Análisis estático de una aplicación JavaScript

Autor: Nicolás Armero Baigorri

Director: Yue Duan

Fecha: 27/08/2023

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

Análisis estático de una aplicación JavaScript

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2022-2023 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Nicolás Armero Baigorri Fecha: 27/ 08/ 2023

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Yue Duan Fecha: 27/ 08/ 2023

Análisis estático de una aplicación JavaScript

Autor: Nicolás Armero Baigorri

Director: Yue Duan

Fecha: 27/08/2023

Índice

ANÁLISIS ESTÁTICO DE UNA APLICACIÓN JAVASCRIPT

Autor: Armero Baigorri, Nicolás.

Director: Yue, Duan.

Entidad Colaboradora: IIT - Illinois Institute of Technology.

Resumen del proyecto

Introducción

Un programa informático se define como una serie de normas escritas en algún lenguaje de programación para que sea ejecutado por un ordenador. Los programas informáticos son un componente software formado por documentación y otros componentes [1].

Desde la creación de los primeros ordenadores en la segunda mitad del siglo pasado, se han usado estas máquinas para llevar a cabo tareas con distinto nivel de complejidad. El aumento de la potencia y de la velocidad de computación ha permitido que a medida que se iban mejorando las capacidades de los ordenadores, se creasen nuevos programas informáticos para poder resolver actividades o problemas cada vez más complejos que los anteriores.

Este aumento de la complejidad de los sistemas informáticos trajo consigo un aumento de las líneas de código. Este aumento de las líneas de código, unido a que los programas eran más complejos conllevó el inevitable incremento de bugs o errores en dichos programas. A medida que los programas se volvían más complejos, también los errores se volvían más complejos tanto de detectar como de corregir.

Por este motivo, las empresas desarrolladoras de software se encontraron ante la necesidad de hallar alguna manera de poder realizar pruebas a sus productos mientras se encontraban en la fase de producción, para así poder solucionar cualquier fallo que pudiese existir antes de lanzar el producto final al mercado.

De esta manera surgieron los programas de análisis de software.
El primero de estos programas fue la herramienta Lint, un programa de análisis estático para C que fue creado y desarrollado por los laboratorios Bell en 1977 [2]. Esta herramienta supuso el pistoletazo de salida para el desarrollo y creación de numerosas aplicaciones para llevar a cabo el análisis de aplicaciones software.

El objetivo del uso de programas de análisis de software es la corrección de errores durante la etapa de producción. Se trata de eliminar los errores en la etapa de producción por varios motivos. Por un lado, es la etapa donde supone un menor coste la corrección de estos fallos, ya que si se trata de corregir un error una vez la aplicación ha sido lanzada al mercado, resulta más complejo, y con lo cual más caro, la corrección de estos errores. Por otro lado, no llevar a cabo un análisis del código de la aplicación supone lanzar al mercado una aplicación con todos aquellos errores que hayan surgido en la aplicación, pudiendo dañar la imagen de la empresa.

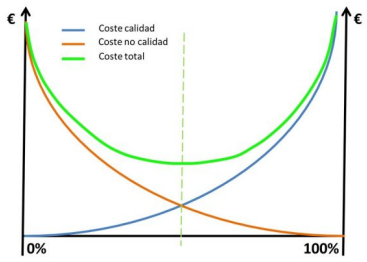


Figura 1: Gráfico que detalla la relación entre los costes de realizar análisis y no realizarlo en la fase de producción.

Una vez ha quedado claro por qué es necesario llevar a cabo un análisis de las aplicaciones software la pregunta que les surge a los desarrolladores es la cantidad de tiempo y de pruebas que se deben realizar. Y a pesar de que esto no tiene una respuesta global, ya que depende mucho del tipo de aplicación y para lo que vaya a usarse, se debe buscar un punto de equilibrio entre el coste de llevar a cabo las pruebas durante la fase de producción y el coste de mantenimiento una vez la aplicación se haya lanzado al mercado [3], tal y como se muestra en la figura 1.

Motivación y objetivos

Pese a que el uso de herramientas de análisis de software es muy común en empresas grandes y medianas de desarrollo de software, no lo es tanto en empresas más pequeñas o en aplicaciones de tipo *open source*. Por este motivo, queríamos analizar una aplicación de este tipo para, por un lado poder ayudar a los desarrolladores de la aplicación que estudiásemos y también demostrar que este tipo de aplicaciones contienen errores que pueden ser depurados con una herramienta de este tipo.

El objetivo de este trabajo era escoger una aplicación gratuita orientada a la educación y realizar las pruebas en ella. El objetivo final era la de ponernos en contacto con los desarrolladores de esta aplicación para hacerles saber de todos los posibles errores que hayamos encontrado.

Resultados

La aplicación que escogimos fue la aplicación de CODAP (Common Online Data Analysis Platform), desarrollada por *Concord Consortium* [4].

Esta aplicación es un «software educacional gratuito para análisis de datos». Esta aplicación esta desarrollada para alumnos entre 11 y 19 años de edad y su objetivo es ayudar a los alumnos a interiorizar y comprender mejor el temario visto en la clase. CODAP permite a creación y descubrimiento de patrones y conexiones entre los datos. También permite realizar análisis sobre texto escrito, permitiendo realizar un conteo de palabras o de signos de puntuación.

Para llevar a cabo el análisis de esta aplicación, estudiamos varias herramientas para poder ver cuál era la más óptima.

La primera que estudiamos fue la herramienta de *ExpoSE* [5]. Esta aplicación de tipo *opensource* está disponible para Linux y se trataba de una herramienta de análisis simbólico para aplicaciones JavaScript. Aunque conseguimos instalar la herramienta y realizar algunas pruebas a la aplicación que venía de ejemplo, tuvimos que descartar esta opción ya que no estaba diseñada para aplicaciones que estuviesen diseñadas para ejecutarse en navegadores, y nuestra aplicación, al ser de este tipo no podía analizarse con esta herramienta.

La segunda que estudiamos fue la herramienta de *Artemis* [6]. Esta aplicación solo estaba disponible para Linux 14. Tras instalar la aplicación observamos que esta sí que podía realizar análisis en CODAP. Sin embargo *Artemis*

se centraba únicamente en análisis de ciberseguridad. Como nuestro objetivo era realizar un análisis más general decidimos descartar esta herramienta.

Finalmente nos decidimos por la herramienta de *CodeQL*, creada por *GitHub*. Esta aplicación es una herramienta de análisis estático que nos permitía analizar la aplicación de *CODAP* tanto con pruebas de seguridad como con pruebas de otros estilos.
CodeQL funciona de una manera diferente a la mayoría de herramientas de análisis de software, ya que se comporta como si la aplicación a analizar fuese una base de datos.
Para poder realizar pruebas con cualquier aplicación, primero se debe transformar el software en una base de datos compatible con la herramienta. Algo que se puede conseguir con unos pocos comandos.

Una vez tuvimos instalada la herramienta y comprendimos su uso comenzamos a realizar las pruebas para determinar qué errores se encontraban en la aplicación. A continuación mostramos algunos de los resultados.

Botones con una descripción inadecuada

Como hemos comentado, nuestro análisis no se quería centrar únicamente en los apartados de seguridad, y esta prueba no se centraba en ello. Esta prueba se centraba en la experiencia del usuario, o «usabilidad» (usability en inglés) de la aplicación.

En esta prueba buscábamos cualquier botón que no funcionase correctamente o que no tuviese una correcta descripción de cara a los usuarios. Para esta prueba, no se usó solo la herramienta de *CodeQL*, sino que se creó tam-

bién un pequeño código de Python para poder sacar una lista de los botones que no tenían una buena descripción. Sin entrar en muchos detalles estos fueron los botones que descubrimos que no tenían una correcta descripción.

CODAP es una herramienta disponible en numerosos idiomas, lo que permite llegar a un número de estudiantes mucho mayor que tan solo con el inglés. Hay muchas zonas del mundo, sobre todo aquellas cuentan con un sistema educativo más débil donde la población no conoce otros idiomas, por ello, la traducción de todas las partes de esta herramienta resulta fundamental.

```
Actions with a bad "Undo" description:

DG.Undo.calculator.calculate
DG.Undo.axisAttributeChangeY2
DG.Undo.webView.show
DG.Undo.guide.configure
DG.Undo.componentTitleChange

Actions with a bad "Redo" description:

DG.Redo.calculator.calculate
DG.Redo.axisAttributeChangeY2
DG.Redo.webView.show
DG.Redo.guide.configure
DG.Redo.componentTitleChange
```

Figura 1: Acciones que no cuentan con una descripción para sus correspondientes botones.

Código imposible de ejecutar

Otra prueba que realizamos fue la de estudiar si había alguna parte del código que no se pudiese ejecutar nunca. Esto podía deberse o bien a una condición que jamás se cumpliría o bien por justo todo lo contrario, por tener una condición que siempre se cumpliese y con lo cual que se ejecutase otra rama del código.

En este caso encontramos el segundo caso, aquí encontramos una parte del código que no se ejecutaría nunca debido a un *if(true) break* que hacía que no se ejecutase el resto de código que había en el bucle.

Hay que aclarar que esto puede no ser un error, ya que esto puede haber sido introducido por los desarrolladores para evitar que esta funcionalidad esté disponible por el motivo que sea. Sin embargo, esta es una práctica poco aconsejable de programación, ya que en futuras versiones pueden activar esta funcionalidad por error y puede que produzca algún fallo que, además, será complicado de solucionar ya que no se cuenta con que esta funcionalidad esté activa.

```
playSound: function(a) {
    var b = this.sounds[a];
    if (true) {
        return
    }
    if (b && b.path && !b.audio) {
        b.audio = new Audio(b.path)
    }
    if (b.audio) {
        if (window.chrome) {
            b.audio.load()
        } else {
            b.audio.currentTime = 0
        }
        b.audio.play()
    }
},
```

Figura 1: Una de las funciones encargadas de reproducir el sonido con un *if(true)* justo antes.

Cross Site Scripting - XSS

El Cross-Site Scripting es un tipo de ataques que se hacen a aquellas páginas web que no han protegido sus formularios y que por lo tanto no filtran el contenido introducido por teclado del usuario [7].

Pese a que *CODAP* no tiene muchos campos en los cuáles el usuario pueda introducir texto, es importante que este se filtre para evitar que un usuario malintencionado pueda introducir algo de código ejecutable.

En este caso no tuvimos ningún resultado, pero a pesar de ello se pueden decir dos cosas. La primera es que aunque no hayamos obtenido ningún re-

sultado esta prueba es importante realizarla en caso de que existiese alguna vulnerabilidad frente a un ataque de este tipo. La segunda es que esta prueba puede servir para aumentar la confianza con la cual se afirma que no hay vulnerabilidades de este tipo, pero ninguna prueba va a poder aumentar el nivel de confianza hasta el 100 %.

Conclusiones

En este trabajo hemos realizado todas las tareas que nos propusimos desde un principio. Hemos explicado en qué consiste el análisis de código en un proyecto de software. Hemos explicado para qué se utiliza y las ventajas que el uso de este tipo de herramientas supone de cara a futuro a la hora de mantener la aplicación software.

Hemos demostrado como es necesario el análisis de las aplicaciones para evitar la introducción no solo de errores, sino de vulnerabilidades también. Hemos llevado a cabo distintos tipos de pruebas para encontrar distintos tipos de errores y vulnerabilidades. Hemos insistido en cómo unas buenas prácticas de programación y una buena organización puede ayudar a reducir este tipo de errores, aunque seguirán existiendo.

STATIC ANALYSIS OF A JAVASCRIPT APPPLICATION

Author: Armero Baigorri, Nicolás.

Supervisor: Yue, Duan.

Collaborating Entity: IIT - Illinois Institute of Technology.

Project summary

Introduction

A computer program is defined as a set of rules written in a programming language to be executed by a computer. A computer program is a software component consisting of documentation and other components [1].

Since the creation of the first computers in the second half of the last century, these machines have been used to carry out tasks with different levels of complexity. The increase in computing power and speed has meant that as the capabilities of computers have improved, new computer programs have been created to solve activities or problems that are increasingly more complex than the previous ones.

This increase in the complexity of computer systems brought with it an increase in the number of lines of code. This increase in lines of code, coupled with more complex programs, led to an inevitable increase in the number of bugs or errors in those programs. As programs became more complex, bugs also became more complex to detect and correct.

For this reason, software development companies were faced with the need to find a way to test their products while they were in the production phase,

in order to fix any bugs that might exist before releasing the final product to the market.

This is how software analysis programs came into being.
The first of these programs was the Lint tool, a static analysis program for C that was created and developed by Bell Laboratories in 1977 [2]. This tool was the starting point for the development and creation of numerous applications for software application analysis.

The purpose of using software analysis programs is to correct errors during the production stage. The aim is to eliminate errors in the production stage for several reasons. On the one hand, it is the stage where it is less expensive to correct these bugs, because if a bug needs to be corrected once the application has been released to the market, it is more complex, and therefore more expensive, to correct these bugs. On the other hand, not carrying out an analysis of the application code means launching an application on the market with all the errors that have arisen in the application, which could damage the company's image.

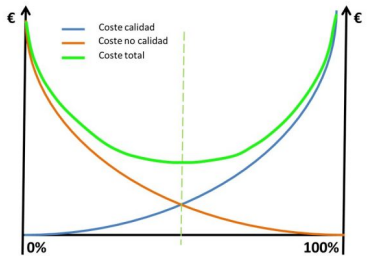


Figura 1: Graph detailing the relationship between the costs of performing analysis and not performing analysis in the production phase.

Once it has become clear why it is necessary to carry out an analysis of software applications, the question that arises for developers is the amount of time and tests to be performed. And although this does not have a global

answer, since it depends a lot on the type of application and what it is going to be used for, a balance must be found between the cost of testing during the production phase and the cost of maintenance once the application has been launched on the market [3], as shown in the figure 1.

Motivation and objectives

Although the use of software analysis tools is very common in large and medium-sized software development companies, it is not so common in smaller companies or in open source applications. For this reason, we wanted to analyze an application of this type in order, on the one hand, to help the developers of the application we were studying and, on the other hand, to demonstrate that this type of application contains errors that can be debugged with a tool of this type.

The objective of this work was to choose a free application oriented to education and to test it. The final goal was to get in touch with the developers of this application to let them know about all the possible bugs we found.

Results

The application we chose was the *CODAP* (Common Online Data Analysis Platform) application, developed by the Concord Consortium [4].

This application is a "free educational software for data analysis". This application is developed for students between 11 and 19 years of age and its objective is to help students internalize and better understand the subject matter seen in class. CODAP allows the creation and discovery of patterns

and connections between data. It also allows to perform analysis on written text, allowing to count words or punctuation marks.

To carry out the analysis of this application, we studied several tools to see which one was the most optimal.

The first one we studied was the *ExpoSE* tool [5]. This opensource type application is available for Linux and was a symbolic analysis tool for JavaScript applications. Although we managed to install the tool and perform some tests on the sample application, we had to discard this option since it was not designed for applications that were designed to run in browsers, and our application, being of this type, could not be analyzed with this tool.

The second one we studied was the *Artemis* tool [6]. This application was only available for Linux 14. After installing the application we noticed that it could perform analysis on CODAP. However, *Artemis* focused only on cybersecurity analysis. As our goal was to perform a more general analysis we decided to discard this tool.

Finally we decided to use the *CodeQL* tool, created by *GitHub*. This application is a static analysis tool that allowed us to analyze the *CODAP* application with both security tests and tests of other styles.
CodeQL works in a different way than most software analysis tools, since it behaves as if the application to be analyzed were a database. In order to test any application, the software must first be transformed into a database compatible with the tool. Something that can be achieved with a few commands.

Once we had the tool installed and understood how to use it, we started

to test to determine what errors were found in the application. Some of the results are shown below.

Buttons with inadequate descriptions

As we have mentioned, our analysis did not want to focus only on the security aspects, and this test did not focus on that. This test focused on the user experience, or usability, of the application.

In this test we were looking for any buttons that did not work correctly or did not have a correct description for the users. For this test, we did not only use the *CodeQL* tool, but we also created a small Python code in order to get a list of the buttons that did not have a good description. Without going into too much detail these were the buttons that we discovered that did not have a correct description.

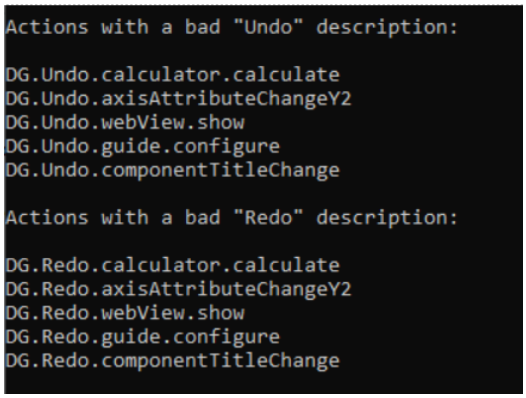


Figura 1: Actions that do not have a description for their corresponding buttons.

CODAP is a tool available in many languages, which makes it possible to reach a much larger number of learners than just English speakers. There are many areas of the world, especially those with weaker education systems where the population does not know other languages, so translation of all parts of the tool is essential.

Unreachable code

Another test we performed was to see if there was any part of the code that could never be executed. This could be due either to a condition that would never be fulfilled or just the opposite, to have a condition that would always be fulfilled and therefore another branch of the code would be executed.

```
playSound: function(a) {
    var b = this.sounds[a];
    if (true) {
        return
    }
    if (b && b.path && !b.audio) {
        b.audio = new Audio(b.path)
    }
    if (b.audio) {
        if (window.chrome) {
            b.audio.load()
        } else {
            b.audio.currentTime = 0
        }
        b.audio.play()
    }
},
```

Figura 1: One of the functions in charge of playing the sound with an if(true) just before.

In this case we find the second case, here we find a part of the code that would never be executed due to an if(true) break; which caused the rest of the code in the loop not to be executed.

It must be clarified that this may not be a bug, as this may have been introduced by the developers to prevent this functionality from being available for whatever reason. However, this is an unadvisable programming practice, since in future versions they may activate this functionality by mistake and it may produce a bug that, in addition, will be complicated to fix since this functionality is not expected to be active.

Cross Site Scripting - XSS

Cross-Site Scripting is a type of attack on web pages that have not protected their forms and therefore do not filter the user's keyboard input [7].

Although *CODAP* does not have many fields in which the user can enter text, it is important that the text is filtered to prevent a malicious user from entering some executable code.

In this case we had no results, but nevertheless two things can be said. The first is that although we have not obtained any result, this test is important to perform in case there is any vulnerability to an attack of this type. The second is that this test can be used to increase the confidence that there are no vulnerabilities of this type, but no test will be able to increase the confidence level to 100 %.

Conclusions

In this work we have done all the tasks we set out to do from the beginning. We have explained what code analysis is in a software project. We have explained what it is used for and the advantages that the use of this type of tools means for the future when maintaining the software application.

We have shown how it is necessary to analyze applications to avoid introducing not only bugs, but vulnerabilities as well. We have carried out different types of tests to find different types of bugs and vulnerabilities. We have insisted on how good programming practices and good organization can help to reduce these types of errors, although they will still exist.

Referencias

[1] Robert G. Wilson Leslie B.; Clark. *Comparative Programming Languages*. Addison Wesley, 1993.

[2] Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[3] Pieter Hooimeijer y Westley Weimer. «Modeling Bug Report Quality». En: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, págs. 34-43. ISBN: 9781595938824. DOI: 10.1145/1321631.1321639. URL: https://doi.org/10.1145/1321631.1321639.

[4] Concord Consortium. *The Concord Consortium*. URL: https://concord.org/.

[5] Blake Loring, Duncan Mitchell y Johannes Kinder. «ExpoSE: practical symbolic execution of standalone JavaScript». En: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 2017, págs. 196-199.

[6] WHK Bester, Cornelia P Inggs y WC Visser. «Test-case generation and bug-finding through symbolic execution». En: *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*. 2012, págs. 1-9.

[7] Shashank Gupta. «Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art». En: *International Journal of System Assurance Engineering and Management* (2017). DOI:

10.1007/s13198-015-0376-0. URL: https://doi.org/10.1007/s13198-015-0376-0.

1. Introducción

En las últimas décadas la cantidad de líneas de código que se crean cada año ha aumentado de manera exponencial. Este aumento del número de líneas de código sumado al hecho de que es mucho más costoso la corrección de errores una vez se ha lanzado al mercado la aplicación [1] hizo que las compañías desarrolladoras de software buscasen maneras de poder detectar estos errores en la fase de producción, de tal manera que se redujesen los costes y también con el objetivo de buscar lanzar una versión final más pulida y refinada mejorando así también la opinión de sus clientes.

De esta manera las empresas más grandes comenzaron a desarrollar técnicas y programas que permitiesen a los desarrolladores ejecutar análisis de sus programas y así poder ver, de una manera rápida y clara, si el funcionamiento de la nueva aplicación es el esperado o por el contrario hay algo que no funcione de manera correcta [2].

Hoy en día todas las empresas, grandes y pequeñas utilizan algún método de análisis de código como una manera de reducir sus costes en la fase de producción.

Realizar este tipo de análisis es crucial para las grandes empresas de software como Microsoft o Apple, cuyos últimas versiones de sistemas operativos cuentan cada uno con más de 80 millones de líneas de código [3]. Pero no son únicamente este tipo de empresas las que realizan este tipo de análisis. Empresas cuyo producto final no es software pero sí que cuentan con una gran cantidad de líneas de código también ponen a prueba su software.

Empresas de defensa como Lockheed Martin, creador del F-35, que lleva más de 10 millones de líneas de código [4] o empresas automovilísticas, ya que los coches modernos pueden superar las 100 millones de líneas de código [5].

Por último, el análisis estático no permite únicamente detectar y arreglar fallos de funcionalidad en el código, sino que permite también encontrar errores de seguridad, algo que cada vez resulta más importante frente a los cada vez mayores ataques a páginas web y empresas.

1.1. Coste del análisis de software

Lo más importante que hay que comprender a la hora de realizar cualquier tipo de análisis en cualquier tipo de aplicación software es que la certeza absoluta es inalcanzable, al menos únicamente con los tests. Esto significa que a la hora de realizar el análisis, el analista o tester no puede afirmar rotundamente con los resultados obtenidos la existencia o no de un bug, fallo o error.

A la hora de analizar los resultados, pueden suceder dos cosas, un resultado positivo, el análisis ha detectado algo inusual o raro; o puede devolver un resultado negativo.
Si el análisis devuelve un resultado negativo, significa que no ha encontrado nada inusual en la aplicación para la prueba específica que se estaba llevando a cabo. Sin embargo, este resultado negativo no permite afirmar a ciencia cierta y con total seguridad que ese error o fallo no se encuentra en el código. Puede ser que el análisis sea incompleto o no sea lo suficientemente exhaustivo

como para poder detectar este fallo.
Un análisis perfecto no sólo es imposible (errores humanos a la hora de diseñar las pruebas e incluso las herramientas que se utilizan para los análisis hace que sea imposible hallar todos los errores en todas las situaciones), sino que es también inasumible por parte de la empresa debido a lo exorbitado que supondría el coste de un análisis tan profundo.
De esta manera, el obtener un resultado negativo no te proporciona una certeza absoluta, sino que te proporciona una certeza mayor de que ese error no se da en el código. A más pruebas mayor certeza.

Por el contrario, uno podría pensar que un resultado positivo proporciona una certeza absoluta de un fallo o error, y aunque la lógica pueda parecer correcta, esto no es así, ya que un resultado positivo tan solo indica la existencia de algo inusual.
Para poder afirmar que se trata de un error, el analista de software debe proporcionar los resultados de sus pruebas a los desarrolladores y ellos esta vez sí, confirmar si finalmente se trata de un bug o no.
Hay que pensar que aunque algo en el código sea inusual, no siempre es un fallo. Puede ser que los desarrolladores los hayan introducido deliberadamente, y que el analista a la hora de crear la prueba no haya comprendido correctamente el funcionamiento de esa parte del código. Puede ser también que esa parte del código esté aún en desarrollo, o que esté comentada porque esa parte no va a introducirse aún en la versión final.
En definitiva, un analista no es capaz de afirmar simplemente con un resultado positivo que haya encontrado un bug en el código.

Como ya hemos dicho anteriormente, el coste del análisis del código es algo

que debe tenerse en cuenta a la hora de gestionar y planificar el proyecto desde el principio. El análisis de código permite aumentar la calidad del producto a la vez que se reduce el coste que supondría el mantenimiento del producto para arreglar aquellos errores que no se habrían detectado si no se hubiese llevado a cabo la fase de análisis.

Por ello es esencial que el director o el órgano directivo responsable del proyecto tenga esto en cuenta para poder calcular la cantidad de análisis necesario para conseguir una buen equilibrio entre el coste en calidad y el coste monetario, tal y como se muestra en la figura 1.

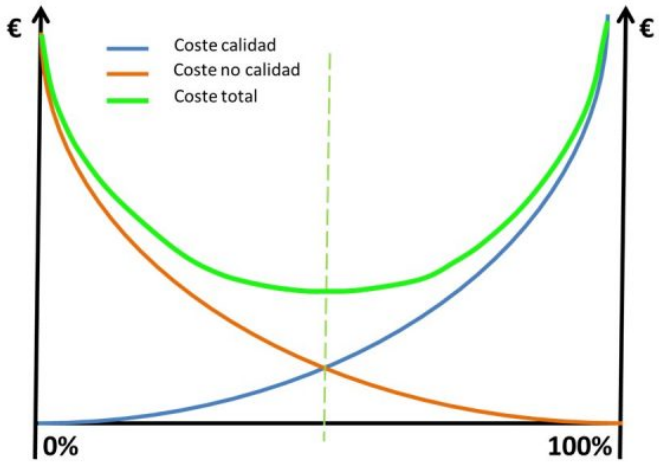


Figura 1: Gráfico que detalla la relación entre los costes de realizar análisis y no realizarlo en la fase de producción. El objetivo es encontrar un punto intermedio en el que se pueda asumir tanto el coste de calidad y el coste residual de no calidad.

2. Estado de la cuestión

Una de las maneras que utilizaron las empresas para buscar y solucionar este tipo de fallos fue lo que se conoce como análisis estático. Este tipo de análisis permite analizar una aplicación sin tener que ejecutarla, por lo que es muy usada en el ámbito de la ciberseguridad para detectar posibles códigos o archivos maliciosos [6].

La que es considerada como la primera herramienta de análisis estático es la herramienta Lint [7]. Esta herramienta fue creada por los Laboratorios Bell en 1977 y se trataba de una herramienta que realizaba análisis simples en aplicaciones programadas en C. La evolución de esta herramienta ha permitido el desarrollo de muchas otras que utilizan distintos tipos de análisis para llevar a cabo el análisis estático de las aplicaciones. Algunos de estos tipos son:

1. Análisis por control de flujo: utilizado para determinar las estructuras de control de un programa. Este se representa a través del *Control Flow Graph* o CFG. Este CFG se encuentra en la mayoría de los programas dentro del propio código de la aplicación. Este término fue introducido por Neil D. Jones y Olin Shivers [8].

2. Análisis por control de datos: el funcionamiento de este método es muy parecido al anterior. Este método estudia como se van trasladando los datos a través de la aplicación y la manera de observarlos es a través de otro gráfico. En este caso este gráfico es conocido como *Data Flow Graph* o DFG por sus siglas en inglés [9].

3. Análisis por control de escape: la mayor diferencia que tiene este método respecto al resto es que este está pensado para descubrir fallos que ya hayan escapado fase de producción y testing, mientras que los otros fueron diseñados para analizar aplicaciones en estas fases. El análisis por control de escape se realiza para «asegurar un continuo progreso se realiza en el producto software y en los procesos de producción y testing.» Esto se llevará acabo «mediante el análisis de defectos que hayan escapado las fases previas y preveniendo que escapes similares ocurran en el futuro» [10].

La cantidad de herramientas disponibles en el mercado es amplia. Pese a esto, no siempre es posible encontrar una que se ajuste a tus requisitos. Por este motivo, muchas empresas deciden crear y desarrollar su propia herramienta. Es el caso de FaceBook por ejemplo. Esta empresa ha desarrollado su propia herramienta llamada Pysa (acrónimo de Python Static Analyzer, o analizador estático de python en español). Esta herramienta permite «analizar flujos de datos, rastrear el uso de datos externos sin procesar en llamadas y realiza rondas iterativas de análisis para construir resúmenes» [11].

2.1. Desarrolladores y «testers»

Los desarrolladores son los ingenieros informáticos que se dedican a la creación, mantenimiento y mejora de programas y aplicaciones software. Su función es la creación de código ejecutable que cumpla con los requisitos del proyecto. En la primera fase, los desarrolladores se centran en la creación del código. Después las fases de mejora y mantenimiento pueden ser simultáneas o no.

Pese a que los desarrolladores realizan pruebas en su propio código, para asegurarse de que funciona correctamente y que cuenta con la funcionalidad deseada, los desarrolladores no se encargan de probar a fondo el código que han creado para comprobar que cumpla con todos los requisitos.

Los encargados de realizar este tipo de pruebas a fondo son siempre los denominados «testers» o ingenieros de pruebas en español. Los «testers» son otros profesionales informáticos que no han participado en la creación del código, por lo que son un grupo independiente al de los desarrolladores.

Existen varias razones por las cuales los ingenieros de pruebas son personas distintas a los desarrolladores [12], siendo las más importantes las que se describen a continuación:

1. Perspectiva imparcial: los ingenieros de pruebas proporcionan una visión externa e imparcial del código que se ha de analizar. Gracias a no estar familiarizados con el código, los «testers» pueden detectar errores que los desarrolladores pasarían por alto debido a su familiaridad con el código.

2. Enfoque en la calidad: mientras que los desarrolladores están más centrados en la agregación de nuevas funcionalidades a la aplicación, los ingenieros de pruebas se centran en la estabilidad y calidad del código. Esto permite que cada equipo se especialice en su área de trabajo. Por ejemplo, los ingenieros de pruebas se centran en la creación de pruebas exhaustivas, incluyendo casos específicos y escenarios límite para asegurarse de que el producto funcione bien en dichos casos.

3. Independencia del proceso de desarrollo: mantener ambos equipos se-

parados permite separar también claramente las responsabilidades de cada uno de los equipos. Además, esta separación también permite que el proceso de prueba no esté influenciado por el proceso de desarrollo o por las relaciones políticas o personales entre los miembros de cada equipo.

Como ya hemos visto, es esencial que los ingenieros de pruebas no se mezclen con la programación de la aplicación. Por este motivo, una vez los ingenieros de pruebas han finalizado toda las pruebas, las documentan y se la envían a los desarrolladores. Tras recibir la documentación enumerando todos los posibles fallos que se han detectado los desarrolladores los revisan para comprobar si efectivamente se tratase de un fallo y en ese caso corregirlo. Pero nunca serán los propios ingenieros de pruebas los que se encarguen de la corrección de dichos errores.

3. Motivación

El uso de herramientas de análisis de código es muy común en empresas grandes y medianas. Empresas como Microsoft, Apple, Facebook o Amazon llevan acabo análisis exhaustivos de sus productos debido a la cantidad de líneas de código que tienen sus proyecots o negocios (millones). Sin embargo, el uso de estas herramientas no es tan común en empresas pequeñas o en aplicaciones de tipo *open-source*.

Hoy en día muchos desarrolladores realizan aplicaciones de este tipo ya sea por investigación o por simple altruismo. También en este mundo digital en el que vivimos docentes en todo el mundo buscan aprovechar la tecnología

para utilizarla en sus clases y la mayoría de las aplicaciones orientadas a la docencia son de este tipo.

Por este motivo vamos a realizar este trabajo en el cual estudiaremos una aplicación *open-source* orientada a la docencia para poder descubrir aquellos errores que los desarrolladores hayan pasado por alto.

4. Objetivos del proyecto

En este trabajo vamos a realizar un análisis de una aplicación web JavaScript mediante una herramienta de análisis estático. Los objetivos que tenemos se enumeran a continuación:

1. Estudio y comprensión del análisis estático: llevamos un estudio del funcionamiento del análisis estático desde sus primeras herramientas hasta ahora.

2. Elección de una herramienta con la que poder realizar el análisis estático: estudiamos diversas herramientas para poder realizar análisis estático y poder elegir la que mejor se adaptase a nuestros objetivos.

    a) Aprender y entender el uso de la aplicación escogida.

    b) Estudiar las diferentes técnicas y estudios que podíamos realizar con esta herramienta.

    c) Comprender las respuestas y análisis que devuelve la aplicación para así poder analizar los resultados.

3. Analizar todos los errores por separado y en conjunto.

4. Concluir como de necesario es llevar a cabo algún tipo de análisis en aplicaciones de este tipo.

5. Página a analizar

El objetivo de este proyecto era el poder analizar una página web basada en JavaScript. Lo que buscábamos era el poder analizar una aplicación que fuese de código abierto y que estuviese orientada a la educación, ya que serían este tipo de aplicaciones las que se estudiarían en futuros proyectos.

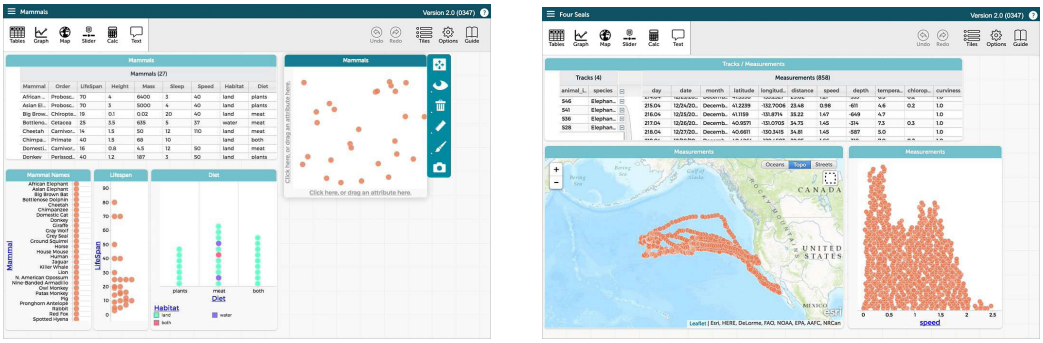
De esta manera escogimos la aplicación de CODAP (Common Online Data Analysis Platform), desarrollada por Concord Consortium [13].

Esta aplicación es un «software educacional gratuito para análisis de datos». Según los autores, esta aplicación está orientada a desarrolladores y alumnos de edades entre 11 y 19 años de edad (cursos 6 a 14 en Estados Unidos).

Esta aplicación permite analizar datos con facilidad, permitiendo la creación de distintas gráficas y ordenar los datos en función de los atributos de los datos. Permite la creación y descubrimiento de patrones y conexiones entre los datos. También permite realizar análisis sobre texto escrito, permitiendo realizar un conteo de palabras o de signos de puntuación. De esta manera puede ayudar a los alumnos a identificar palabras que usen demasiado, para poder cambiarlas por sinónimos o también identificar el mal uso de los signos de puntuación, ya sea por su poco o excesivo uso. Estos tipos de errores de los alumnos es algo complicado de analizar y corregir por parte del maestro,

debido a la individualización a la que tiene que llegar con cada alumno para corregir este error. Sin embargo, con el uso de esta aplicación el propio alumno puede observar y trabajar para corregirlo por su cuenta.

CODAP cuenta también con más herramientas que pueden ayudar a los alumnos al aprendizaje en interiorización del temario. Pese a que todas las herramientas y posibilidades que hemos explicado ya se pueden realizar, y desde hace tiempo, con otras aplicaciones, en lo que más destaca esta aplicación es en lo gráfica que resulta la aplicación, lo cual permite a los alumnos utilizar y comprender las herramientas con mucha más facilidad que con otras herramientas que haya disponibles.



(a) (b)

Figura 2: (a) Muestra el análisis de datos mediante el uso de gráficas. (b) Muestra el análisis de datos apoyándose en mapas para ayudar a la visualización de los datos

6. Recursos a emplear

Al principio, cuando estuvimos estudiando distintas aplicaciones para poder realizar análisis simbólico a aplicaciones JavaScript, estudiamos distintas

herramientas para ver cuál era la más óptima, pero en todas ellas había algo que no nos permitía escoger esa como la opción.

La primera herramienta que estudiamos fue *ExpoSE* [14] una herramienta gratuita que se puede acceder desde GitHub. Esta herramienta estaba disponible para Linux. Se consiguió instalar la herramienta y realizar pruebas a una aplicación de ejemplo que incluía la propia aplicación. Sin embargo, a la hora de probar esta herramienta con nuestra aplicación comenzó a dar errores y resultados contradictorios e incompletos.
Tras pasar varias semanas estudiando por qué podía estar sucediendo esto, nos pusimos en contacto con el autor de la herramienta. El autor nos explicó que *ExpoSE* no estaba diseñada para realizar análisis en aplicaciones diseñadas para ejecutarse en un navegador, es decir, para analizar aplicaciones web, por lo que tuvimos que descartar esta herramienta y pasar a otra.

Otra herramienta que también estudiamos fue *Artemis* [15]. Esta herramienta estaba disponible para sistemas Linux 14, por lo que lo instalamos en una máquina virtual con un Linux antiguo. Al contrario que ExpoSE, esta aplicación sí que podía realizar análisis en aplicaciones web. Sin embargo, Artemis se centraba únicamente en casos de seguridad. Debido a que en el proyecto buscábamos realizar un análisis general decidimos descartar el uso de esta herramienta ya que nos limitaba mucho a la hora de analizar nuestra aplicación.

Estas dos fueron las dos más prometedoras que encontramos. En el resto de aplicaciones encontrábamos limitaciones parecidas a las anteriores o algunas nuevas, como por ejemplo que no realizasen realmente un análisis dinámico.

También se estudiaron herramientas de pago. Este tipo de aplicaciones eran las más completas y sencillas de utilizar, por lo que hubiesen sido un activo muy importante para poder realizar este proyecto. Finalmente se abandonó esta idea porque el objetivo es analizar código abierto y gratuito. Uno de los objetivos es demostrar que existen herramientas que los desarrolladores de este tipo de aplicaciones pueden usar sin necesidad de tener un presupuesto para obtener una licencia de este tipo de software. Queremos demostrar que es posible el desarrollo de una aplicación software de manera gratuita que tenga todos los pasos o procedimientos que cualquier desarrollo de software profesional de una empresa tuviese, incluyendo la parte de análisis de código.

Tras esto decidimos cambiar a una herramienta de análisis estático. La herramienta que escogimos fue *CodeQL* de GitHub. Esta herramienta convierte la aplicación a analizar en una base de datos a la cual puedes realizar consultas para realizar el análisis [16].

Con esta herramienta realizaremos el análisis de la aplicación centrándonos principalmente en errores de seguridad y de uso (usability en inglés).

6.1. Análisis simbólico y estático

Al no encontrar ninguna herramienta que fuese lo suficientemente adecuada para la realización del proyecto tuvimos que cambiar de método de análisis. Esto suponía también cambiar la manera en la que queríamos realizar el análisis y obtener los resultados, ya que los dos métodos de análisis presentan diferencias importantes entre ellos.

El análisis de código simbólico ,comenzó a desarrollarse en las décadas

de 1960 y 1970. Uno de los mayores hitos fundamentales en esta área fue el desarrollo del lenguaje de programación LISP (LISt Processing) en 1958 por John McCarthy [17]. Este lenguaje fue el primero que utilizó estructuras de datos basadas en listas, lo que suponía una gran ventaja para manipulaciones simbólicas.

Desde entonces el análisis ha mejorado y evolucionado mucho a lo largo de los años gracias al desarrollo de técnicas de verificación de programas. Hoy en día el análisis simbólico se utiliza en una gran variedad de aplicaciones de informática y de software, como en la verificación de programas u optimización de código, entre otras.

Sin duda, por lo que más se caracteriza el análisis simbólico es por que no tiene la necesidad de que el usuario introduzca parámetros iniciales. Esto es debido a que trabaja con símbolos y expresiones abstractas, en vez de con valores concretos.
Esto permite al análisis simbólico poder explorar todas las posibles rutas de ejecución de una aplicación o programa. Por este motivo intentamos realizar este tipo de análisis, ya que es muy útil a la hora de tener que analizar una aplicación ya finalizada.

Por otro lado, el análisis estático ya lo hemos introducido en secciones anteriores. Este tipo de análisis es una etapa habitual en cualquier proyecto de desarrollo software profesional, por lo que se encuentra siempre en constante cambio y modificación, especialmente para que cada empresa lo adapte a sus necesidades y a las de su código.

La mayor ventaja del análisis estático es que no necesita ejecutar el código

que vaya a analizar. De esta manera su uso es muy habitual en entornos de ciberseguridad, para poder analizar si un código contiene alguna amenaza o código malicioso, ya que gracias al análisis estático puede ser descubierto antes de que ese código maligno se ejecute e infecte la máquina.

Resumiendo, ambos tipos de análisis son diferentes, por lo que son complementarios. Mientras que el análisis simbólico se centra en estudiar el comportamiento de un programa, el análisis estático se centra más en la revisión del código. Normalmente, el análisis estático se utiliza en la fase de producción, para ir probando el código a medida que se genera y el análisis simbólico se utiliza más adelante para probar funcionalidades completas de la aplicación que se esté desarrollando.

6.2. Instalación y uso de la herramienta

La versión que se decidió utilizar fue la extensión de CodeQL para el editor de código de Visual Studio. Escogimos esta versión porque nos pareció la manera más sencilla de utilizar esta herramienta y además es la versión que recomiendan los propios autores. Para ello nos instalamos dos extensiones, la extensión de CodeQL y la extensión de CodeQL Agent. Aunque la segunda no es esencial, sí que es recomendable utilizarla para llevar a cabo los estudios de la aplicación que se vaya a analizar.

Otra ventaja de utilizar la extensión de CodeQL es que no hubo que descargarse ningún instalador, lo que nos evitó problemas de compatibilidad con otros programas o lenguajes ya instalados en el equipo y que sí que nos ocurrió con otras aplicaciones que estudiamos antes de decidirnos por esta.

Sin embargo, aunque no hubo que instalar ningún ejecutable, sí que debimos instalar la carpeta con los comandos necesarios para poder transformar la aplicación a estudiar, es decir, CODAP, para poder realizar las llamadas contra su código.

CodeQL no permite el análisis de páginas web directamente. Es decir, no permite introducir un link a la página web como parámetro y de esa manera realizar el estudio.
Por este motivo, tuvimos que descargar el código fuente de la aplicación para poder realizar la transformación a base de datos y así poder importarlo a CodeQL y realizar las consultas.

```
C:\Users\nicol\Personal\Chicago\TFM\Prueba\codeql>codeql database create codap_db --language=javascript --source-root Codap
Initializing database at C:\Users\nicol\Personal\Chicago\TFM\Prueba\codeql\codap_db.
Running build command: []
```

Figura 3: Comando para la creación de una base de datos. Necesita tres parámetros: la carpeta de destino de la base de datos; el lenguaje de la fuente; y la carpeta con el código fuente.

Tras esto solo quedaba importar la base de datos recién creada en CodeQL y comenzar a realizar consultas.

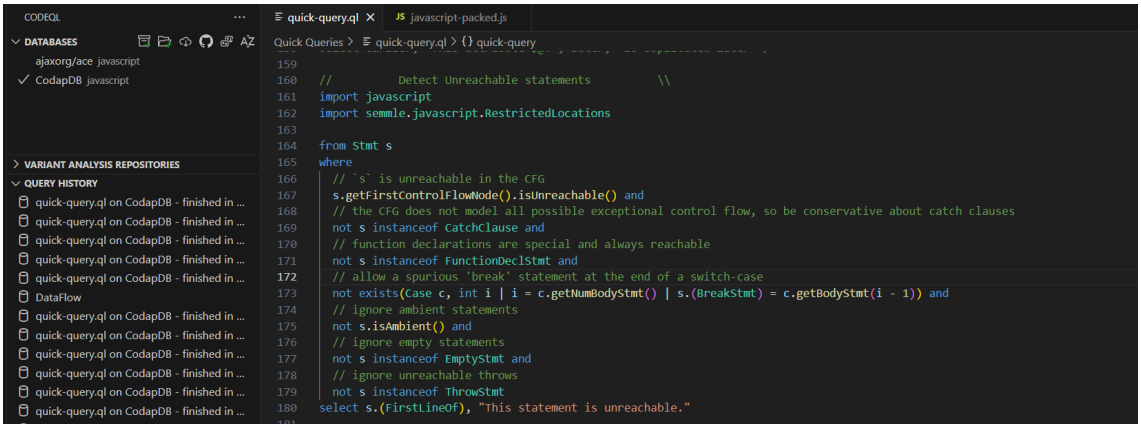


Figura 4: Resultado de instalar la base de datos en CodeQL. Se puede observar también la base de datos de prueba, llamada ajaxorg/ace, que ofrce GitHub para aprender y practicar con su herramienta

7. Resultados

En esta sección nos centraremos en presentar y explicar las pruebas que hemos realizado así como sus resultados. Explicaremos por qué decidimos llevar a cabo esa prueba en concreto y mostraremos mediante imágenes tanto el código utilizado para realizar dicha prueba. También explicaremos que efectos pueden tener los fallos en la aplicación y por qué sería recomendable arreglarlos.

También, en el caso en el que el error sea reproducible desde la propia aplicación se mostrarán mediante imágenes para que sea más claro.

7.1. Resultados positivos

Las pruebas que explicaremos en esta apartado son aquellas en las que detectamos posibles fallos. Estas pruebas son las que en un futuro documentaremos y enviaremos a los desarrolladores de CODAP para, en caso de que lo consideren oportuno, las corrijan.

En estas pruebas realizamos pruebas tanto de experiencia del usuario como de seguridad.

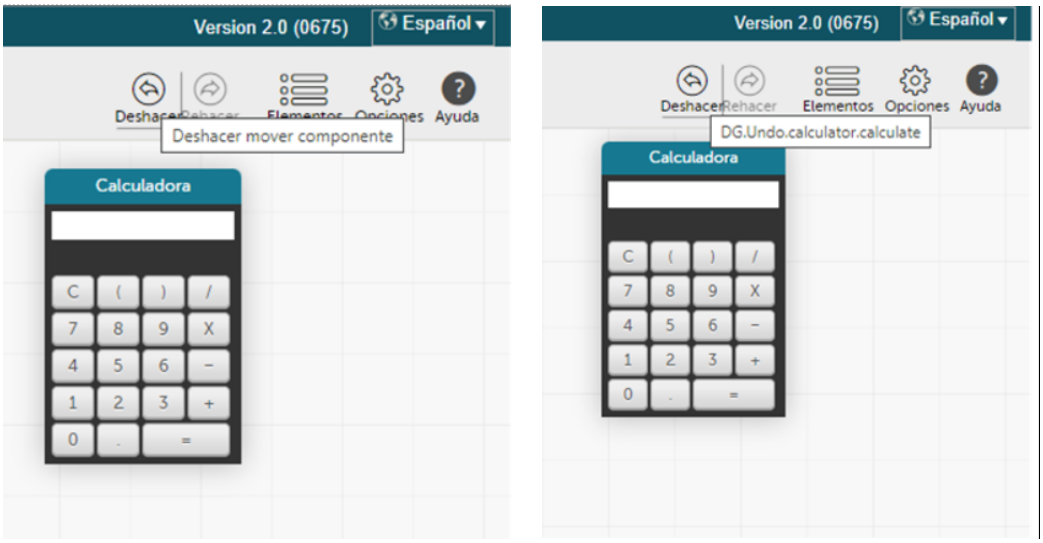
7.1.1. Botones con una descripción inadecuada

Una de las ventajas de CODAP es que está disponible en hasta 16 idiomas diferentes. Por este motivo, los botones deben mostrar una descripción adecuada en cada uno de los idiomas para que sus usuarios puedan aprovechar al máximo sus funcionalidades.

Durante la investigación de la herramienta, y haciendo pruebas con CodeQL, nos percatamos que bajo ciertas circunstancias, estos botones no daban una correcta descripción.
Para probar esto no utilizamos únicamente CodeQL, sino que realizamos un pequeño programa en Python para ayudarnos.
Lo que hicimos fue sacar con CodeQL, todas las descripciones y sus respectivas llamadas. Posteriormente, con el programa de Python buscamos aquellas llamadas que no tenían hijo, y con lo cual no podían mostrar una descripción correcta.

Este error se detectó en los botones de «Redo» y «Undo», o rehacer y deshacer en español. Estos botones mostraban una descripción en la que

relataban rehacer o deshacer la última acción. Sin embargo, bajo ciertas circunstancias, estos botones no mostraban descripción, si no que mostraban directamente la llamada al no encontrar la descripción.



(a) (b)

Figura 5: (a) Ejemplo de un botón con una descripción adecuada. (b) Ejemplo de un botón con una descripción inadecuada.

Como se puede ver en la figura 5, no todas las acciones tenían descripciones inadecuadas. Por este motivo lo que hicimos fue sacar todas las acciones posibles con esos dos botones y sacar todas las descripciones con CodeQL. Tras esto, guardamos las 4 tablas (dos por cada botón) como csv para así poder compararlas fácilmente con Python.

En Python cargamos las cuatro tablas y comparamos la de cada botón para así poder encontrar las acciones que no contasen con una descripción y poder imprimir esas acciones por pantalla.

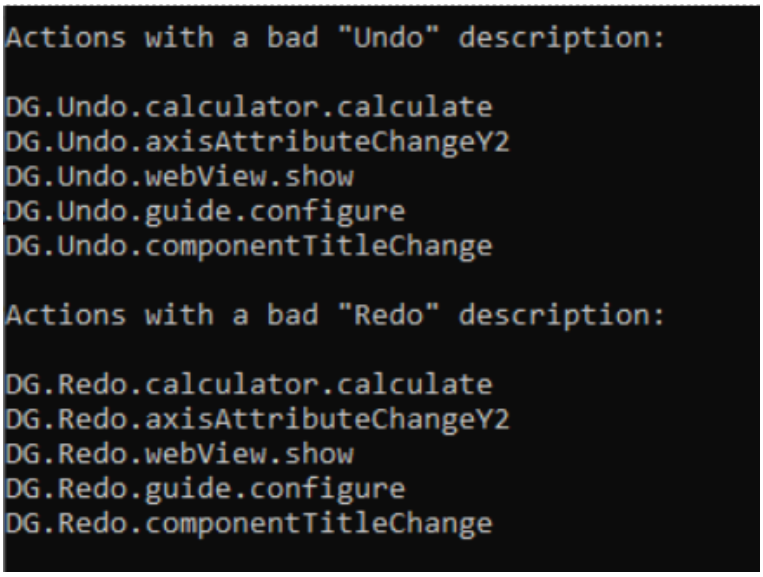


Figura 6: Acciones que no cuentan con una descripción para sus correspondientes botones.

En la imagen 6, podemos observar como en el caso de ambos botones son las mismas acciones las que no cuentan con una descripción adecuada.

Aunque es cierto que este error no influye en la aplicación directamente, sí que influye en cómo los usuarios utilizan y se sienten al usar la herramienta. Como se puede ver, cuando la acción no cuenta con una descripción se pone el comando de dicha acción en inglés. Recordemos que CODAP está disponible en varios idiomas y sus usuarios no tienen por qué saber inglés o interpretar esos comandos, por lo que sería conveniente corregir este problema.

7.1.2. Código imposible de ejecutar

Otra prueba que hicimos con CodeQL era para detectar código que fuese imposible de ejecutar.

Normalmente este tipo de pruebas se realizan con análisis simbólico, ya que al recorrer el código dando valores simbólicos puede detectar fácilmente si hay alguna parte del código a la que sea imposible acceder.

Sin embargo, aunque fuese más fácil con análisis simbólico, CodeQL permitía realizar también esta prueba. El resultado que obtuvimos fue la mayoría simples variables que no tenían ninguna función. Sin embargo sí que hubo una parte de este código imposible de ejecutar que nos llamó la atención.

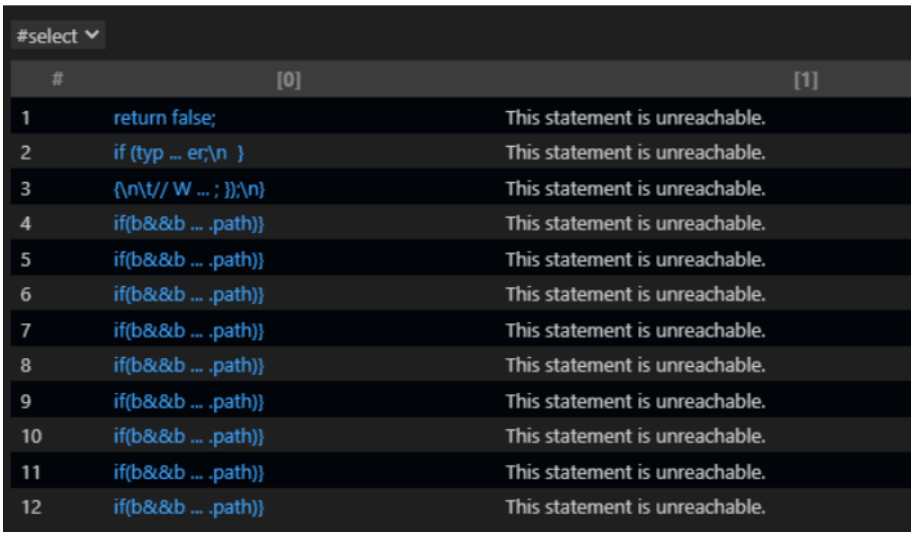


Figura 7: Resultados con todas las partes del código imposibles de ejecutar.

Este código hacía referencia a los sonidos de la aplicación. Actualmente la aplicación no cuenta con ningún tipo de sonido, y se debe a que todo el código que se refiere a los sonidos no se puede ejecutar porque siempre antes de la función «*PlaySound()*» había un «*if True {skip}*». Esto hacía que siempre se saltase la línea en la que se llamaba para reproducir el sonido. En la imagen 8 se puede observar esto.

```
playSound: function(a) {
    var b = this.sounds[a];
    if (true) {
        return
    }
    if (b && b.path && !b.audio) {
        b.audio = new Audio(b.path)
    }
    if (b.audio) {
        if (window.chrome) {
            b.audio.load()
        } else {
            b.audio.currentTime = 0
        }
        b.audio.play()
    }
},
```

Figura 8: Una de las funciones encargadas de reproducir el sonido con un «if(true)» justo antes.

Estos códigos imposibles de ejecutar (tanto el de los sonidos, como en el de las variables) no tienen ningún impacto directo sobre la propia aplicación, ya que los sonidos no son indispensables en esta aplicación.

Sin embargo, este tipo de códigos pueden suponer un problema en el futuro. Este tipo de códigos, que tienen alguna manera de evitar que se ejecuten, pueden volverse ejecutables de manera indeseada en alguna versión futura en la que, sin querer, se rompa esta protección. Es decir, este tipo de códigos suponen una mala práctica de cara al mantenimiento de la aplicación, ya que es algo que los desarrolladores y los ingenieros de prácticas no buscarán, ya que es algo que no existía anteriormente y no es una función nueva que hayan introducido los desarrolladores. O al menos no intencionalmente.

7.1.3. Variables que no se usan

En programación, es una práctica habitual la creación da variables antes de su uso, así como la asignación de algún valor inicial antes de donde se vaya a utilizar. Incluso, dependiendo del lenguaje, esto puede ser hasta necesario.

Sin embargo, si no se lleva una buena organización y se siguen unas buenas prácticas, puede ocurrir que esas variables que se han creado se sobrescriban antes de usarse, por lo que la primera instancia se volvería inútil. También puede pasar que se cree una variable que no se llegue a usar nunca al final.

Figura 9: Muestra de parte de las variables que no se usan en el código de CODAP.

Algo que llama la atención de esta prueba, es la cantidad de variables que se han encontrado que no se usen. Como se puede ver en la imagen 9 la cantidad de variables encontradas son más de 23.500, lo cual es sorprendente incluso para una aplicación tan compleja y completa.

Esto es una mala práctica no solo porque ensucie el código, haciendo que este sea más complicado de comprender, y por lo tanto, dificultando cualquier tarea de mantenimiento o actualización de la herramienta, sino que, al igual que en error anterior, puede provocar funcionalidades indeseadas.

La existencia de variables que no cumplan ninguna función pueden provocar que al realizar las tareas de mantenimiento estas expresiones tengan efectos que no sean intencionales y perturbe así el servicio que da la aplicación. Al introducir código en una zona, puede usarse una variable que estuviese previamente instanciada y podría provocar que el código utilizase ese valor, creando un fallo en la funcionalidad del sistema.

Sin embargo, este problema puede suponer también fallos de seguridad. Al tener variables sin utilizar, puede ser que en un futuro ese tipo de variables cambien, pudiendo provocar problemas de seguridad de overflow. Si un programa utiliza, por ejemplo, enteros normales (32 bits) y más tarde, por requisitos del sistema se cambian todos los enteros a otro tipo (con más bits), los que estaban en desuso, si no se han modificado, pueden provocar problemas de overflow, ya que los nuevos números con los que trabaja la aplicación podrían no caber en un entero normal.
Esto es algo que un atacante podría utilizar para afectar al comportamiento correcto del programa. También podría usarse para realizar un acceso no autorizado a la memoria. Estos accesos a la memoria no autorizados pueden suponer desde bloqueos del programa hasta fugas de información sensible. Pese a que CODAP es una herramienta que se accede a través de un internet, también puede descargarse y acceder a la herramienta en local utilizando un navegador, por lo que puede suponer incluso una amenaza para el equipo

del usuario.

7.1.4. Expresiones sin efecto

A la hora de programar una aplicación web es importante no definir únicamente cómo ha de comportarse el programa, es decir, su funciaonalidad. También es importante definir como se va a comportar visualmente en el navegador. El tamaño de cada componente y como se va a mostrar en función del dispositivo del usuario es esencial para una buena experiencia de los usuarios.

A la hora de estudiar las expresiones que establecen el comportamiento visual de la aplicación nos sorprendió ver como había numerosas expresiones que no tenían efecto. Nos sorprendió debido a que visualmente la herramienta es muy atractiva, y durante su uso, no hemos encontrado ningún problema visual.

Figura 10: Muestra de parte de las que no tienen ningún tipo de efecto en el código de CODAP.

A riesgo de sonar repetitivo, todo aquel código que no tenga una funcionalidad en la aplicación debería ser eliminado para limpiar el código y evitar futuros problemas a la hora de mantener el software.

7.1.5. Etiquetas incorrectas

Aunque el foco principal lo hayamos puesto en analizar el código JavaScript (en lo que está programada la aplicación), pero al tratarse de una aplicación JavaScript creada para usarse desde un navegador también tiene una parte de HTML. Por este motivo realizamos también pruebas contra la parte de HTML.

En esta prueba probábamos las etiquetas HTML, para comprobar que no hubiese nada raro en ellas.
Para esta prueba comprobamos que todas las etiquetas fuesen correctas (es decir que no hubiese ninguna que no exista o que tenga una errata) y también nos aseguramos que todas se cerraban correctamente.
Y ese fue justo el error que nos encontramos, ya que había etiquetas que no estaban correctamente cerradas. Esto puede provocar fallos en el HTML o incluso vulnerabilidades de seguridad que pueden ser usados por los atacantes.

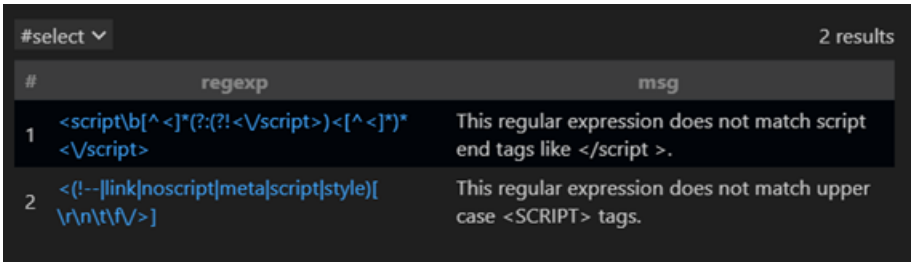


Figura 11: Etiquetas mal cerradas

7.2. Resultados negativos

Los resultados negativos son aquellos que no han devuelto ningún resultado después de lanzar la query. Esto significa que no hemos encontrado nada inusual en el código, por lo que no se reportan estas pruebas a los desarrolladores. Aunque como ya hemos comentado anteriormente, esto no significa que no exista ese tipo de error en el código, ya que no se puede realizar ninguna prueba que asegure esto al 100 %.

Pese a que el resultado de estas pruebas no las utilizaremos más adelante para darle feedback a los desarrolladores, son parte esencial del trabajo de análisis, ya que, al fin y al cabo, no sabes que pruebas van a dar un resultado positivo o un resultado negativo

7.2.1. Cross-Site Scripting (XSS)

El Cross-Site Scripting, también conocido como XSS, es una vulnerabilidad de seguridad en aplicaciones web que ocurren cuando un atacante inyecta código malicioso en un campo de texto de una aplicación web.

Este tipo de ataque se produce cuando una aplicación web no filtra de manera correcta los datos que introducen los usuarios en los campos de texto para luego mostrarlos en páginas web sin modificar los caracteres especiales. Este código malicioso será después ejecutado en el navegador de otro usuario.

Este tipo de vulnerabilidad está clasificada como grave, ya que puede permitir robar la información confidencial de otros usuarios, tales como cookies o credenciales de inicio de sesión.

7.2.2. Atributos duplicados

Todas aquellas personas que hayan hecho un mínimo de programación sabrán que las acciones de copiar y pegar son muy habituales, ya sea código de alguna página web o de uno mismo. Sin embargo el realizar este tipo de acciones pueden introducir algunos errores que no sean identificables fácilmente ya que pueden no afectar a la ejecución del programa o pueden no hacerlo en ese momento.

Uno de estos errores es tener código duplicado. Por este motivo estudiamos si *CODAP* contenía algún atributo que estuviese duplicado dentro del código, y pudimos comprobar que no era así.

De nuevo este error es del tipo que afecta al futuro de la aplicación, ya que dificulta su mantenimiento y lo hace más arriesgado.

8. Conclusiones

En este trabajo hemos explicado en qué consiste el análisis de código en un proyecto de software. Hemos explicado para qué se utiliza y las ventajas que el uso de este tipo de herramientas supone de cara a futuro a la hora de mantener la aplicación software.

En este trabajo hemos demostrado como es necesario el análisis de las aplicaciones para evitar la introducción no solo de errores, sino de vulnerabilidades también. Hemos llevado a cabo distintos tipos de pruebas en las que hemos obtenido tanto resultados positivos como negativos, los cuales se deberán entregar a los desarrolladores de la aplicación para que estudien y

corrijan aquellos errores que consideren necesarios.
Hemos insistido en cómo unas buenas prácticas de programación y una buena organización puede ayudar a reducir este tipo de errores, aunque seguirán existiendo.

El análisis de aplicaciones, tanto estático como dinámico, son y seguirán siendo fundamentales para los grandes proyectos de software, pero cada vez más estas herramientas deben llegar a aquellas empresas más pequeñas y a los desarrolladores de herramientas de tipo *open source*, no solo para que puedan mejorar sus aplicaciones, sino para que también puedan ayudar a hacer un internet cada vez más seguro para todos acabando con todas las vulnerabilidades posibles.

8.1. Trabajo futuro

En este trabajo hemos completado todas las metas que nos propusimos en un principio. Sin embargo, se pueden llevar a cabo más tareas relacionadas con el trabajo que hemos realizado.

Una posible tarea es la de entregar todas las pruebas que hayan dado positivo a los desarrolladores, con una documentación adecuada indicando como se ha descubierto el error y como se puede replicar. El contacto entre los desarrolladores y los verificadores debe ser constante, de manera que las tareas de corrección de los fallos sea lo más rápida y eficiente posible.

Otra posible tarea para continuar con el proyecto sería la de realizar un análisis simbólico de la aplicación. Pese a que nosotros finalmente descartamos realizar este tipo de análisis por falta de una buena herramienta, no

significa que, en un futuro, si se encuentra una herramienta adecuada no se pueda realizar este tipo de análisis.
Como ya hemos comentado, el análisis simbólico y el análisis estático son complementarios, por lo que realizar un análisis simbólico posterior al estático no puede hacer sino aumentar la calidad del código y del software analizado una vez se hayan reportado y corregido los fallos detectados.

Por último, una posible tarea sería la de generalizar las pruebas realizadas. En este trabajo nos hemos centrado únicamente en probar la aplicación de CODAP. Es por esta razón que la mayoría de pruebas no funcionarían en otra aplicación, aunque fuese similar a la de CODAP. Por este motivo, una tarea que sería muy útil es encontrar una manera de que las pruebas realizadas en este trabajo sirviese para cualquier aplicación que esté basada en Node.js (en lo que CODAP está basado). Esto sería una gran ventaja para aquellos desarrolladores que no tengan mucha experiencia en el análisis de código, de manera que puedan simplemente ejecutar queries ya creadas previamente y poder observar los resultados de una manera rápida y sencilla.
Esto también permitiría el poder realizar un estudio de varias aplicaciones diferentes a la vez, pudiendo sacar estadísticas comunes y pudiendo observar patrones que se repitan en distintas aplicaciones, hayan sido creadas por distintos desarrolladores o no.

Anexos

A. Relación del proyecto con los Objetivos de Desarrollo Sostenible (ODS)

Debido a la importancia social y económica que los ODS están adquiriendo en la actualidad, es importante ver cómo nuestro proyecto posibilita el desarrollo de estos objetivos. Que este proyecto pueda contribuir a desarrollar los ODS fue algo que tuvimos en cuenta desde el principio del trabajo.

Por tanto, vamos a reproducir las metas más concretas a las que puede contribuir el proyecto y a comentar en qué medida podría hacerlo.

1. *4.6 De aquí a 2030, asegurar que todos los jóvenes y una proporción considerable de los adultos, tanto hombres como mujeres, estén alfabetizados y tengan nociones elementales de aritmética*

   Como ya hemos comentado, CODAP es una herramienta que está orientada a la educación. Y esto no es casual, desde el principio del proyecto nuestro objetivo era realizar un análisis sobre software producido de manera abierta y que estuviese orientado a la educación. Por este motivo se eligió estudiar sobre CODAP.

   Herramientas como CODAP, que son gratuitas y se encuentran e varios idiomas, pueden favorecer sobre todo a aquellos países menos desarrollados en los que los colegios no se puedan permitir la adquisición de licencias para sus alumnos de cada uno de los software que requieren

para su educación. Estas herramientas pueden ayudar a los alumnos a comprender y asentar los conocimientos vistos en clase, mejorando la calidad y el nivel de su educación.

La existencia de este tipo de herramientas puede ayudar a cumplir con el Objetivo de Desarrollo Sostenible. Por eso, realizar un análisis a la aplicación para asegurar que funciona de manera correcta y que es segura para los alumnos es una parte fundamental en la creación de la herramienta.

2. *10.3 Garantizar la igualdad de oportunidades y reducir la desigualdad de resultados, incluso eliminando las leyes, políticas y prácticas discriminatorias y promoviendo legislaciones, políticas y medidas adecuadas a ese respecto*

   En muchos casos, la desigualdad en los ámbitos profesionales comienza con una importante y evidente desigualdad en el nivel de educación. Si no damos las oportunidades a aquellos más desfavorecidos para que puedan acceder a una buena educación, y que por lo tanto puedan acceder a mejores puestos y más especializados, jamás podremos acabar con la desigualdad en la que cada persona tenga igualdad de oportunidades de una manera real.

   Por este motivo, consideramos que el aportar una herramienta que haya sido analizada y verificada tanto en su nivel de seguridad como de funcionalidad, es esencial para, poco a poco, acabar con la desigualdad, especialmente en los países más pobres.

Asegurarnos que nuestros proyectos cumplen con los Objetivos de Desa-

rrollo Sostenible de la ONU es responsabilidad de todos para construir cada vez un planeta mejor.

Los países más desarrollados debemos dar ejemplo, realizando el mayor esfuerzo para la consecución de estos objetivos y mostrando el camino a aquellos países, que por los motivos que sean, aún no pueden llevar a cabo estos objetivos de manera independiente.

B. Código utilizado para las pruebas

Para no saturar el documento principal con imágenes de código que tampoco ayudan a la comprensión del proyecto y de los resultados obtenidos, hemos decidido poner en este segundo anexo todos los códigos que hemos utilizado.

Al igual que en la sección de los resultados, se añadirán las imágenes por cada prueba, incluyendo una breve explicación si se considerase necesario.

Botones con una descripción inadecuada

Como hemos comentado en la sección de resultados, en esta prueba no se utilizó únicamente CodeQL, sino que se creó un programa en Python para realizar una tarea que no era posible en CodeQL.

```
import javascript

from Property p, Property q
where
  p.getFile().getAbsolutePath() = "C:/Users/nicol/Personal/Chicago/TFM/C
  q.getFile().getAbsolutePath() = "C:/Users/nicol/Personal/Chicago/TFM/C
  q.getName().matches("DG.Redo.%") and
  p.getName() = "redoString" and
  p.getInit().toString().matches("\"DG.Red%")
select q.getName()
```

Figura 12: Código de CodeQL para obtener las llamadas a los botones y a sus descripciones

En la figure 12, se muestra el código necesario para sacar las llamadas a

los botones de, en este caso, del botón de rehacer. En este código buscamos cualquier función que comience con «DG.REdo.» para así poder identificar las funciones a las que hace referencia. Se repitió lo mismo modificando el código para coger las funciones y las descripciones del botón de deshacer.

Una vez habíamos obtenido las 4 tablas (funciones y descripciones para rehacer y deshacer), nos creamos un pequeño programa de Python para que comparase las llamadas de cada botón viendo si había alguna llamada al botón que no tuviese su correspondiente llamada a su descripción. Este progrma guardaba aquellas acciones que no tenían una descripción adecuada y al final las imprimía por pantalla.

La imagen 13 muestra este código al completo.

C: > Users > nicol > Personal > Chicago > TFM > Piton > CheckUndos.py

```python
import csv

with open("UndoDescriptions.csv") as fu:
    fichu = csv.reader(fu, delimiter=',')
    list1U = [row[0] for row in fichu]

with open("Undo.csv") as ju:
    ficheu = csv.reader(ju, delimiter=',')
    list2U = [row[0] for row in ficheu]

with open("RedoDescriptions.csv") as fr:
    fichr = csv.reader(fr, delimiter=',')
    list1R = [row[0] for row in fichr]

with open("Redo.csv") as jr:
    ficher = csv.reader(jr, delimiter=',')
    list2R = [row[0] for row in ficher]


badUndo = []
badRedo = []

for i in list2U:
    if i in list1U:
       x=0
    else:
        badUndo.append(i)

for l in list2R:
    if l in list1R:
       x=0
    else:
        badRedo.append(l)

print("\nActions with a bad \"Undo\" description:\n")
for k in badUndo:
    print(k)

print("\nActions with a bad \"Redo\" description:\n")
for m in badRedo:
    print(m)

print("\n")
```

Figura 13: Código de Python. Este código compara las acciones de los botones y las descripciones de dichas acciones. Si una acción no tiene una descripción la guarda para imprimirla al final por pantalla.

Código imposible de ejecutar

```
import javascript
import semmle.javascript.RestrictedLocations

from Stmt s
where
  // `s` is unreachable in the CFG
  s.getFirstControlFlowNode().isUnreachable() and
  // the CFG does not model all possible exceptional control flow, so be conservative about catch clauses
  not s instanceof CatchClause and
  // function declarations are special and always reachable
  not s instanceof FunctionDeclStmt and
  // allow a spurious 'break' statement at the end of a switch-case
  not exists(Case c, int i | i = c.getNumBodyStmt() | s.(BreakStmt) = c.getBodyStmt(i - 1)) and
  // ignore ambient statements
  not s.isAmbient() and
  // ignore empty statements
  not s instanceof EmptyStmt and
  // ignore unreachable throws
  not s instanceof ThrowStmt
select s.(FirstLineOf), "This statement is unreachable."
```

Figura 14: Código utilizado para analizar aquellas partes del código que no se pudiesen ejecutar.

Variables que no se usan

```
import javascript

from VarDef def, LocalVariable v
where v = def.getAVariable() and
    not exists (VarUse use | def = use.getADef())
select def, "Dead store of local variable."
```

Figura 15: Código utilizado para analizar aquellas variables que se inicializan pero luego no se llegan a usar o se sobreescriben sin que hayan sido utilizadas.

Expresiones sin efecto

```
import javascript

from Expr e
where e.isPure() and
  e.getParent() instanceof ExprStmt
select e, "This expression has no effect."
```

Figura 16: Código utilizado para analizar aquellas expresiones que no tienen ningún efecto en la aplicación y que por lo tanto sería conveniente eliminar.

Etiquetas incorrectas

```
private import semmle.javascript.security.regexp.RegExpTreeView::RegExpTreeView as TreeView
import codeql.regex.nfa.BadTagFilterQuery::Make<TreeView>

from HtmlMatchingRegExp regexp, string msg
where msg = min(string m | isBadRegexpFilter(regexp, m) | m order by m.length(), m)
select regexp, msg
```

Figura 17: Código utilizado para analizar aquellas etiquetas que fuesen incorrectas, ya sea por erratas o por no haberse abierto o cerrado correctamente.

Cross-Site Scripting

```
import javascript
import semmle.javascript.security.dataflow.DomBasedXssQuery
import DataFlow::PathGraph

from DataFlow::Configuration cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink,
  sink.getNode().(Sink).getVulnerabilityKind() + " vulnerability due to $@.", source.getNode(),
  "user-provided value"
```

Figura 18: Este código muestra el código que hemos ejecutado para buscar posibles vulnerabilidades de Cross-SiteScripting.

Atributos duplicados

```
import javascript

/**
 * Holds if `earlier` and `later` are attribute definitions with the same name
 * and the same value, where `earlier` appears textually before `later`.
 */
Quick Evaluation: duplicate
predicate duplicate(DOM::AttributeDefinition earlier, DOM::AttributeDefinition later) {
  exists(DOM::ElementDefinition elt, int i, int j |
    earlier = elt.getAttribute(i) and later = elt.getAttribute(j)
  |
    i < j and
    earlier.getName() = later.getName() and
    earlier.getStringValue() = later.getStringValue()
  )
}

from DOM::AttributeDefinition earlier, DOM::AttributeDefinition later
where duplicate(earlier, later) and not duplicate(_, earlier)
select earlier, "This attribute $@.", later, "is duplicated later"
```

Figura 19: Código utilizado para analizar el código en busca de posibles atributos duplicados.

Referencias

[1] Pieter Hooimeijer y Westley Weimer. «Modeling Bug Report Quality». En: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: Association for Computing Machinery, 2007, págs. 34-43. ISBN: 9781595938824. DOI: 10.1145/1321631.1321639. URL: https://doi.org/10.1145/1321631.1321639.

[2] Jernej Novak, Andrej Krajnc y Rok Zontar. «Taxonomy of static code analysis tools». En: (ene. de 2010).

[3] *¿Cuántas líneas de código hay en las aplicaciones de uso diario?* URL: https://www.proydesa.org/portal/noticias/1429-cuantas-lineas-de-codigo-hay-en-aplicaciones-de-uso-diario.

[4] Laurent Lagneau. *El Pentágono ha solucionado cinco deficiencias graves del F-35, pero ha encontrado cuatro nuevas.* 2020. URL: http://galaxiamilitar.es/el-pentagono-ha-solucionado-cinco-deficiencias-graves-del-f-35-pero-ha-encontrado-cuatro-nuevas/.

[5] *De línea de producción... a línea de código*. 2001. URL: https://www.hibridosyelectricos.com/coches/coche-electrico-tiene-100-millones-lineas-codigo-mas-caza-f35_39994_102.html.

[6] B. Chess y G. McGraw. «Static analysis for security». En: *IEEE Security and Privacy* 2.6 (2004), págs. 76-79. DOI: 10.1109/MSP.2004.111.

[7] Stephen C Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[8] Neil D. Jones. «Flow analysis of lambda expressions». En: *Automata, Languages and Programming*. Ed. por Shimon Even y Oded Kariv. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, págs. 114-128. ISBN: 978-3-540-38745-9.

[9] P. D. Bruza y Th. P. van der Weide. «Assessing the Quality of Hypertext Views». En: *SIGIR Forum* 24.3 (nov. de 1990), págs. 6-25. ISSN: 0163-5840. DOI: 10.1145/101306.101307. URL: https://doi.org/10.1145/101306.101307.

[10] Mary Ann Vandermark. «Defect Escape Analysis: Test Process Improvement». En: (jun. de 2003). URL: https://www.agileconnection.com/sites/default/files/article/file/2012/XDD6919filelistfilename1_0.pdf.

[11] *Pysa, un analizador estático para Python ofrecido por Facebook*. 2023. URL: https://blog.desdelinux.net/pysa-un-analizador-estatico-para-python-ofrecido-por-facebook/.

[12] Mark Grechanik et al. *Bridging gaps between developers and testers in globally-distributed software development*. 2010.

[13] Concord Consortium. *The Concord Consortium*. URL: https://concord.org/.

[14] Blake Loring, Duncan Mitchell y Johannes Kinder. «ExpoSE: practical symbolic execution of standalone JavaScript». En: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 2017, págs. 196-199.

[15] WHK Bester, Cornelia P Inggs y WC Visser. «Test-case generation and bug-finding through symbolic execution». En: *Proceedings of the South*

African Institute for Computer Scientists and Information Technologists Conference. 2012, págs. 1-9.

[16] Dongjun Youn, Sungho Lee y Sukyoung Ryu. «Declarative static analysis for multilingual programs using CodeQL». En: *Software: Practice and Experience* (2023).

[17] Peter Seibel. *Practical Common Lisp*. Apress Berkeley, CA, 2005. DOI: https://doi.org/10.1007/978-1-4302-0017-8.