Un estudio empírico de herramientas de análisis estático para Revisión de código seguro

Wachiraphan Charoenwet Universidad de Melbourne, Australia

wcharoenwet@student.unimelb.edu.au

Patanamon Thongtanunam Universidad de Melbourne, Australia

patanamon.t@unimelb.edu.au

Van Thuan Pham Universidad de Melbourne, Australia

thuan.pham@unimelb.edu.au

Cristobal Treude Universidad de Gestión de Singapur

ctreude@smu.edu.sg

Abstracto

La identificación temprana de problemas de seguridad en el desarrollo de software es vital para minimizar sus impactos imprevistos. La revisión de código es un método de análisis manual ampliamente utilizado que busca descubrir problemas de seguridad, así como otros problemas de codificación, en proyectos de software. Si bien algunos estudios sugieren que las herramientas automatizadas de pruebas estáticas de seguridad de aplicaciones (SAST) podrían mejorar la identificación de problemas de seguridad, existe un conocimiento limitado sobre su eficacia práctica para respaldar la revisión segura de código. Además, la mayoría de los estudios SAST se basan en versiones sintéticas o totalmente vulnerables del programa en cuestión, que podrían no representar con precisión los cambios reales en el código durante el proceso de revisión.

Para abordar esta brecha, estudiamos SAST de C/C++ utilizando un conjunto de datos de cambios de código reales que contribuyeron a vulnerabilidades explotables. Más allá de la efectividad de SAST, cuantificamos los beneficios potenciales cuando las funciones modificadas se priorizan mediante advertencias de SAST. Nuestro conjunto de datos comprende 319 vulnerabilidades del mundo real de 815 confirmaciones que contribuyen a la vulnerabilidad (VCC) en 92 proyectos de C y C++. El resultado revela que un solo SAST puede producir advertencias en funciones vulnerables del 52% de las VCC. Priorizar las funciones modificadas con advertencias de SAST puede mejorar la precisión (es decir, el 12% de la precisión y el 5,6% de la recuperación) y reducir la falsa alarma inicial (líneas de código en funciones no vulnerables inspeccionadas hasta la primera función vulnerable) en un 13%. Sin embargo, al menos el 76% de las advertencias en funciones vulnerables son irrelevantes para las VCC, y el 22% de las VCC permanecen sin detectar debido a las limitaciones de las reglas de SAST. Nuestros hallazgos resaltan los beneficios y las brechas restantes de las revisiones de código seguro compatibles con SAST y los desafíos que deben abordarse en el trabajo futuro.

Conceptos CCS •

Seguridad y privacidad → Ingeniería de seguridad de software; • Software y su ingeniería → Creación y gestión de software .

ISSTA '24, 16 al 20 de septiembre de 2024, Viena, Austria

ACM ISBN 979-8-4007-0612-7/24/09 https://
doi.org/10.1145/3650212.3680313

Palabras clave

Revisión de código, herramienta de pruebas de seguridad de aplicaciones estáticas, cambios de código Priorización

Formato de Referencia ACM:

Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham y Christoph Treude. 2024. Un Estudio Empírico de Herramientas de Análisis Estático para la Revisión Segura de Código. En Actas del 33.º Simposio Internacional ACM SIGSOFT sobre Pruebas y Análisis de Software (ISSTA '24), 16-20 de septiembre de 2024, Viena, Austria. ACM, Nueva York, NY, EE. UU., 13 páginas. https://doi.org/10.1145/3650212.3680313

1 Introducción

Gestionar los problemas de seguridad en los productos de software es crucial porque dichos problemas, especialmente las vulnerabilidades explotables, pueden afectar exponencialmente a los usuarios finales y requerir más recursos para resolverlos si se descubren en una etapa posterior. En línea con el concepto Shift-Left, que aboga por la detección temprana de problemas en el software [71], trabajos previos [33] argumentan que la revisión manual del código es un enfoque adoptado por numerosos proyectos de software para identificar y mitigar problemas antes de fusionar nuevos cambios de código en la base de código existente. A diferencia de la práctica tradicional, la revisión de código moderna carece de requisitos concretos [31] y puede requerir múltiples iteraciones para completarse. El esfuerzo que los revisores, que normalmente tienen muchas limitaciones de recursos, deben invertir plantea un desafío persistente para las revisiones de código seguro (es decir, encontrar problemas de seguridad en el código revisado). De hecho, identificar problemas que podrían contribuir a las vulnerabilidades a menudo requiere conocimientos de seguridad y una inspección meticulosa por parte de los revisores [33], y por lo tanto, cuarenta los desafíos.

Aunque estudios previos de revisión de código [51, 56, 61] sugirieron que las herramientas de análisis estático podrían ayudar a los revisores a identificar problemas relacionados con los estilos de codificación, ninguno ha explorado su eficacia como herramientas de prueba de seguridad de aplicaciones estáticas (SAST, en adelante). Por el contrario, estudios actuales sobre SAST [28, 29, 49, 55, 63] han investigado su eficacia basándose en las versiones de software publicadas que contienen Vulnerabilidades explotables. Los trabajos existentes no han examinado las herramientas con vulnerabilidades que aparecen gradualmente durante el proceso de desarrollo de software. En las revisiones de código, los cambios suelen ser pequeños [58] y contienen problemas de seguridad difíciles de identificar y fáciles de pasar por alto [33]. Además, una vulnerabilidad completa puede requerir la contribución de cambios de código de múltiples confirmaciones de código [45]. Con los diferentes contextos y la naturaleza de las revisiones de código , no está claro hasta qué punto las SAST pueden ser rentables.

Ayudar a los revisores a detectar problemas de seguridad. Hasta donde sabemos , trabajos previos no han investigado la efectividad de los SAST mediante confirmaciones de código ni para revisiones de código.

Investigamos la capacidad de los SAST para soportar revisiones de código seguro al abordar tres aspectos clave: 1 efectividad de los SAST en confirmaciones de código vulnerable, 2 beneficios potenciales de las revisiones de código soportadas por SAST y 3 tiempo de espera asociado. Nuestro estudio empírico utiliza 815 confirmaciones que contribuyen a vulnerabilidades (VCC) del mundo real en 92 proyectos de software de C y C++. Estas VCC involucran 1,060 funciones vulnerables, que contribuyen a 319 vulnerabilidades explotables. Con este conjunto de datos, estudiamos cinco SAST de C y C++ comúnmente usados [49], que emplean diversas técnicas de análisis estático para detectar vulnerabilidades, es decir, Cppcheck, CodeChecker, CodeQL, Flawnder e Infer. Para medir la efectividad de los SAST, analizamos sobre cuántas VCC pueden advertir las herramientas. Para aprovechar los beneficios potenciales de las revisiones de código con el apoyo de SAST, analizamos cuántas funciones vulnerables se pueden identificar en un esfuerzo limitado de revisión de código cuando las funciones modificadas se priorizan según la información de las advertencias de SAST [44] (en adelante, priorización basada en advertencias). Para ello, medimos la exactitud mediante la precisión y la recuperación en un esfuerzo de revisión fijo (es decir, el 25 % de las líneas de código en las funciones modificadas) y la Falsa Alarma Inicial (IFA), el porcentaje de líneas de código en las funciones advertidas hasta la aparición de las primeras funciones vulnerables. Además, medimos el tiempo que cada herramienta tarda en analizar las VCC.

Nuestros resultados muestran que un solo SAST puede generar advertencias en los cambios vulnerables (es decir, archivos modificados y funciones modificadas) de más de la mitad de las VCC. En concreto, Flawnder puede generar advertencias en al menos una función vulnerable para el 52% y en un archivo vulnerable para el 89% de 815 VCC. La combinación de advertencias de múltiples SAST puede lograr una mayor tasa de detección de vulnerabilidades, es decir, del 78% a nivel de función.

Con respecto a las revisiones de código, priorizar las funciones modificadas utilizando las advertencias de los SAST durante las revisiones de código puede mejorar el descubrimiento de cambios vulnerables al aumentar hasta un 12% de precisión y un 5,6% de recuperación y reducir hasta un 13% de IFA al 25% del esfuerzo de revisión de código. Por último, el tiempo de cálculo de los SAST debe ajustarse al período de espera de la revisión de código, ya que el tiempo de cálculo promedio en los VCC seleccionados varía de 20 segundos a 45 minutos, con un aumento notable cuando el tamaño de los proyectos alcanza los 50k-100k LOC. Sin embargo, los SAST aún necesitan una mejora porque al menos el 76% de las advertencias son irrelevantes para la vulnerabilidad en los VCC correspondientes. Además, el 22% de los VCC (10% de las vulnerabilidades explotables) no reciben ninguna advertencia de ninguna de las cinco herramientas estudiadas.

Novedad y contribución: Hasta donde sabemos, este artículo es el primero en:

- Investigar la efectividad de los SAST de C y C++ en las confirmaciones de código, que están sujetas a revisiones de código, que introdujeron vulnerabilidades del mundo real en 92 proyectos de diversos dominios de aplicación, cubriendo ocho tipos de vulnerabilidades de CWE •
- Investigar los beneficios potenciales de la priorización basada en advertencias para la identificación de vulnerabilidades en las funciones modificadas de los VCC durante las revisiones de código • Proporcionar un marco de evaluación comparativa automatizado, versátil y extensible para la evaluación de SAST utilizando conjuntos de datos de confirmación de código de fuente abierta en varios lenguajes de programación [35]
- Publicar un conjunto de datos anotado de 1064 archivos vulnerables y 1060 funciones vulnerables en 815 confirmaciones [36]

Organización del trabajo: El resto de este trabajo se organiza de la siguiente manera: la Sección 2 explica los antecedentes de este estudio. La Sección 3 presenta el diseño del estudio. La Sección 4 presenta los enfoques del estudio y los resultados de cada pregunta de investigación. La Sección 5 analiza los hallazgos. La sección 6 informa sobre las amenazas a la validez. La sección 7 analiza trabajos relacionados . Finalmente, la sección 8 presenta la conclusión.

2 Antecedentes y definiciones

En esta sección, describimos brevemente los conceptos y términos clave utilizados en este documento y analizamos las limitaciones y los desafíos.

La revisión segura de código se refiere a una práctica de revisión de código que se centra en los aspectos de seguridad del sistema de software. Esta práctica se ajusta al concepto de desplazamiento a la izquierda, que promueve el control de calidad temprano en el desarrollo de software [71] para reducir el impacto de problemas de seguridad no detectados. Durante el ciclo de desarrollo, los desarrolladores envían una confirmación de código que contiene pequeños cambios [58] para que otros desarrolladores la revisen antes de integrarla en el código base. Los revisores pueden inspeccionar los cambios en el código y proporcionar retroalimentación sobre diversos aspectos, como la seguridad [64], la funcionalidad y la mantenibilidad [31, 50]. Los autores pueden modificar el envío para abordar la retroalimentación de los revisores. Estas actividades continúan hasta que el código... Los cambios se aceptan y se integran en el código base. Sin embargo, a pesar de las prácticas generalizadas de revisión de código, identificar problemas de seguridad puede ser un desafío para los revisores. Estudios recientes sobre revisión segura de código [33, 34] destacaron que identificar manualmente problemas de seguridad en el código es difícil porque los revisores pueden no tener los conocimientos necesarios ni la suficiente conciencia de posibles problemas.

La confirmación de contribución a la vulnerabilidad (VCC) es un conjunto de cambios de código que abarca el código modificado que contribuyó a una vulnerabilidad. Representa el objeto del proceso de revisión de código, cuyo objetivo es descubrir las vulnerabilidades durante el ciclo de desarrollo. Iannone et al. [45] realizaron un estudio a gran escala sobre el ciclo de vida de las vulnerabilidades, revelando que una vulnerabilidad real puede estar compuesta por múltiples confirmaciones a lo largo del tiempo. Por ejemplo, una inyección SQL requiere un promedio de 18 VCC. Se puede inferir que una vulnerabilidad puede prevenirse si los problemas en la VCC son identificados por los revisores y corregidos por los desarrolladores durante la revisión de código.

La herramienta de prueba de seguridad de aplicaciones estáticas (SAST) es una herramienta de garantía de calidad que identifica ciertos problemas relacionados con la seguridad en el código fuente utilizando el conjunto predefinido de reglas sin ejecutar el programa . Consideramos SAST que emplean dos técnicas de análisis generales [49]: 1 basada en sintáctica y 2 basada en semántica. Si bien muchos estudios [29, 39–41, 48, 49, 57, 60] evaluaron empíricamente las SAST en la identificación de problemas de seguridad, descuidan el contexto de las revisiones de código y los VCC. Por ejemplo, Lipp et al. [49] evaluaron la efectividad de las herramientas con un conjunto de datos parcialmente sintéticos, y un estudio reciente de Li et al. [48] evaluó las herramientas en las versiones vulnerables de programas del mundo real. Las diferencias clave entre nuestro trabajo y los estudios recientes de SAST se muestran en la Tabla 1. Si bien estudios anteriores relacionados con revisiones de código [51, 56, 61] investigaron las capacidades de los SAST para ayudar a los revisores a identificar estilos o patrones de codificación, aún existen lagunas de conocimiento sobre las capacidades de los SAST para

Tabla 1: Diferencias clave entre este trabajo y el reciente
Estudios SAST centrados en la identificación de vulnerabilidades.

Estudiar	Lang.	Asunto Proyectos	CVE	CWE Grupos	Ejecutivo. Tiempo	Confirmaciones de código
Li y otros [48]	Java	63	165	7		-
Lipp y otros [49]	C/C++	9	192	5		-
Nuestro estudio	C/C++	92	319	8		815 VCC

Identificación de vulnerabilidades. En concreto, se han abordado los siguientes aspectos: no se han incorporado a los estudios existentes: 1 el contexto de cambios de código, por ejemplo, archivos y funciones modificados, 2 los beneficios de revisiones de código respaldadas por advertencias SAST y 3 la cantidad de tiempo que los revisores deben esperar para obtener los resultados del SAST.

3 Diseño del estudio

Para cerrar las brechas de conocimiento del código seguro respaldado por SAST En las revisiones, estudiamos los SAST utilizando las confirmaciones que contribuyen a las vulnerabilidades del mundo real. Evaluar los SAST con este conjunto de datos ayudará. Los investigadores y profesionales comprenden mejor las capacidades de Las herramientas para ayudar a los revisores a identificar posibles vulnerabilidades durante el proceso de revisión de código. También investigamos los SAST. tiempo de computación.

A continuación, presentamos nuestras preguntas de investigación y el estudio. Diseño. La descripción general del proceso de estudio se muestra en la Figura 1.

3.1 Preguntas de investigación

En este estudio, nos propusimos responder las siguientes preguntas de investigación.

RQ1 Eficacia: ¿Con qué eficacia pueden los SAST detectar vulnerabilidades en confirmaciones que contribuyen a ellas?

Motivación: Trabajos previos [29, 39–41, 57, 60] estudiaron SAST con conjuntos de datos de referencia sintéticos o ciertas versiones de programas que Contienen vulnerabilidades completas. Es posible que los conjuntos de datos existentes no representen las vulnerabilidades que aparecen gradualmente a partir de los cambios de código. Durante el proceso de desarrollo. Investigación de SAST mediante VCC. Debería ampliar la comprensión de la capacidad de la herramienta para detectar vulnerabilidades en este contexto.

Enfoque: ejecutamos SAST en los VCC seleccionados y determinamos El número de VCC que los SAST pueden generar alertas sobre cambios de código vulnerables. En este trabajo, consideramos el alcance del código. cambios en los VCC a nivel de función y de le. También analizamos los tipos de vulnerabilidades en VCC que los SAST pueden detectar.

RQ2 Priorización basada en advertencias: ¿Cómo podemos utilizar SAST? ¿Advertencias para priorizar los cambios para las revisiones de código seguro?

Motivación: Los revisores tienen una capacidad de revisión limitada [33, 61]. Trabajos previos [30, 32, 43] argumentaron que las advertencias de SAST pueden ayudar a reducir El revisor Muske y Serebrenik [53] informaron que los desarrolladores priorizan las advertencias de SAST a las que deben prestar atención. Por lo tanto, nuestra hipótesis es que la priorización basada en advertencias de cambios Las funciones podrían reducir el esfuerzo de revisión.

Enfoque: Nos propusimos investigar si, dentro de un esfuerzo de revisión limitado, la priorización basada en advertencias puede ayudar a los revisores a identificar funciones más vulnerables. Medimos la precisión de la identificación de funciones vulnerables priorizadas por las advertencias SAST dentro de un esfuerzo de revisión fijo (es decir, 100% líneas de código en funciones modificadas).

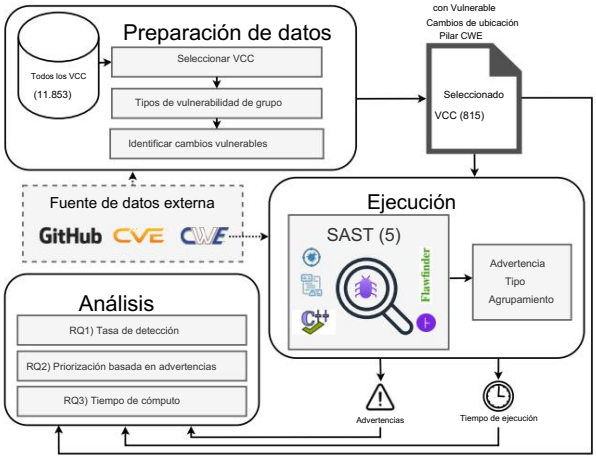


Figura 1: Descripción general de nuestro enfoque de estudio.

RQ3 Tiempo de espera: ¿Cuánto tiempo de cálculo se requiere?

Motivación: Kudrjavets et al. [47] sugirieron que las pruebas automatizadas debe ejecutarse durante el período no productivo de revisiones de código. Para incorporar SAST sin retrasar las revisiones de código, el El tiempo de cálculo debería ajustarse a este período de inactividad. Sin embargo, el tiempo de cálculo de las SAST de C/C++ en VCC sigue sin explorarse.

Enfoque: Medimos el tiempo de cálculo desde que se ejecuta la herramienta comienza a analizar una confirmación hasta que se producen las advertencias. Además , analizamos el tiempo de cálculo por las líneas de Código para comprender si el tiempo depende del tamaño del sistema [48].

3.2 Preparación de datos

Dado que los conjuntos de datos de referencia de estudios anteriores [40, 49] se obtienen de versiones publicadas, es posible que no representen la naturaleza de cambios de código en el desarrollo de software y revisión de código procesos donde los cambios son pequeños y pueden estar mezclados con otros cambios. Por lo tanto, un nuevo conjunto de datos de cambios de código que contribuyen a las vulnerabilidades explotables es más adecuado. Describimos los datos Proceso de preparación para nuestro estudio a continuación.

3.2.1 Conjunto de datos. Para crear el conjunto de datos, comenzamos con la vulnerabilidad. Conjunto de datos de confirmaciones contribuyentes (VCC) proporcionado por Iannone et al. [45]. Este conjunto de datos es adecuado para nuestro estudio porque recopiló confirmaciones de código del mundo real que se han identificado como contribuyentes a vulnerabilidades explotables. Los conjuntos de datos de VCC se recopilaron de Proyectos de GitHub asociados con vulnerabilidades en Common Base de datos de vulnerabilidades y exposiciones (CVE) [3]: la colección de vulnerabilidades explotables en sistemas de software que son de dominio público reportado por la comunidad. Dado un commit de xing registrado en Se identificó un CVE y un VCC utilizando el algoritmo SZZ [62]. Cada VCC incluye metainformación, es decir, la URL del repositorio, la confirmación Número de identificación (Commit SHA), vulnerabilidad relacionada (CVE) número) y el tipo de vulnerabilidad relacionada (elemento CWE). Debería ser Se observó que el conjunto de datos VCC no proporciona los cambios reales en cada confirmación, por ejemplo, archivos vulnerables o funciones vulnerables.

3.2.2 Selección de VCC. Aunque el conjunto de datos VCC de Iannone et al. [45] es considerable, algunos VCC no pueden analizarse mediante SAST debido a

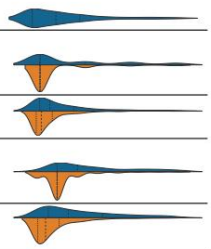
Tabla 2: Tipos de vulnerabilidad de VCC seleccionados. Un solo VCC Puede estar relacionado con más de un pilar de CWE.

Nombre corto	Pilar CWE	#VCC
Control de acceso	CWE-284 Control de acceso inadecuado†	38
Resource Ctrl CWE-664	Control inadecuado de un recurso a través de su Vida	538
Cal incorrecta	CWE-682 Cálculo incorrecto	78
Flujo de control CWE-691	Gestión insuficiente del flujo de control	56
Falla del mecanismo de protección	Protect Mech CWE-693†	13
Comprobación del estado	CWE-703 Verificación o manejo inadecuado de elementos excepcionales Condiciones	20
Neutralización CWE-707	Neutralización inadecuada	56
El estándar de	CWE-710 Adherencia inadecuada a los estándares de codificación†	45

codificación † indica los pilares adicionales de [49]

Tabla 3: Demografía de los cambios de código en VCC seleccionados. Los valores medios y medianos comparan las tendencias centrales. entre cada par de elementos.

Descripción	Distribución media- mediana	
Vulnerabilidades CVE (319)		
Los VCC contribuyeron a la CVE	2.5	2
VCC con archivos vulnerables (815)		
Archivos modificados (9,851)	12	3
Archivos vulnerables (1.064)	1	1
LOC en archivos modificados	9,956	2.492
LOC en archivos vulnerables	1,820	1.034
VCC con funciones vulnerables (697)		
Funciones modificadas (34.541)	38.9	6
Funciones vulnerables (1.060)	1.5	1
LOC en funciones modificadas	3.304	531
LOC en funciones vulnerables	221	114



lenguajes de programación incompatibles o problemas de compilación; otros Puede que no tenga suficiente información para nuestro análisis (por ejemplo, falta información) Tipo de vulnerabilidad). Seleccionamos las VCC según los siguientes criterios:

(1) De proyectos C/C++: Seleccionamos VCC de los proyectos implementado en C o C++ porque estos lenguajes permiten programas para interactuar con operaciones de bajo nivel que pueden conducir a vulnerabilidades explotables [67]. Para determinar esto, obtuvimos el lenguaje de programación principal del proyecto de la API de GitHub.

(2) Tener un tipo de vulnerabilidad asignado: Ya que analizaremos El tipo de vulnerabilidad que los SAST pueden detectar, debemos asegurarnos que los VCC seleccionados tengan suficiente información sobre vulnerabilidades. Al CVE asociado de un VCC generalmente se le asigna una vulnerabilidad Escriba en la taxonomía de Enumeración de Debilidades Comunes (CWE). Por lo tanto, seleccionamos VCC con al menos un elemento CWE asignado al CVE asociado.

(3) Compilable: Algunos SAST requieren la información del proceso de compilación. Por lo tanto, los VCC seleccionados deben ser compilables . Sin embargo, diversos procesos de compilación y dependencias en Muchos proyectos C/C++ en el conjunto de datos VCC plantean desafíos para Pruebas de compilación automatizadas, lo que aumenta las posibilidades de compilación. Fallo. Compilar manualmente cada VCC también es inviable debido a Debido a la gran cantidad de VCC, para mitigar el problema, primero seleccionamos los proyectos que utilizan el proceso de compilación estándar, es decir, GNU Autotools tal como se ha utilizado en muchos proyectos de C y C++ (más del 38%) en el conjunto de datos original de VCC. Luego, seleccionamos VCC que se puede compilar con éxito siguiendo las instrucciones en el documento del proyecto.

Con base en el primer criterio obtuvimos 5.354 VCC en 341 proyectos del conjunto de datos VCC. Luego, utilizando el segundo criterio para el Obtuvimos VCC, identificamos 5.015 VCC vinculados a 1.654 CVE con Al menos un elemento CWE. Finalmente, 1.057 VCC con 371 CVE de 92 Los proyectos pasaron la prueba de compilación en nuestro tercer criterio.

3.2.3 Agrupación por tipo de vulnerabilidad. Utilizamos información de vulnerabilidad en un elemento CWE para analizar el tipo de vulnerabilidad en este estudio. Los elementos CWE representan ciertas debilidades que pueden conducir a vulnerabilidades en un sistema de software. Sin embargo, si bien los elementos CWE permiten análisis detallados de las vulnerabilidades individuales, no incorporan Las vulnerabilidades similares en el conjunto de datos VCC que se asignan con Diferentes elementos de CWE. Por lo tanto, agrupamos los CWE asignados Elementos de VCC a un pilar CWE, que es el nivel superior del CWE jerarquía, que representa categorías abstractas de los elementos de CWE [40]. Tenga en cuenta que no utilizamos otros métodos de agrupación CWE como Categorías CWE [15] porque solo cubren 400 elementos CWE de más de 900 elementos de CWE en los diez pilares de la jerarquía de CWE. Para identificar el pilar asociado de un elemento CWE, rastreamos el CWE El elemento padre a la raíz del árbol. La Figura 2 ilustra nuestro enfoque de agrupación. Por ejemplo, los elementos CWE usan elementos no inicializados. Variable (CWE-457) [16] y desreferencia de puntero caducado (CWE-825) [17] se agruparán en el Control Inadecuado de un Pilar Recurso a través de su vida útil (CWE-664) [18].

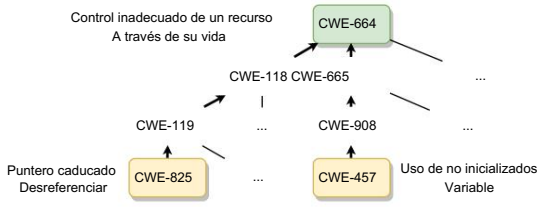


Figura 2: Un ejemplo ilustrativo de agrupación de elementos CWE (CWE-825 y CWE-457) al pilar CWE (CWE-664).

En total, nuestro conjunto de datos cubre ocho pilares de CWE, lo que supone más diverso que el conjunto de datos de referencia del trabajo anterior [49] (Tabla 1). La Tabla 2 muestra el número de VCC en cada uno de los pilares del CWE.

3.2.4 Identificación de la ubicación de cambios vulnerables en los VCC. Dado que El objetivo es investigar si los SAST pueden proporcionar advertencias a los ubicaciones correctas del código vulnerable, debemos identificar el alcance de Código vulnerable en una VCC. Consideramos dos niveles de granularidad, es decir, archivos modificados y funciones modificadas en VCC. No identificamos Los cambios vulnerables a nivel de línea debido a líneas modificadas en Las confirmaciones de xing pueden no corresponder directamente a líneas vulnerables en VCC [59]; por lo tanto, la confiabilidad puede verse reducida. Sin embargo, También se evaluaron manualmente las advertencias de SAST para comprender su significado real. relevancia para los VCC en análisis posteriores (véase Sección 5.1).

Utilizando una intuición similar a la del trabajo anterior [62], consideramos que un archivo modificado (o una función modificada) es vulnerable si fue Modificado en las confirmaciones de vulnerabilidad asociadas. Siguiendo el enfoque de [45], obtuvimos las confirmaciones de vulnerabilidad de Hipervínculos de GitHub en los registros CVE de las VCC. Los archivos modificados En los VCC se consideran vulnerables (vulnerables en adelante) Cuando estos archivos también se cambiaron en las confirmaciones de Xing. Para

Para las funciones modificadas, utilizamos el paquete lizard [27] para identificarlas en un VCC y las confirmaciones de Xing asociadas. Las funciones modificadas en los VCC se consideran vulnerables (en adelante, funciones vulnerables) cuando también se modificaron en las confirmaciones de Xing.

Localizamos con éxito cambios de código vulnerables en 815 VCC donde al menos un archivo de código fuente está modificado en ambos VCC y xing commits. Estas VCC contribuyeron a 319 vulnerabilidades explotables en la base de datos CVE. La Tabla 3 muestra la demografía de las VCC seleccionadas, destacando que los cambios vulnerables pueden ser difíciles de identificar porque suelen ser pequeños en comparación con todos los cambios de código en cada VCC. Utilizamos estas VCC en nuestro análisis.

3.3 Ejecución

Esta sección describe los SAST estudiados, la configuración del experimento y la agrupación de tipos de advertencia.

3.3.1 Herramientas estudiadas. Nuestro objetivo es analizar las SAST disponibles para proyectos de código abierto (OSS) en C y C++. En concreto, seleccionamos las SAST que: 1) están disponibles de forma gratuita, 2) se pueden ejecutar en la interfaz de línea de comandos (CLI), ya que necesitamos automatizar el proceso de compilación, y 3) reciben mantenimiento activo de los desarrolladores (actualizado en los últimos 12 meses).

Para seleccionar las SAST, primero obtenemos 53 SAST de estudios previos [49, 54, 63] y del Sistema de Métricas de Garantía de Software y Evaluación de Herramientas (SAMATE) del NIST [21] como candidatas.[1] Luego, examinamos la documentación de cada herramienta para comprender sus especificaciones. De la lista inicial, eliminamos 12 SAST que no son compatibles con los lenguajes C y C++. Posteriormente, excluimos 26 SAST comerciales, dos SAST que no operan en la interfaz de línea de comandos y cuatro SAST que no reciben mantenimiento activo. También omitimos cuatro SAST, incluyendo aquellos que operan en un servidor externo, son extensiones del compilador (no emplean reglas adicionales) o carecen de información sobre las advertencias disponibles. Finalmente, tres herramientas semánticas (CodeChecker[23], CodeQL [24] e Infer [20]); una herramienta sintáctica (Flawnder [19]); y una herramienta híbrida (Cppcheck [25]) cumplen nuestros criterios.

3.3.2 Configuración del experimento. En este estudio, utilizamos las últimas versiones estables de los SAST seleccionados: Cppcheck v2.10, CodeChecker v6.22.2, CodeQL v2.13.3, Flawnder v2.0.19 e Infer v1.10. Para cada herramienta, utilizamos la configuración recomendada por sus desarrolladores para obtener resultados fiables (como también sugiere [52]). En particular, para Flawnder y Cppcheck, utilizamos las reglas fácilmente habilitadas. Para CodeQL, utilizamos el conjunto de consultas LGTM [13], que contiene el mayor número de reglas. Para CodeChecker e Infer, utilizamos la configuración recomendada. Cabe destacar que no habilitamos todas las reglas para CodeChecker e Infer, ya que los desarrolladores de las herramientas lo desaconsejan.

Pipeline: Desarrollamos un marco automatizado para facilitar la ejecución de SAST. Cada VCC se descarga y prepara antes de ejecutar SAST. Los pasos de preparación varían según el SAST. Para los SAST que requieren compilación (p. ej., CodeChecker, CodeQL e Infer), actualizamos Makefile ejecutando el script automatizado que los proyectos han preparado para la compilación, es decir, autogen.sh, bootstrap.sh y build.sh. Si ninguno de los scripts está presente, Use el comando autoreconf para generar los archivos necesarios. Luego, ejecute el archivo de configuración (configure). Tras la preparación, ejecute SAST y recopile las advertencias generadas para su posterior análisis. Tenga en cuenta que el tiempo de preparación también se incluye en el tiempo de cálculo.

La canalización opera en contenedores Docker aislados para controlar el entorno y los recursos de cada experimento (es decir, ejecuta automáticamente SAST en un conjunto seleccionado de confirmaciones de código objetivo). Almacena las advertencias generadas en el equipo host para facilitar el acceso y administra una base de datos centralizada que recopila el estado de ejecución, el número de advertencias y las marcas de tiempo para su posterior análisis.

Infraestructura: Para ejecutar nuestros experimentos, utilizamos una máquina virtual con una CPU virtual de 32 núcleos y 288 GB de memoria. Ejecutamos cada uno de los SAST estudiados en un contenedor Docker Ubuntu 22.04 con un CPU virtual de 4 núcleos. Debido a la limitación de tiempo, finalizamos una ejecución si esta continúa durante más de cinco horas en cualquier VCC.

3.3.3 Agrupación por tipo de advertencia. Dado que nuestro objetivo es examinar si un SAST proporciona advertencias que coinciden con el tipo de vulnerabilidad de una VCC, mapeamos y agrupamos las advertencias de los SAST estudiados en pilares CWE. Para Cppcheck, CodeQL y Flawnder, cada advertencia tiene asignado un elemento CWE. Por lo tanto, utilizamos el mismo enfoque descrito en la agrupación de vulnerabilidades para agrupar los elementos CWE en pilares CWE (véase la Sección 3.2.3).

Dos herramientas SAST, CodeChecker e Infer, no asignan elementos de CWE en las advertencias. Por lo tanto, las advertencias generadas por estas herramientas deben asignarse a los pilares de CWE más relevantes. Utilizamos la asignación entre advertencias e elementos de CWE proporcionada por Lipp et al. [49]. Sin embargo, el mapeo [49] no cubre todos los tipos de advertencia en nuestros VCC, que contienen vulnerabilidades más diversas. Consultamos los documentos oficiales [14, 22, 26] para mapear las advertencias restantes sin elementos CWE. Por ejemplo, asignamos el elemento CWE Reachable Assertion (CWE-617) a la advertencia bugprone-assert-side-effect3 de CodeChecker. De las 178 advertencias que necesitaban el nuevo mapeo, el 5% no se puede vincular a ningún elemento CWE.4 El proceso de mapeo lo realiza principalmente el primer autor. Para garantizar la precisión del proceso de mapeo, el tercer autor, que tiene más de 10 años de experiencia en pruebas de seguridad de software, se unió al primer autor para establecer el marco fundacional y ayudó.

con advertencias ambiguas encontradas a lo largo del proceso. Finalmente, los elementos de CWE se agrupan en el pilar CWE, como se describe en la Sección 3.2.3.

4 Análisis y resultados Informamos

los resultados de nuestro análisis y respondemos las preguntas de investigación en esta sección.

4.1 Efectividad de detección (RQ1)

Enfoque: Para responder a nuestra RQ1: ¿Con qué eficacia pueden los SAST detectar vulnerabilidades en confirmaciones que contribuyen a vulnerabilidades?, medimos cuántos VCC en los que los SAST pueden producir advertencias sobre los cambios vulnerables y analizamos los tipos de vulnerabilidades.
En concreto, ejecutamos los SAST estudiados en los 815 seleccionados

1La lista exhaustiva de SAST evaluados se incluye en el paquete de datos 2https://codechecker.readthedocs.io/en/latest/analyzer/user_guide/#enable-all y https://github.com/facebook/infer/issues/1114#issuecomment-506284374

3https://clang.llvm.org/extra/clang-tidy/checks/bugprone/assert-side-efecto.html

Incluimos los mapeos completos en nuestro paquete de datos

Tabla 4: Nombre y descripción de los escenarios de detección en dos niveles de cambio.

Nombre del escenario por nivel de cambio		Descripción
Archivo	Función	
S1:1F-Cualquiera	S5:1Fn-Cualquiera	Al menos un cambio vulnerable en el VCC recibe advertencias con cualquier tipo.
S2:1F-Lo mismo	S6:1Fn-Mismo	Al menos un cambio vulnerable en el VCC recibe advertencias con el mismo tipo de vulnerabilidad que el VCC.
S3:allF-Cualquiera	S7:allFn-Cualquiera	Todos los cambios vulnerables en el VCC reciben advertencias con cualquier tipo.
S4:allF-Igual	S8:allFn-Igual	Todos los cambios vulnerables en el VCC reciben advertencias con el mismo tipo de vulnerabilidad que el VCC.

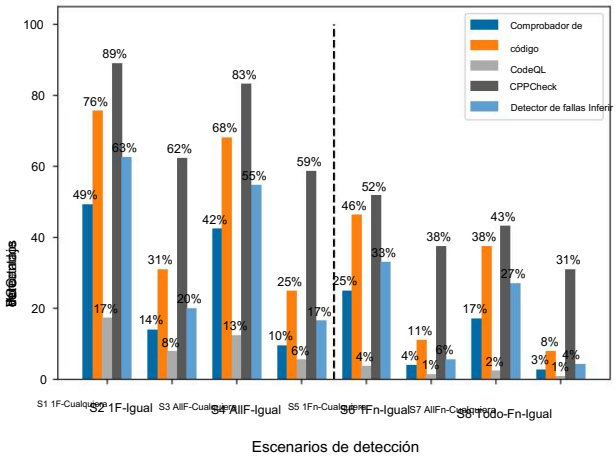


Figura 3: Porcentajes de VCC que las herramientas pueden detectar en diferentes escenarios.

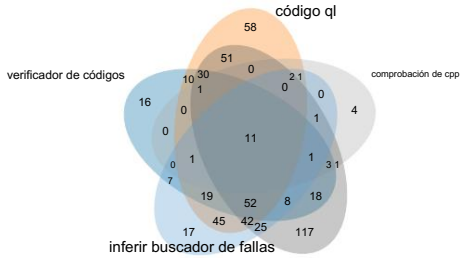


Figura 4: Un diagrama de Venn que muestra los VCC para los cuales las herramientas pueden producir advertencias en las funciones vulnerables (S5:1Fn-Cualquiera)

> Hallazgo (tasa de detección): Flawnder puede producir advertencias en los cambios vulnerables en el 89% de los VCC a nivel de nivel y en el 52% de los VCC a nivel de función.

Investigamos más a fondo si la combinación de SAST puede mejorar la tasa de detección. Es posible que diferentes SAST puedan detectar diferentes tipos de vulnerabilidad. Examinamos la tasa de detección de múltiples SAST a nivel de función (S5: 1Fn-Any) como Lipp et al. [49] señalaron que el nivel de función es una granularidad apropiada para analizar las advertencias de SAST. La Figura 4 muestra que múltiples herramientas (es decir, CodeChecker, CodeQL, Flawnder e Infer) pueden detectar comúnmente 52 VCC, que representan solo el 6%. Esto sugiere que las herramientas no detectan regularmente las mismas VCC. En ese sentido, cuando se consideran todas las advertencias de las cinco herramientas, detectan colectivamente 541 VCC (78%), lo que aumenta sustancialmente en 26 puntos porcentuales en comparación con la tasa de detección en S5:1Fn-Any de una sola herramienta con la tasa de detección más alta (es decir, Flawnder). Esto destaca que la combinación de estas herramientas mejora su efectividad en la identificación de un mayor número de VCC.

> Búsqueda (combinación de herramientas): la combinación de múltiples SAST puede aumentar la efectividad en la identificación de vulnerabilidades en VCC en un 26 %.

VCC y examinar si las herramientas generan advertencias para los archivos y funciones vulnerables en los VCC. Similar a

En trabajos anteriores [48, 49], para cada granularidad (es decir, nivel de función o le), consideramos la tasa de detección en función de cuatro escenarios (ver Tabla 4).

Resultados: Consideramos los resultados en tres aspectos: 1) tasa de detección, 2) vulnerabilidades detectadas por tipo y 3) vulnerabilidades no detectadas.

Tasa de detección: La Figura 3 muestra los porcentajes de VCC que un SAST puede detectar en cada escenario. Flawnder generó advertencias en archivos vulnerables y funciones del mayor número de VCC. Por lo tanto, ofrece potencialmente la menor cantidad de falsos negativos, es decir, un número bajo de archivos y funciones vulnerables que no reciben una respuesta. Advertencia. En concreto, Flawnder genera advertencias en al menos un archivo vulnerable (S1:1F-Any) para el 89 % de las VCC. A nivel de función, que tiene un alcance menor de cambios de código, Flawnder genera advertencias en al menos una función vulnerable (S5:1Fn-Any) en el 52 % de las VCC. Por el contrario, Cppcheck generó advertencias en archivos y funciones vulnerables del menor número de VCC en los ocho escenarios.

Vulnerabilidades detectadas por tipo: Los SAST pueden generar advertencias en VCC de casi todos los tipos de vulnerabilidad en nuestro conjunto de datos. La Tabla 5 muestra el número y el porcentaje de VCC en los que los SAST generan advertencias con el mismo tipo de vulnerabilidad en al menos una función vulnerable (S6: 1Fn-Same) y en todas las funciones vulnerables (S8: allFn-Same) por pilares de CWE.

- Flawnder advierte a la mayoría de los VCC relacionados con Resource Ctrl (CWE -664) (38%), p. ej., posible desbordamiento de búfer [10] y Neutralización (CWE-707) (16%), p. ej., posible inyección de comandos debido a una validación de entrada insuficiente [5].

- CodeQL advierte a la mayoría de los VCC relacionados con Access Ctrl (CWE-284) (3%), p. ej., posible fuga de información [12] ; Incorrect Cal (CWE- 682) (12%), p. ej., posible DoS causado por cálculos incorrectos [4] ; y Control Flow (CWE-691) (4%), p. ej., bucle infinito causado por el tipo de datos incorrecto de la variable de bucle [6].

Tabla 5: Número de VCC que reciben advertencias en todas las funciones vulnerables (S8:allFn-Same) o al menos en una función vulnerable (S6:1Fn-Same) y porcentajes por pilares de CWE. El primer y el segundo número corresponden a S8 y S6, respectivamente.

Pilar CWE	Comprobador de código	CódigoQL	Comprobación de Cpp	Flawnder	Inferir
Control de acceso CWE-284	-	1/1 (3%/3%)	-	-	-
Control de recursos CWE-664	17/44 (3%/4%)	43/63 (8%/12%)	4/8 (1%/1%)	205/249 (38%/46%)	24/33 (4%/6%)
Cal incorrecta CWE-682	-	9/11 (12%/14%)	1/1 (1%/1%)	2/2 (2%/2%)	-
Flujo de control CWE-691	-	2/2 (4%/4%)	-	-	-
Comprobación del estado CWE-703	4/4 (20%/20%)	-	-	-	-
Neutralización CWE-707	3/3 (5%/5%)	-	-	9/12 (16%/21%)	-
Estándar de codificación CWE-710	8/12 (16%/27%)	2/2 (4%/4%)	1/1 (2%/2%)	-	6/7 (13%/16%)

- CodeChecker advierte a la mayoría de los VCC relacionados con Cond Check (CWE-703) (20%), por ejemplo, desreferencias de puntero nulo causadas por el orden incorrecto de argumentos [11] y Coding Std (CWE-710), (16%), por ejemplo, la declaración if incorrecta [7].

Búsqueda (tipos de vulnerabilidad): CodeQL detecta la mayoría de los tipos de vulnerabilidad en VCC (5 de 8). Solo CodeQL detecta vulnerabilidades de tipo Control de Acceso (CWE-284) y Flujo de Control (CWE-691), mientras que solo CodeChecker detecta vulnerabilidades de tipo Comprobación de Cond (CWE-703).
Flawnder puede detectar la mayoría de los VCC de tipo Resource Ctrl (CWE-664) y Neutralización (CWE-707).

Vulnerabilidades no detectadas: 156 VCC (22%) distribuidos en 33 proyectos (35%) no recibieron ninguna advertencia en las funciones vulnerables (S5:1Fn-Any). En particular, las vulnerabilidades relacionadas con CWE-664, CWE-682, CWE-693, CWE-707 y CWE-710 son los tipos comunes (143 de las 156 VCC) que no fueron detectadas por los SAST estudiados, que representan el 10% de las vulnerabilidades estudiadas. Sorprendentemente, algunas de estas vulnerabilidades no detectadas son sencillas y deberían haber sido identificadas por los SAST. Por ejemplo, el CVE-2018-17294 [9] implica una sobrelectura de búfer debido a una comprobación de longitud faltante en un bucle, desviándose del patrón de codificación regular [2]. Además, las vulnerabilidades relacionadas con Protect Mech (CWE-693) pasaron desapercibidas para los SAST a pesar de la existencia de reglas de apoyo. La observación manual reveló que las reglas actuales son estrictas con respecto a patrones de código específicos de las vulnerabilidades; una pequeña desviación puede hacer que la vulnerabilidad pase desapercibida. Por ejemplo, CVE-2017-13083 [8] permite la ejecución de código externo a través de una conexión insegura. Las consultas CodeQL existentes no detectaron estas vulnerabilidades, ya que utilizaban expresiones regulares o se centraban en funciones OpenSSL, mientras que el código vulnerable verificaba certificados HTTP con la API de Windows.

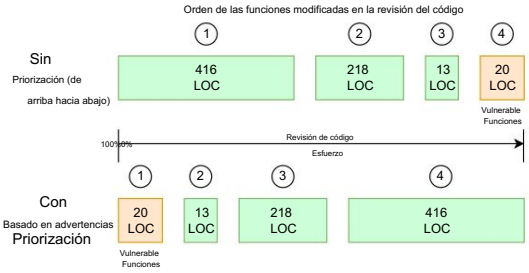


Figura 5: Una ilustración para representar funciones modificadas sin priorización (arriba) y con priorización basada en advertencias (abajo) utilizando un VCC real de libsndle.

Hallazgo (vulnerabilidad no detectada): El 22 % de las VCC no reciben advertencias en las funciones vulnerables. CWE -664, CWE-682, CWE-693, CWE-707 y CWE-710 son los tipos de vulnerabilidad comunes que las herramientas pasan por alto a pesar de la presencia de reglas correspondientes en los SAST.

4.2 Priorización basada en advertencias (RQ2)

Enfoque: Para responder a nuestra RQ2: ¿Cómo podemos usar las advertencias de SAST para priorizar los cambios para las revisiones de código seguro?, analizamos cuántas funciones vulnerables se pueden identificar dentro de un esfuerzo fijo (% líneas de código; %LOC) cuando las funciones modificadas en un VCC se priorizan según las advertencias de un SAST. La Figura 5 ilustra que la priorización basada en advertencias puede afectar el orden de las funciones modificadas en la revisión de código de un VCC.5 Sin una priorización (revisión de los archivos modificados alfabéticamente [1], de arriba hacia abajo según el número de línea en cada archivo), los revisores pueden dejar de revisar los cambios antes de llegar a la función vulnerable, o una función vulnerable puede revisarse al final, lo que requiere un mayor esfuerzo de revisión para encontrar una vulnerabilidad. Por el contrario, cuando una función vulnerable se prioriza mediante advertencias de SAST, puede ahorrar esfuerzo de revisión de código en la identificación de vulnerabilidades. Utilizamos las advertencias del escenario S5 (1Fn-Any) para responder esta pregunta porque las advertencias pueden guiar a los revisores a la hora de priorizar los cambios de código, aunque los tipos difieren de las vulnerabilidades informadas.

Exploramos tres enfoques de priorización basados en la información de advertencia [66, 68]: 1 Cantidad de advertencia (WA): una función modificada con una gran cantidad de advertencias revisadas primero, 2 Densidad de advertencia (WD): una función modificada con una gran cantidad de advertencias por LOC revisadas primero [66], y 3 Gravedad de advertencia (WS): una función modificada con advertencias de alta gravedad6 revisadas primero [68].

Inspirados por estudios previos [44, 70], utilizamos las siguientes métricas Para medir la efectividad de la priorización basada en alertas:

- Falsa alarma inicial (IFA): la proporción de líneas de código en las funciones modificadas que el revisor debe revisar hasta

5https://github.com/libsndle/libsndle/commit/1fc6bb8e0faa130e1071faaf98991ac8e4555310b

Para facilitar el análisis, convertimos los niveles de gravedad de las advertencias de cada herramienta a tres niveles: Alto, Medio y Bajo. Por ejemplo, los tipos de gravedad de las advertencias de CodeQL, Error, Advertencia y Nota, se asignan a Alto, Medio y Bajo, respectivamente.

La tabla de mapeo está incluida en el paquete de datos.

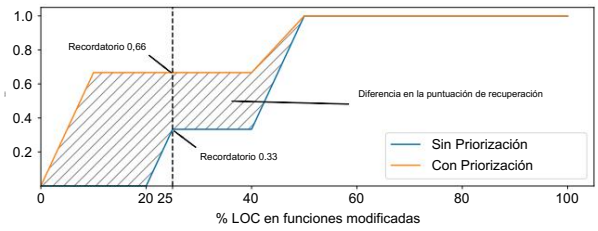


Figura 6: Un ejemplo de Recall@ %LOC con y sin priorización basada en advertencias de un VCC real de Leptonica.

Revisión de la primera función vulnerable. Un IFA alto indica que se requiere mayor esfuerzo para revisar los cambios no vulnerables. • Precisión en el % del LOC: La proporción de funciones vulnerables priorizadas dentro del % del LOC superior en comparación con el número total de funciones modificadas priorizadas dentro del % del LOC superior, es decir, + .

• Recall@ %LOC: La proporción de funciones vulnerables que se priorizan dentro del % LOC superior en comparación con el número total de funciones vulnerables en un VCC, es decir, Yo / y yo y yo+yo .

Establecemos =25% LOC para Precisión@ %LOC y Recuperación@ %LOC porque el IFA de revisar funciones modificadas sin priorización basada en advertencias es, en promedio, del 25% LOC en nuestro conjunto de datos VCC; es decir, los revisores suelen necesitar revisar el 25% del LOC de las funciones modificadas hasta que se revisa la primera función vulnerable. También experimentamos con =30%, 40% y 50% LOC para garantizar la consistencia.

Para cuantificar la mejora de una priorización basada en advertencias mediante un SAST, en comparación con la no priorización, medimos el per- − ganancia de rendimiento = Σ ( ) / Σ [70]. La figura 6 muestra Un ejemplo de un VCC[7] donde las funciones modificadas se priorizan según los niveles de gravedad de las advertencias de CodeChecker. Por ejemplo, con =25%LOC, la priorización basada en advertencias puede mejorar la recuperación en un 100%. (0,66−0,33 / 0,33), en comparación con una no priorización.

Además, utilizamos la prueba de rangos con signo de Wilcoxon para confirmar si las diferencias de rendimiento entre las revisiones de código con y sin soporte de herramientas en los VCC son estadísticamente diferentes.

Dado que comparamos la diferencia de rendimiento entre las cinco herramientas, utilizamos la corrección de Bonferroni para calcular el nuevo umbral de aceptación con un nivel de confianza del 95 %, ya que = (0.05 / 5) = 0,01.

Resultados: La Tabla 6 muestra que el IFA puede reducirse en un 13% cuando las funciones se priorizan según las advertencias de CodeQL. Cuando el esfuerzo de revisión se fija en un 25% del LOC de las funciones modificadas, la priorización basada en advertencias puede aumentar la precisión en un 12% al utilizar advertencias de CodeQL. Las puntuaciones de recuperación también pueden aumentar en un 5,6% al utilizar advertencias de CodeChecker. La Tabla 6 también muestra que la precisión y la recuperación de la priorización basada en advertencias son estadísticamente superiores a las de la no priorización. Sin embargo, también observamos que el rendimiento de algunas priorizaciones basadas en advertencias (p. ej., basadas en la cantidad de advertencias de Flawnder) puede ser inferior al de la no priorización. También comprobamos el rendimiento con otros valores de (es decir, = 30, 40 y 50 % LOC) y los análisis estadísticos confirman que el desempeño de la priorización basada en advertencias es mejor que la no priorización.

Búsqueda (priorización basada en advertencias): En comparación con la priorización sin priorización, la priorización basada en advertencias mediante las advertencias de Cod-eQL ofrece una mejora sustancial en la precisión (12 %) y la falsa alarma inicial (13 %). CodeChecker ofrece una mejora sustancial en la recuperación (5,6 %).

Los diferentes enfoques de priorización pueden producir diferentes ganancias de rendimiento. Como se ve en la Tabla 6, la priorización de funciones modificadas dentro de VCC con densidad de advertencia (WD) generalmente produce la mayor mejora para la recuperación (hasta un 5,6 %) y la IFA (hasta un 13,3 %), independientemente de las SAST. En términos de precisión, WD proporciona una mejora ligeramente menor que la cantidad de advertencia (WA), pero WA normalmente ofrece una puntuación de recuperación más baja y una IFA más alta. Por otro lado, la gravedad de la advertencia (WS), que los desarrolladores suelen utilizar durante el proceso de desarrollo [68], produce la mejora de rendimiento más baja en comparación con otros enfoques de priorización en todas las métricas.

Hallazgo (enfoque de priorización): Priorizar las funciones modificadas mediante la densidad de advertencia (WD) generalmente produce la mayor mejora entre los tres enfoques.

4.3 Tiempo de espera (RQ3)

Enfoque: Para responder a nuestra pregunta 3: ¿Cuánto tiempo de cálculo se requiere?, medimos el tiempo de cálculo de cada experimento, es decir, la ejecución de una herramienta en un VCC. Este período abarca todo el proceso, incluyendo los demás pasos necesarios.

Resultados: La Tabla 7 presenta las estadísticas de tiempo de cálculo. Flawnder fue el que menos tiempo de cálculo requirió, con un promedio de 20 segundos, gracias a su enfoque de análisis sintáctico que omite la compilación. Por otro lado, CodeChecker, CodeQL e Infer, que requieren compilación, tuvieron tiempos de cálculo promedio comparables, de entre 213 y 456 segundos. Sorprendentemente, Cppcheck, que emplea técnicas semánticas y sintácticas sin compilación, tuvo el mayor tiempo de cálculo, con un promedio de 2702 segundos.

Hallazgo (tiempo promedio): El tiempo promedio de cálculo SAST de C/C++ está entre 20 segundos y 45 minutos.
Flawnder es la herramienta más rápida.

El tiempo de cálculo varía según el tamaño del sistema. Medimos el tamaño del sistema (líneas de código; LOC) en cada VCC y examinamos los tiempos de cálculo en todos los VCC. El tiempo de cálculo promedio en sistemas de diferentes tamaños revela que Cppcheck es notablemente más lento (más de 10 minutos) para sistemas con un LOC superior a 10 000-15 000, mientras que CodeQL, Infer y CodeChecker son más lentos para sistemas con un LOC superior a 50 000-100 000. Por el contrario, el tiempo de cálculo de Flawnder se mantiene relativamente constante. El tiempo de cálculo prolongado de Cppcheck, especialmente para sistemas con un LOC superior a 10 000-15 000, puede deberse a su secuencial

7https://github.com/danbloomberg/leptonica/commit/cef50b2cb3a8be397d81d4d32388a3918374e1b5

Tabla 6: Porcentaje de diferencias de desempeño cuando las funciones modificadas se clasifican con priorización basada en advertencias en Escenario S5 (1Fn-Cualquiera) al 25% del LOC en funciones modificadas.

Herramienta	Prec@25% (WA)	Prec@25% (WD)	Prec@25% (WS)	Recordatorio al 25% (WA)	Recordatorio al 25% (WD)	Recordatorio al 25% (WS)	IFA (WA)	IFA (WD)	IFA (WS)
Comprobador de código	9,36%‡	8,09%‡	7,87%‡	2,65%	5,66%	2.00%	-6,87%	-10,29%†	-5,81%
CódigoQL	12,09%‡	10,98%‡	9,93%‡	-4,72%	3,43%	-3,37%	-2,51%	-13,34%‡	-2,36%
Cppcheck	2,10%	2,10%	2,10%	0,81%	0,81%	0,81%	-0,44%	-0,39%	-0,44%
Flawnder	9,87%†	2,51%	8,84%	-8,58%	-0,38%	-7,94%	7,85%	-4,23%	6,21%
Inferir	9,04%‡	9,14%‡	9,05%‡	-1,98%	3,25%	-1,87%	-3,24%	-10,79%	-3,36%

‡ fuertemente significativo: < 0,002; † moderadamente significativo: 0,002 ≤ < 0,01 (se ajusta con la corrección de Bonferroni)

Tabla 7: Estadísticas de tiempo de cálculo

Tiempo(s)	Comprobador de código	CódigoQL	Comprobación de Cpp	Flawnder	Inferir
Mínimo	3	26	1	1	2
Significar	213	343	2.702	20	456
Mediana	74	148	1.546	6	184
Máximo	7.268	3.314	16.974	88	2.305

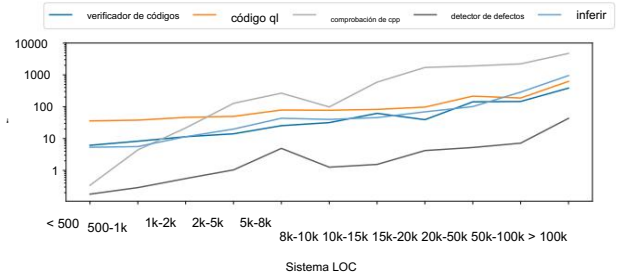


Figura 7: Tiempo de cálculo promedio en varios sistemas tamaños en VCC (LOC total)

El enfoque de análisis, lo que genera cuellos de botella en proyectos más grandes.

Hallazgo (tamaño del sistema): Las herramientas requieren un mayor tiempo de cálculo cuando el tamaño del sistema es mayor que 50kLOC-100kLOC.

5 Discusión

5.1 Rendimiento de SAST para la revisión de código

Con base en nuestros resultados empíricos, cada SAST ha demostrado ser prometedor. y limitaciones para la revisión de código. Aunque Flawnder tiene la tasa de detección más alta (Figura 3) y requiere el menor tiempo de cálculo, La priorización basada en advertencias utilizando Flawnder produce resultados relativamente bajos. rendimiento (Tabla 6). Esto se debe en parte a que Flawnder produce un mayor número de advertencias a las funciones modificadas que las otras SAST. Esto sugiere que la abundancia de advertencias de Flawnder puede No es ideal para priorizar funciones modificadas con un esfuerzo de revisión limitado. Por otro lado, aunque CodeQL y CodeChecker tienen tasas de detección más bajas en comparación con Flawnder, sus advertencias ayudan eficazmente a priorizar las funciones modificadas. Al usar Advertencias de CodeQL para priorización, la falsa alarma inicial (IFA) También se mejoró la precisión en la detección de funciones vulnerables. en comparación con la no priorización. De manera similar, CodeChecker puede ofrecer una Mejora sustancial en el recuerdo de la priorización basada en advertencias. Nuestros hallazgos sugieren que los SAST podrían ayudar a realizar revisiones de código seguras al Ayudando a priorizar las funciones modificadas según las advertencias. Sin embargo, Es necesario un mayor perfeccionamiento para apoyar mejor estas prácticas.

Aunque analizamos las advertencias y los tipos de vulnerabilidad en función En los pilares de CWE en este estudio, el elemento CWE de una advertencia dentro El mismo pilar CWE que el VCC puede no coincidir con el elemento CWE de la vulnerabilidad en el CVE correspondiente [48]. Por lo tanto, examinamos si el elemento CWE de la advertencia coincide con el elemento CWE de la vulnerabilidad para comprender la precisión del pilar CWE proxy. Tomamos una muestra de 400 advertencias de las 7564 advertencias que todos SAST producidos en funciones vulnerables independientemente de la vulnerabilidad tipos (S5:1Fn-Cualquiera), que nos permiten sacar una conclusión con una Nivel de confianza del 95% y un margen de error del 5% [38]. Siguiendo a Li En el enfoque de et al. [48] , determinamos si el elemento CWE de una advertencia coincide con el elemento CWE de un VCC utilizando las descripciones de elementos CWE. Descubrimos que el 66,42% de las advertencias dentro de los mismos pilares de CWE ya que el VCC coincide con los elementos CWE de los VCC. También encontramos que 10,27% de las advertencias con los diferentes pilares CWE de la Los VCC en realidad coinciden con los elementos CWE de los VCC.8

Para proporcionar información más rigurosa más allá de la tasa de detección en A nivel de función, investigamos más a fondo la relevancia de la advertencias en las líneas de código en VCC (es decir, las líneas advertidas) a la vulnerabilidad reportada en los CVE. Para ello, manualmente Analizar si las advertencias realmente reflejan las vulnerabilidades. en los VCC. Examinamos la siguiente información: los advertidos líneas, la descripción de la vulnerabilidad en el CVE asociado y la confirmación de xing. Clasificamos las advertencias en tres grupos (siguiendo a Thung et al. [65]): 1) Relevante: la línea advertida y la Las advertencias asociadas son relevantes para la vulnerabilidad y el xing confirmar, 2) Irrelevante: la línea de advertencia y la advertencia son completamente irrelevantes para la vulnerabilidad, y 3) Inseguro: la advertencia es relevante para la vulnerabilidad, pero la línea de advertencia no lo es; y viceversa versa. El grupo relevante estima una tasa de verdaderos positivos de las advertencias . El grupo irrelevante estima las tasas de falsos positivos de las Advertencias. Informamos del grupo inseguro, ya que aún puede ser beneficioso. En las revisiones de código, donde los revisores adquieren mayor conciencia de seguridad. de las advertencias adyacentes [33]. Basado en el mismo conjunto de muestras de 400 advertencias, encontramos que el 7% de las advertencias son relevantes para el Línea de advertencia y vulnerabilidad, el 76% de las advertencias son irrelevantes a la línea de advertencia y vulnerabilidad, y el 17% de las advertencias son ya sea relevante para la línea advertida o la vulnerabilidad. Estos resultados,

8Los resultados de la evaluación se incluyen en el paquete de datos.

Junto con los hallazgos de la Pregunta 2 (Sección 4.2), se sugiere que la información de advertencia podría ayudar a los revisores a priorizar las funciones que revisarán primero. Sin embargo, sigue siendo esencial que los revisores dediquen un esfuerzo meticuloso a examinar e identificar las vulnerabilidades dentro de estas funciones priorizadas.

5.2 Implicaciones y sugerencias En esta sección analizamos las implicaciones para los profesionales, los desarrolladores de SAST y los investigadores.

5.2.1 Para profesionales. Nuestros resultados resaltan el potencial del uso de SAST para revisiones seguras de código. Para maximizar los beneficios de SAST... Para las revisiones de código seguro compatibles, ofrecemos las siguientes recomendaciones . 1 Las SAST pueden aprovecharse para reducir el esfuerzo de revisión de código. La Tabla 6 muestra que, con un esfuerzo fijo (25 % del nivel de complejidad de las funciones modificadas), la priorización basada en advertencias puede aumentar la eficiencia en la identificación de funciones vulnerables en comparación con la no priorización. RQ2 muestra que la priorización basada en la densidad de advertencias generalmente proporciona una mejora sustancial, lo que sugiere que se podría adoptar este enfoque de priorización . Además, RQ3 muestra que el tiempo de computación de SAST es relativamente corto, lo que debería estar dentro del período de espera de revisión de código [47]. 2 Las funciones modificadas que reciben advertencias de SAST deben investigarse cuidadosamente independientemente de los tipos de advertencia. RQ1 muestra que los tipos de advertencia pueden no ser directamente relevantes para la vulnerabilidad de los VCC. La Figura 3 muestra que, independientemente de los tipos de advertencia, los SAST pueden producir advertencias en las funciones vulnerables (S5:1Fn-Any); sin embargo, solo entre el 1% y el 46% de los VCC reciben advertencias con el mismo tipo que sus CVE (S6:1Fn-Same). Este resultado sugiere que los revisores no deben limitar el enfoque de la revisión de seguridad a los tipos de advertencia que produjo la herramienta. 3 SAST debe seleccionarse cuidadosamente en función de las necesidades de seguridad del proyecto y las limitaciones de recursos. Descubrimos que varios factores pueden afectar la efectividad de los SAST, por ejemplo, los tipos de vulnerabilidad, el tiempo disponible y los recursos computacionales. Los proyectos de software deben determinar sus necesidades de seguridad y las limitaciones de recursos al elegir los SAST. Si el tipo de vulnerabilidad correspondiente es la principal preocupación, se deben reconocer las necesidades de seguridad específicas, como los tipos de vulnerabilidad que conducen a un impacto crítico. Por ejemplo, la Tabla 5 muestra que solo CodeQL puede detectar el tipo de vulnerabilidad Access Ctrl (CWE-284) y Control Flow (CWE-691), mientras que solo CodeChecker puede detectar el tipo de vulnerabilidad Cond Check (CWE-703). Alternativamente, cuando la potencia computacional es abundante, la combinación de SAST puede aumentar la efectividad de la detección de vulnerabilidades en un 26 % (Figura 4). Si el proyecto es excepcionalmente grande, el tiempo de computacional se debe evaluar deliberadamente, ya que RQ3 indica un aumento sustancial para proyectos que superan los 50 kLOC-100 kLOC. Además , encontramos que la configuración de SAST puede exigir experiencia técnica . Sin embargo, con el apoyo de herramientas de automatización de compilación, la configuración de SAST debería ser un esfuerzo de implementación único.

5.2.2 Para los desarrolladores de SAST. A pesar del valor potencial de los SAST, consideramos que su incorporación en las revisiones de código seguro aún requiere apoyo adicional. Ofrecemos las siguientes recomendaciones a los desarrolladores de SAST. 1. Proporcionar asesoramiento sobre vulnerabilidades en las advertencias. Nuestra evaluación manual muestra que algunas advertencias son directamente relevantes para las VCC. Sin embargo, algunas advertencias con pilares de CWE no coincidentes también pueden identificar vulnerabilidades. La Figura 3 también muestra que las VCC Con los tipos correctos de vulnerabilidad se puede reducir hasta un 35% a nivel de función, es decir, CodeQL en S5 (1Fn-Any) y S6 (1Fn-Same). Incluso si se detecta el problema relevante, la información de advertencia imprecisa puede distraer a los revisores. SAST debe resaltar el riesgo si un problema puede conducir a una vulnerabilidad, de lo contrario, la advertencia puede percibirse como no vulnerable. 2 La información de severidad de las advertencias debe mejorarse. La Tabla 6 muestra que priorizar las funciones modificadas con severidad de advertencia no produce una mejora sustancial en comparación con otros enfoques. Dado que los desarrolladores generalmente siguen advertencias de alta severidad durante el proceso de desarrollo [68], la severidad de las advertencias debería ser más confiable.

5.2.3 Para investigadores. Algunas prácticas en las revisiones de código seguro soportadas por SAST no se comprenden bien. Sugerimos que trabajos futuros exploren los siguientes aspectos. 1 Se deben explorar las reglas para código no convencional/no terminado. Los cambios de código pueden ser desordenados durante el ciclo de desarrollo y las revisiones de código. En nuestra RQ1, observamos que el código de desarrollo que se desvía de las reglas predefinidas puede causar vulnerabilidades no detectadas. Los modelos de IA (por ejemplo, modelos de lenguaje grandes) pueden ayudar a detectar vulnerabilidades sin precedentes más allá de las reglas definidas por el desarrollador de los SAST. Sin embargo, un trabajo reciente [69] sugiere que su capacidad de detección de vulnerabilidades podría estar restringida por una comprensión semántica insuficiente del código complejo. Por lo tanto, estudios futuros deberían investigar las soluciones que detectan vulnerabilidades en dicho código. 2 Se deben investigar técnicas efectivas para usar SAST para la revisión de código seguro . La Tabla 6 muestra que la priorización basada en advertencias con los tres enfoques puede mejorar la precisión, la recuperación y el IFA en comparación con la no priorización. Mientras tanto, ciertos enfoques pueden causar inconvenientes en el rendimiento, por ejemplo, priorizar las funciones modificadas mediante la cantidad de advertencia de Flawnder puede reducir la recuperación y aumentar el IFA. Nuestra evaluación manual también destaca las altas tasas de falsos positivos de las advertencias (más del 76 %) en cambios vulnerables, lo que puede obstaculizar en gran medida la efectividad de las revisiones de código [37]. Estos resultados y los hallazgos en RQ2 (Sección 4.2) sugieren que las advertencias pueden ayudar a los revisores a priorizar funciones, pero los revisores deben examinar e identificar cuidadosamente las vulnerabilidades dentro de las funciones priorizadas. De hecho, las técnicas efectivas para usar SAST para la revisión segura de código aún están en gran parte sin explorar, lo que es un desafío abierto para el trabajo futuro. 3 Las vulnerabilidades no detectadas deben priorizarse. Nuestro resultado enfatiza que todos los SAST aún pueden pasar por alto muchas vulnerabilidades en los cambios de código, al mostrar que el 22% de los VCC no reciben las advertencias en los cambios de código vulnerables de ninguna herramienta (Sección 4.1). La imposibilidad de detectar vulnerabilidades reales enfatiza la importancia de mejorar los SAST. El trabajo futuro debe explorar las soluciones a estas deficiencias y minimizar las vulnerabilidades no detectadas. 4 Se debe investigar un enfoque conjunto de múltiples SAST . La Figura 4 muestra que la combinación de herramientas puede mejorar la efectividad de la identificación de vulnerabilidades. Sin embargo, no es viable para proyectos con recursos informáticos limitados utilizar muchos SAST, especialmente para proyectos grandes (Figura 7). Una guía práctica para combinar SAST es útil para los profesionales.

6 Amenazas a la validez En esta sección analizamos las posibles amenazas a la validez de este estudio.

Validez interna: La calidad del conjunto de datos de VCC afecta los resultados del estudio. Hemos verificado cambios vulnerables y descartado VCC que no pudieron ser

vinculado a las confirmaciones de xing y excluyó una confirmación redundante causada por un proyecto renombrado. Para la selección de SAST, elegimos cuidadosamente los SAST populares que los profesionales pueden emplear en el proceso de desarrollo. Éramos conscientes de que habilitar diferentes reglas de SAST puede afectar la efectividad en general. Nos adherimos a las reglas predeterminadas porque los desarrolladores de herramientas lo recomiendan encarecidamente. Además, encontramos problemas de falta de memoria al intentar habilitar reglas adicionales. Tras el mapeo manual de las advertencias al pilar CWE (Sección 3.3.3), pueden ocurrir sesgos humanos. Para abordar esto, los dos autores primero sientan las bases para el mapeo antes de proceder en colaboración con las advertencias restantes. De manera similar, reconocemos el error potencial durante la evaluación manual de las advertencias. Por último, como MITRE actualiza regularmente las entradas CWE y CVE , no se puede garantizar la sostenibilidad de los resultados.

Validez de constructo: La validez de constructo se refiere a los indicadores medidos. Según la literatura previa [53, 66, 68] , los revisores Nosotros asumimos priorizan las funciones modificadas con las advertencias durante las revisiones de código. Analizamos el rendimiento de la priorización con diferentes esfuerzos fijos (LOC del %) para garantizar la consistencia. Para aumentar la validez, exploramos diversos enfoques de priorización, como la cantidad, la densidad y la gravedad de las advertencias. Sin embargo, los revisores pueden adoptar otros enfoques al revisar los cambios de código, lo que introduce posibles variaciones más allá de los constructos examinados.

Validez externa: La validez externa concierne a la generalización de los resultados. Nuestros hallazgos están respaldados por la amplitud de nuestro conjunto de datos, que contiene 815 confirmaciones de 92 proyectos diversos que contribuyeron a 319 vulnerabilidades. Nuestros resultados se basan en cinco SAST. Un conjunto más amplio de SAST puede mejorar la generalización de los hallazgos. No obstante, estos cinco SAST se usan comúnmente, se mantienen activamente y son compatibles con los sistemas estudiados y ejecutables a través de CLI (Sección 3.3.1). Para respaldar el trabajo futuro, publicamos un marco [35] que el trabajo futuro puede extender para SAST adicionales y conjuntos de datos de cambio de código en diversos lenguajes. También reconocemos que los hallazgos con respecto a la relevancia de la advertencia (Sección 5.1) se basan en las advertencias de muestra, que pueden tener una generalización limitada. Para transparencia y para facilitar el trabajo futuro, publicamos el marco de referencia, los conjuntos de datos y los resultados de la evaluación.

7 Trabajo relacionado

La efectividad de los SAST en los VCC no se ha investigado previamente . La mayoría de los trabajos existentes se centraron principalmente en evaluar los SAST en las versiones publicadas de software o los programas diseñados que contienen vulnerabilidades conocidas. Los conjuntos de datos de las confirmaciones de código enviadas por los desarrolladores para revisiones de código no suelen usarse en los estudios de SAST. Lipp et al. [49] evalúa cinco SAST de C/C++ gratuitos y uno comercial e informó que los SAST pueden pasar por alto entre el 47 % y el 80 % de las vulnerabilidades. Su conjunto de datos de referencia incluye 111 vulnerabilidades portadas del conjunto de datos Magma [42] y 81 vulnerabilidades de versiones publicadas de Binutils y FFmpeg. En este trabajo, utilizamos el conjunto de datos de las confirmaciones que contribuyen a la vulnerabilidad, que incluyen funciones vulnerables y archivos vulnerables, y descubrimos que los SAST pueden detectar al menos una función vulnerable del 78 % de los VCC. Kannavara [46] evaluó la efectividad de Klocwork, un SAST multilenguaje, en el kernel de Linux. Sin embargo, solo una versión del kernel de Linux... Se utilizó. En este trabajo, investigamos múltiples versiones de código (815 VCC) del proceso de desarrollo. Un estudio reciente de Li et al. [48] investigó un contexto estrechamente relevante para nuestro trabajo. Evaluaron siete SAST utilizando confirmaciones de código e informaron que los SAST detectan solo el 12,7 % de las vulnerabilidades con los tipos de vulnerabilidad correctos . Sin embargo, se centraron en los SAST de Java y siete tipos de vulnerabilidad CWE. En este trabajo, estudiamos los SAST de C/C++ utilizando el conjunto de datos VCC con ocho tipos de vulnerabilidad CWE.

A diferencia de trabajos anteriores, nos centramos en las confirmaciones de código de desarrollo del mundo real que son objeto de revisiones de código. Este novedoso conjunto de datos puede mejorar la comprensión del rendimiento de SAST, ofreciendo información sobre su rendimiento en la fase de desarrollo de un proyecto de software. También somos los primeros en investigar la priorización basada en advertencias y demostrar que la eficacia en la identificación de vulnerabilidades puede mejorarse cuando las funciones modificadas se clasifican con la información de las advertencias de SAST.

8 Conclusión

En este estudio, realizamos una investigación empírica para comprender los beneficios prácticos de los SAST de C y C++ para las revisiones de código seguro. Los resultados muestran que el 52% de los VCC pueden ser advertidos por una sola herramienta en las funciones modificadas que contienen código vulnerable. Al combinar herramientas, la efectividad de detección puede aumentar en un 26%. Para las revisiones de código seguro, priorizar las funciones modificadas con advertencias de SAST puede mejorar la precisión (es decir, 12% de precisión y 5,6% de recuperación) y reducir la falsa alarma inicial (es decir, líneas de código en funciones no vulnerables que deben inspeccionarse hasta la primera función vulnerable) en un 13%. Además, el tiempo de cálculo promedio de los SAST debe coincidir con el tiempo de espera de las revisiones de código. Sin embargo, al menos el 76% de las advertencias en funciones vulnerables pueden ser irrelevantes para la vulnerabilidad en los VCC, y el 22% de los VCC pueden pasar desapercibidos.

Con base en estos hallazgos, sugerimos que los proyectos de software seleccionen SAST que se ajusten a sus necesidades de seguridad y limitaciones de recursos, aprovechen los SAST para optimizar los esfuerzos de revisión de código e inspeccionen las funciones modificadas con advertencias de SAST, independientemente de los tipos de advertencia. Por otro lado, los desarrolladores de SAST deben mejorar el asesoramiento sobre vulnerabilidades en las advertencias y mejorar la información de gravedad de las advertencias. El trabajo futuro puede mejorar los SAST en las vulnerabilidades no detectadas, explorar las reglas de SAST para el código de desarrollo, investigar técnicas efectivas para usar SAST en revisiones de código seguro y explorar un enfoque de conjunto que involucre múltiples SAST. También publicamos el marco de evaluación de SAST [35] y el conjunto de datos para respaldar futuras investigaciones.

Declaración de disponibilidad de datos

Hemos publicado los conjuntos de datos y scripts utilizados en este estudio para fomentar trabajos futuros en revisiones de código seguro respaldadas por SAST [36].

Agradecimientos

Este trabajo fue financiado por Nectar Research Cloud, una plataforma de investigación colaborativa apoyada por Australian Research Data Commons (ARDC) financiado por NCRIS . Patanamon Thongta-nunam y Van-Thuan Pham fueron apoyados por dos proyectos del Discovery Early Career Researcher Award (DE-CRA) del Australian Research Council (DE210101091 y DE230100473). Agradecemos sinceramente a los revisores anónimos por sus valiosos comentarios.

Referencias

[1] [nd]. Acerca de la comparación de ramas en solicitudes de extracción - Documentación de GitHub. https://docs.github.com/es/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about- comparison-branches-in-pull-requests/ [2] [nd]. CTR51-

CPP. Uso de referencias, punteros e iteradores válidos para referenciar elementos de un contenedor - Estándar de codificación C++ SEI CERT - Con- https://wiki.sei.cmu.edu/ uencia. conuence/display/cplusplus/CTR51-CPP.+Utilice+referencias+válidas%2C+punteros%2C+e+iteradores+para+referenciar+elementos+de+un+contenedor

[3] [nd]. CVE | Resumen. https://www.cve.org/About/Overview [4] 2015. NVD - CVE-2015-8895. https://nvd.nist.gov/vuln/detail/CVE-2015-8895 [5] 2015. NVD - CVE-2015-9059. https://nvd.nist.gov/vuln/detail/CVE-2015-9059 [6] 2017. NVD - CVE-2017-12997. https://nvd.nist.gov/vuln/detail/CVE-2017-12997 [7] 2017. NVD - CVE-2017-13040. https://nvd.nist.gov/vuln/detail/CVE-2017-13040 [8] 2017. NVD - CVE-2017-13083. https://nvd.nist.gov/vuln/detail/CVE-2017-13083 [9] 2018. NVD - CVE-2018-17294. https://nvd.nist.gov/vuln/detail/CVE-2018-17294 [10] 2018. NVD - CVE-2018-7186. https://nvd.nist.gov/vuln/detail/CVE-2018-7186 [11] 2018. NVD - CVE-2018-7485. https://nvd.nist.gov/vuln/detail/CVE-2018-7485 [12] 2020. NVD - CVE-2020-5260. https://nvd.nist.gov/vuln/detail/CVE-2020-5260 [13] 2021. Consultas LGTM estándar para C/C++. https://github.com/github/codeql/ blob/principal/cpp/ql/src/codeql-suites/cpp-lgtm.qls [14] 2022. Comprobadores de Clang-Tidy. https://releases.llvm.org/13.0.1/tools/clang/tools/ extra/docs/clang-tidy/checks/list.html [15] 2023. CWE - CWE-457: Uso de variable no inicializada (4.12). https://cwe.mitre. org/data/denitions/457.html [16] 2023. CWE - CWE-664: Control inadecuado de un recurso a lo largo de su vida útil (4.12). https://cwe.mitre.org/data/denitions/664.html

[17] 2023. CWE - CWE-699: Desarrollo de software (4.12). https://cwe.mitre.org/datos/definiciones/699.html

[18] 2023. CWE - CWE-825: Desreferencia de puntero expirado (4.12). https://cwe.mitre. org/data/denitions/825.html [19] 2023.

Flawnder. https://dwheeler.com/awnder/ [20] 2023. Analizador estático de inferencia. https://fbinfer.com/ [21] 2023.

Analizadores de seguridad de código fuente - NIST. https://www.nist.gov/itl/ssd/grupo de calidad de software/analizadores de seguridad de código fuente [22] 2024. Comprobadores de Clang. https://clang.llvm.org/docs/analyzer/checkers.html [23] 2024. CodeChecker. https://codechecker.readthedocs.io/es/latest/ [24] 2024. CodeQL. https://codeql.github.com/

[25] 2024. Cppcheck: una herramienta para el análisis de código estático C/C++. https://cppcheck.sourceforge.io/

[26] 2024. Inferir - Tipos de problemas. https://fbinfer.com/docs/all-issue-types/ [27]

2024. terryyin/lizard: Un analizador de complejidad de código simple que no se preocupa por los archivos de encabezado de C/C++ ni por las importaciones de Java, compatible con la mayoría de los lenguajes populares. https://github.com/terryyin/lizard [28] Bushra Aloraini y Meiyappan Nagappan. 2017. Evaluación de herramientas de análisis estático de última generación, gratuitas y de código abierto, contra errores de buer en aplicaciones Android. En Actas de la Conferencia Internacional IEEE sobre Mantenimiento y Evolución de Software 2017, ICSME 2017. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 295–306. https://doi.org/10.1109/ICSME.2017.77 [29] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi y Yoshiki Higo. 2019. Un estudio empírico de las advertencias de seguridad de herramientas de prueba de seguridad de aplicaciones estáticas. Journal of Systems and Software 158 (12 de diciembre de 2019), 110427. https://doi.org/10.1016/J.JSS.2019.110427 [30] Vipin Balachandran. 2013. Reducción del esfuerzo humano y mejora de la calidad en las revisiones de código por pares mediante análisis estático automático y recomendaciones de revisores . En Actas de la Conferencia Internacional sobre Ingeniería de Software. 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[31] Moritz Beller, Alberto Bacchelli, Andy Zaidman y Elmar Juergens. 2014. Revisiones de código modernas en proyectos de código abierto: ¿Qué problemas resuelven?. En la 11.ª Conferencia de trabajo sobre minería de repositorios de software, MSR 2014 - Actas. Association for Computing Machinery, 202–211. https://doi.org/10.1145/2597073.2597082

[32] Moritz Beller, Radjino Bholanath, Shane McIntosh y Andy Zaidman. 2016. Análisis del estado del análisis estático: Una evaluación a gran escala en software de código abierto. 23.ª Conferencia Internacional IEEE sobre Análisis, Evolución y Reingeniería de Software, SANER 2016 1 (5 de febrero de 2016), 470–481. https://doi.org/10.1109/SANER.2016.105

[33] Larissa Braz y Alberto Bacchelli. 2022. Seguridad del software durante la revisión de código moderna: la perspectiva del desarrollador. En 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 2022). Association for Computing Machinery, Nueva York, EE. UU., 810–821. https://doi.org/10.1145/3540250.3549135 [34] Larissa Braz, Enrico Fregnan, Gul Calikli y Alberto Bacchelli. 2021. ¿Por qué los desarrolladores no detectan una validación de entrada incorrecta? '; DROP TABLE Artículos; -. En Actas - Conferencia Internacional sobre Ingeniería de Software. IEEE Computer Society, 499–511. https://doi.org/10.1109/ICSE43902.2021.00054

[35] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham y Christoph Treude. 2024. Un marco de evaluación comparativa automatizada extensible para la evaluación de herramientas de análisis estático mediante confirmaciones de código abierto. https://doi.org/10.5281/zenodo.10259921

[36] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham y Christoph Treude. 2024. Conjunto de datos para un estudio empírico de herramientas de análisis estático para la revisión segura de código. https://doi.org/10.5281/zenodo.12653656 [37]

Maria Christakis y Christian Bird. 2016. Lo que los desarrolladores desean y necesitan del análisis de programas: un estudio empírico. En Actas de la 31.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada. Asociación para la Maquinaria de Computación (ACM), 332–343. https://doi.org/10.1145/2970276.2970347

[38] Marco Di Biase, Magiel Bruntink y Alberto Bacchelli. 2016. Una perspectiva de seguridad en la revisión de código: El caso de Chromium. En Actas - 2016 IEEE 16.ª Conferencia Internacional de Trabajo sobre Análisis y Manipulación de Código Fuente, SCAM 2016. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 21-30. https://doi.org/10.1109/SCAM.2016.30 [39] Michael D. Ernst. 2004.

Análisis estático y dinámico: sinergia y dualidad. En PASTE '04: Actas del 5.º taller ACM SIGPLAN-SIGSOFT sobre análisis de programas para herramientas de software e ingeniería. Asociación para la Maquinaria de Computación (ACM), 35–35. https://doi.org/10.1145/996821.996823

[40] Katerina Goseva-Popstojanova y Andrei Perhinschi. 2015. Sobre la capacidad del análisis de código estático para detectar vulnerabilidades de seguridad. Tecnología de la Información y el Software 68 (12 de 2015), 18–33. https://doi.org/10.1016/J.INFSOF.2015.08.002 [41]

Mark Harman y Peter O'Hearn. 2018. De startups a ampliaciones: Oportunidades y problemas abiertos para el análisis estático y dinámico de programas. En Actas de la 18.ª Conferencia Internacional de Trabajo del IEEE sobre Análisis y Manipulación de Código Fuente , SCAM 2018. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 1–23. https://doi.org/10.1109/SCAM.2018.00009

[42] Ahmad Hazimeh, Adrian Herrera y Mathias Payer. 2020. Magma: Un punto de referencia para la fuzzing de la verdad fundamental. En Actas de la ACM sobre Medición y Análisis de Sistemas Informáticos, vol. 4. Asociación para la Maquinaria Informática (ACM), 1–29. https://doi.org/10.1145/3428334 [43] Sarah

Heckman y Laurie Williams. 2008. Sobre el establecimiento de un punto de referencia para la evaluación de técnicas de priorización y clasificación de alertas de análisis estático. En ESEM'08: Actas del Simposio Internacional ACM-IEEE 2008 sobre Ingeniería y Medición de Software Empírico. 41–50. https://doi.org/10.1145/1414004.1414013 [44] Yang Hong,

Chakkrit Kla Tantithamthavorn y Patanamon Pick Thongtanunam. 2022. ¿Dónde debería consultar? Líneas de recomendación que los revisores deberían tener en cuenta. En Actas - Conferencia Internacional IEEE 2022 sobre Análisis, Evolución y Reingeniería de Software, SANER 2022. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 1034–1045. https://doi.org/10.1109/SANER53432.2022.00121 [45] Emanuele Iannone, Roberta Guadagni,

Filomena Ferrucci, Andrea De Lucia y Fabio Palomba. 2022. La vida secreta de las vulnerabilidades del software: un estudio empírico a gran escala. Transacciones IEEE sobre ingeniería de software (2022). https://doi.org/10.1109/TSE.2022.3140868

[46] Raghudeep Kannavara. 2012. Asegurando código abierto mediante análisis estático. En Actas de la 5.ª Conferencia Internacional IEEE sobre Pruebas, Verificación y Validación de Software, ICST 2012. 429–436. https://doi.org/10.1109/ICST.2012.123 [47] Gunnar

Kudrjavets, Aditya Kumar, Nachiappan Nagappan y Ayushi Rastogi. 2022. Minería de datos de revisión de código para comprender los tiempos de espera entre la aceptación y la fusión: un análisis empírico. En la 19.ª Conferencia Internacional IEEE/ACM sobre Minería de Repositorios de Software (MSR) de 2022. IEEE Computer Society, 579-590 . https://doi.org/10.1145/3524842.3528432

[48] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu y Yixiang Chen. 2023. Comparación y evaluación de herramientas de pruebas de seguridad de aplicaciones estáticas (SAST) para Java. En las actas de la 31.ª Conferencia y Simposio Conjunto Europeo de Ingeniería de Software de la ACM sobre los Fundamentos de la Ingeniería de Software (ESEC/FSE '23). Asociación para la Maquinaria de Computación. https://doi.org/10.1145/3611643.3616262

[49] Stephan Lipp, Sebastian Banescu y Alexander Pretschner. 2022. Un estudio empírico sobre la efectividad de los analizadores estáticos de código C para la detección de vulnerabilidades. En Actas del 31.º Simposio Internacional ACM SIGSOFT sobre Pruebas y Análisis de Software. ACM, Nueva York, NY, EE. UU., 544–555. https://doi.org/10.1145/3533767.3534380

[50] Mika V. Mäntylä y Casper Lassenius. 2009. ¿Qué tipos de defectos se detectan realmente en las revisiones de código? IEEE Transactions on Software Engineering 35, 3 (2009), 430–448. https://doi.org/10.1109/TSE.2008.71

[51] Sahar Mehrpour y Thomas D. LaToza. 2022. ¿Pueden las herramientas de análisis estático detectar más defectos? Ingeniería de Software Empírica 28, 1 (2022). https://doi.org/10.1007/ S10664-022-10232-4 [52] Jonathan Metzman,

László Szekeres, Laurent Simon, Read Sprabery y Ab-hishek Arya. 2021. FuzzBench: Una plataforma y servicio abierto de evaluación comparativa de fuzzers. En ESEC/FSE 2021 - Actas de la 29.ª Reunión Conjunta de la ACM, Conferencia Europea de Ingeniería de Software y Simposio sobre los Fundamentos de la Ingeniería de Software, vol. 21. Association for Computing Machinery, Inc., 1393–1403. https://doi.org/10.1145/3468264.3473932

[53] Tukaram Muske y Alexander Serebrenik. 2016. Estudio de enfoques para el manejo de alarmas de análisis estático. En Actas - 2016 IEEE 16th International Working

Conferencia sobre análisis y manipulación de código fuente, SCAM 2016. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 157–166. https://doi.org/10.1109/SCAM.2016.25

[54] Marcus Nachtigall, Michael Schlichtig y Eric Bodden. 2022. Un estudio a gran escala de los criterios de usabilidad abordados por herramientas de análisis estático. En ISSTA 2022 - Actas del 31.º Simposio Internacional ACM SIGSOFT sobre Pruebas y Análisis de Software. Association for Computing Machinery, Inc., 532–543. https: //doi.org/10.1145/3533767.3534374 [55] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia y Marco Vieira. 2017. Sobre la combinación de diversas herramientas de análisis estático para la seguridad web: un estudio empírico. En actas - 2017 13.ª Conferencia europea sobre informática confiable, EDCC 2017. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 121–128. https://doi.org/10.1109/EDCC.2017.16 [56] Sebastiano Panichella, Venera Arnaoudova, Massimiliano Di Penta y Giuliano Antoniol. 2015. ¿Podrían las herramientas de análisis estático ayudar a los desarrolladores con las revisiones de código?

En la 22.ª Conferencia Internacional IEEE sobre Análisis, Evolución y Reingeniería de Software de 2015, SANER 2015 - Actas. Instituto de Ingenieros Eléctricos y Electrónicos Inc., 161–170. https://doi.org/10.1109/SANER.2015.7081826 [57] Massimiliano Di Penta, Luigi Cerulo y Lerina Aversano. 2009. La vida y la muerte de las vulnerabilidades detectadas estáticamente: Un estudio empírico. Tecnología de la Información y el Software 51, 10 (10 de 2009), 1469–1484. https://doi.org/10.1016/J.INFSOF.2009.04.013

[58] Peter C. Rigby y Christian Bird. 2013. Prácticas Convergentes de Revisión por Pares de Software Contemporáneo. En Actas de la 9.ª Reunión Conjunta sobre Fundamentos de Ingeniería de Software (2013) - ESEC/FSE 2013. ACM Press, Nueva York, EE. UU. https://doi.org/10.1145/2491411 [59] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza y Rocco Oliveto. 2021. Evaluación de implementaciones de SZZ mediante un oráculo orientado al desarrollador. En Actas de la Conferencia Internacional sobre Ingeniería de Software. IEEE Computer Society, 436–447. https: //doi.org/10.1109/ICSE43902.2021.00049

[60] Caitlin Sadowski, Jerey Van Gogh, Ciera Jaspan, Emma Söderberg y Collin Winter. 2015. Tricorder: Construyendo un ecosistema de análisis de programas. En Actas de la Conferencia Internacional de Ingeniería de Software, vol. 1. IEEE Computer Society, 598–608. https://doi.org/10.1109/ICSE.2015.76

[61] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T. Stolee y Brittany Johnson. 2017. Evaluación de cómo las herramientas de análisis estático pueden reducir el esfuerzo de revisión de código. En Actas del Simposio IEEE sobre Lenguajes Visuales y Computación Centrada en el Ser Humano, VL/HCC, vol. 2017-octubre. IEEE Computer Society, 101–105. https://doi.org/10.1109/VLHCC.2017.8103456

[62] Śliwerski, Jacek, Zimmermann, Thomas y Zeller, Andreas. 2005. ¿Cuándo inducen los cambios xes? ACM SIGSOFT, Notas de Ingeniería de Software 30, 4 (5, 2005), 1–5. https://doi.org/10.1145/1082983.1083147

[63] Darko Stefanović, Danilo Nikolić, Dušanka Dakić, Ivana Spasojević y Sonja Ristić. 2020. Herramientas de análisis de código estático: una revisión sistemática de la literatura. En Anales de DAAAM y Actas del Simposio Internacional DAAAM, vol. 31. DAAAM Internacional Viena, 565–573. https://doi.org/10.2507/31ST.DAAAM.PROCEDIMIENTOS.078

[64] Christopher Thompson y David Wagner. 2017. Un estudio a gran escala sobre la revisión de código moderna y la seguridad en proyectos de código abierto. En la Serie de Actas de la Conferencia Internacional ACM. Association for Computing Machinery, 83–92. https://doi.org/10.1145/3127005.3127014

[65] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman y Premkumar T. Devanbu. 2012. ¿Hasta qué punto podríamos detectar defectos de campo? Un estudio empírico de falsos negativos en herramientas estáticas de detección de errores. 27.ª Conferencia Internacional IEEE/ACM sobre Ingeniería de Software Automatizada, ASE 2012 - Actas, 50–59. https://doi.org/10.1145/2351676.2351685 [66] Alexander Trautsch, Steen Herbold y Jens Grabowski. 2023. ¿Valen la pena las herramientas de análisis estático automatizado? Una investigación sobre la densidad relativa de advertencias y la calidad del software externo en el ejemplo de los proyectos de código abierto Apache. Ingeniería de Software Empírica 28, 3 (6 2023), 1–21. https://doi.org/10.1007/S10664- 023-10301-2

[67] Stephen Turner. 2014. Vulnerabilidades de seguridad de los diez principales lenguajes de programación : C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl y Ruby. Revista de Investigación Tecnológica (2014), 1–16. http://gauss.ececs.uc.edu/Courses/ c6056/pdf/131731.pdf [68] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall y Andy Zaidman. 2020. Cómo interactúan los desarrolladores con herramientas de análisis estático en diferentes contextos. Ingeniería de Software Empírica 25, 2 (3 2020), 1419–1457 . https://doi.org/10.1007/S10664-019-09750-5 [69] Zhilong Wang, Lan Zhang, Chen Cao y Peng Liu. 2023. La efectividad de los modelos de lenguaje grandes (ChatGPT y CodeBERT) para el análisis de código orientado a la seguridad. (7 2023). https://doi.org/10.48550/arXiv.2307.12488 [70] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tan-tithamthavorn, Hideaki Hata y Kenichi Matsumoto. 2022. Predicción de líneas defectuosas mediante una técnica independiente del modelo. Transacciones IEEE sobre ingeniería de software 48, 5 (5 2022), 1480–1496. https://doi.org/10.1109/TSE.2020.3023177 [71] Charles Weir, Sammy Migues y Laurie Williams. 2022. Explorando el cambio en la responsabilidad de la seguridad. IEEE Security and Privacy 20, 6 (2022), 8–17. https: //doi.org/10.1109/MSEC.2022.3150238

Recibido el 12/04/2024; aceptado el 03/07/2024