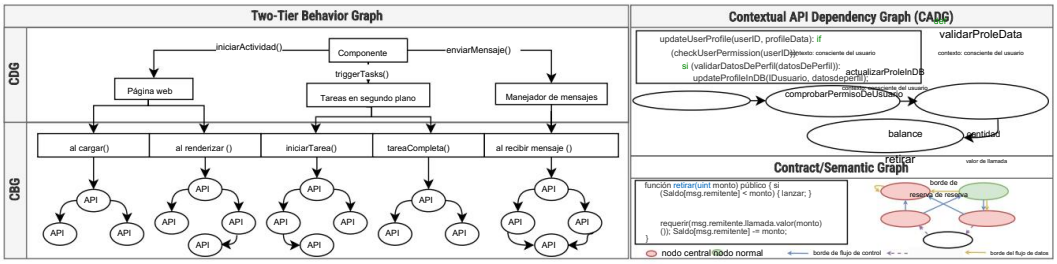

Fig. 10. Ejemplos de CDG, CBG y CADG.

El CBG: onLoad() y onRender(). Ambos nodos conducen a nodos API, que pueden representar cualquiera de los tipos de nodos mencionados. Este patrón continúa: el nodo de tareas en segundo plano del CDG conduce a los nodos beginTask() y complexTask() , que a su vez conducen a nodos API que representan las actividades posteriores de las API en el programa. Finalmente, el nodo de manejo de mensajes del CDG conduce al nodo onMessageReceive() del CBG, que conduce a otros nodos API que representan su funcionalidad dentro del programa.

La Figura 10 también muestra un ejemplo del CADG para el fragmento de código mostrado. Comenzamos con la función updateUserProfile y tenemos un borde de flujo de control hacia el nodo de seguridad checkUserPermission con el contexto "user-conscious". Desde aquí, pasamos a otro nodo de seguridad, validateProfileData , con el mismo contexto: "user-conscious". Finalmente, pasamos a updateProfileDB con el mismo contexto "user-conscious". La misma figura también muestra un grafo Contract-t/Semantic para el fragmento de código proporcionado. Los nodos "retreat" y " balance" son nodos principales, y la variable "call.value" es un nodo principal porque podrían ser la causa de una vulnerabilidad. La variable "mount" es un nodo normal, ya que no contribuye directamente a un problema de seguridad. La invocación de "retreat" tiene bordes de flujo de control hacia la invocación "balance" y la variable "r El nodo balance actúa como un borde de flujo de datos hacia sí mismo, ya que accede a datos de sí mismo y recibe o depende de datos de la variable amount. El nodo call.value también recibe datos de amount, por lo que existe un borde de flujo de datos de amount a call.value. Existen bordes de flujo de control de retirar a balance y amount , ya que invoca estos dos nodos. Tras llamar a amount , se invoca call.value , lo que resulta en un borde de flujo de control de amount a call.value. Finalmente, existe un borde de reserva de call.value al nodo de reserva , y de este a retirar .

A.3 Representaciones léxicas La

Figura 11 proporciona un ejemplo de los tokens resultantes de diferentes técnicas de tokenización, incluido un tokenizador estándar, tokenización de subpalabras BPE, SentencePiece y WordPiece.

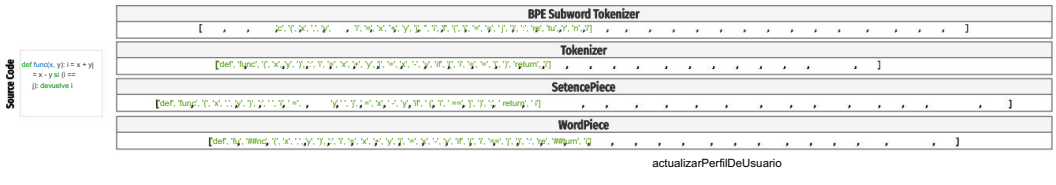

Fig. 11. Ejemplos de diferentes algoritmos de tokenización.

Recibido el 15 de marzo de 2024; revisado el 12 de febrero de 2025; aceptado el 24 de febrero de 2025