Análisis de vulnerabilidades en las dependencias de proyectos de desarrollo software

Antonio Germán Márquez Trujillo

Tutores: José Ángel Galindo Duarte
David Benavides Cuevas

Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

Máster Universitario en Ingeniería del Software: Cloud, Datos y Gestión TI

Julio 2022

Dedico este trabajo de fin de máster a mis padres, por ser los que han permitido que esté aquí. Gracias por la educación que me habéis dado, por las ganas de ser siempre un poquito más. A toda mi familia y en especial a mi abuela María por ser un apoyo en malos momentos. También se lo dedico a mis amigos, a los que se cuentan con los dedos de una mano y siempre son sinceros cuando se necesita.

Agradecimientos

Gracias a Ángel, José y David B. por su guía y apoyo constante a lo largo del desarrollo de este trabajo de fin de máster. Gracias a todo el equipo de Diverso por saber escucharme en los momentos en los que necesitaba nuevas ideas o información sobre algo que desconocía. Gracias a todo el hub por darme esta oportunidad y los materiales necesario para desarrollar este trabajo. Y por último, gracias a David Romero por el apoyo constante incluso en momentos personales difíciles.

Abstract

Security has become a crucial factor in the development of software systems. The number of dependencies in software systems is becoming a source of countless bugs and vulnerabilities. Due to the scalability in the number of dependencies and their versions, i.e., the variability within dependencies, the developer of a software system cannot analyze it manually. In the past, the software product line community proposed several techniques and mechanisms to cope with the problems that arise when dealing with variability and dependency management in such systems. In this memory, we present a solution that allows automated dependency analysis for vulnerabilities within software projects based on techniques from the software product line community.

Our solution first inspects the software dependencies and their available versions, then generates a graph of dependencies, each with their available versions and to which security-related information is subsequently attributed. In other words, vulnerabilities or CVEs (Common Vulnerabilities and Exposures) are the formal and centralized way for the cybersecurity community to represent the information on a vulnerability, the systems it affects, whether it can be exploited, and the severity or impact it has. The systems involved are specified in a list of CPEs (Common Platform Enumeration - CPE) within the CVE, which are diagrams with information about the affected systems. And the impact is defined based on the CVSS (Common Vulnerability Scoring System), which is a standard system for scoring the impact or severity of a vulnerability on a range from 0 to 10. This is done by making use of the Vulnerabilities Data Base (VDB), which is a repository of data on vulnerabilities discovered and cataloged to date.

Then, it is translated into a formal model, in this case, based on the Satisfiability Modulo Theories (SMT). This allows us to transform a model, in this case the dependency graph, into a Boolean satisfiability problem using propositional logic that we can reason about and analyse. To this end, we design a set of analysis and reasoning operations on these problems that allow us to extract useful information about the set of vulnerabilities of the project's configuration space, that is, the set

of dependencies with their associated versions. As well as information for advice on the security risk of these projects and their possible configurations, allowing the developer to automatically perceive this information which would be very time-consuming if done manually. We also compared our solution with others on the market, in which we observed that our solution can detect more vulnerabilities in the dependencies in different projects.

Resumen

La seguridad se ha convertido en un factor crucial en el desarrollo de los sistemas software. La cantidad de dependencias existentes en los sistemas software se está convirtiendo en una fuente de innumerables errores y vulnerabilidades, y debido a la escalabilidad en el número de las mismas y sus versiones, es decir, a la variabilidad dentro de las dependencias, al desarrollador de un sistema software no puede analizarlo de forma manual. En el pasado, la comunidad de líneas de productos software propuso varias técnicas y mecanismos para hacer frente a los problemas que surgen al tratar la variabilidad y la gestión de dependencias en dichos sistemas. En esta memoria presentamos una solución, que permite el análisis automatizado de las dependencias en busca de vulnerabilidades dentro de los proyectos de software basándose en técnicas de la comunidad de líneas de productos software.

Nuestra solución inspecciona primero las dependencias del software y sus versiones disponibles, luego genera un grafo de dependencias cada una con sus versiones disponibles y a las que posteriormente se les atribuye información relativa a la seguridad. Es decir, las vulnerabilidades o CVE (Common Vulnerabilities and Exposures), que son la manera formal y centralizada por la comunidad de la ciberseguridad de representar la información de una vulnerabilidad, los sistemas a los que afecta, si se puede explotar y la severidad o impacto que tiene. Los sistemas a los que afecta se especifica en una lista de CPEs (Common Platform Enumeration - CPE) dentro del CVE, estos son esquemas con información sobre los sistemas afectados. Y el impacto se define en base al CVSS (Common Vulnerability Scoring System) el cual es un sistema común para puntuar en un rango del 0 al 10 el impacto o severidad de una vulnerabilidad. Todo esto haciendo uso de las bases de datos de vulnerabilidades (Vulnerabilities Data Base - VDB), que son repositorios de datos sobre las vulnerabilidades descubiertas y catalogadas hasta la actualidad.

Luego, se traduce a un modelo formal, en este caso, basado en las teorías de satisfacibilidad módulo (Satisfiability Modulo Theories - SMT). Que nos permite transformar un modelo, en este caso el grafo de dependencias, a un problema de satisfacibilidad booleana usando lógica proposicional el cual podamos razonar y

analizar. Para ello diseñamos un conjunto de operaciones de análisis y razonamiento sobre estos problemas que permiten extraer información útil sobre el conjunto de vulnerabilidades del espacio de configuraciones del proyecto, es decir, el conjunto de dependencias con sus versiones asociadas. Así como de información para el asesoramiento sobre el riesgo de seguridad de estos proyectos y sus posibles configuraciones, haciendo que el desarrollador de forma automática perciba esta información que de manera manual sería muy costoso en tiempo. Así mismo comparamos nuestra solución con otras existentes en el mercado, en las que observamos que nuestra solución es capaz de detectar más vulnerabilidades en las dependencias en diferentes proyectos.

Índice

Índice de figuras xiii

Índice de tablas xv

1 Introducción 1

1.1 Contexto . . . 1
1.2 Motivación . . . 2
1.3 Descripción del problema . . . 5
1.4 Objetivos . . . 6
1.5 Ámbito de investigación . . . 7
1.6 Contribución . . . 7

2 Estado del arte 9

2.1 Introducción . . . 9
2.2 Minado de datos en repositorios . . . 9
2.3 Variabilidad . . . 11
2.3.1 Sistemas de alta variabilidad . . . 11
2.3.2 Modelos de variabilidad . . . 11
2.3.3 Cardinalidad . . . 13
2.3.4 Análisis automático . . . 17
2.4 Ciberseguridad . . . 19
2.4.1 ¿Qué es? . . . 19
2.4.2 La necesidad de la ciberseguridad . . . 19
2.4.3 Bases de datos de vulnerabilidades . . . 20
2.4.4 Inconsistencias . . . 21
2.5 Soluciones existentes . . . 22
2.6 Sumario . . . 22

3 Propuesta **25**
3.1 Introducción 25
3.2 Solución propuesta 25
3.3 Extracción del grafo de dependencias 26
3.3.1 Minado de la información 27
3.3.2 Filtrado de las versiones 27
3.3.3 Construcción nodo a nodo 27
3.4 Atribución del grafo 28
3.5 Transformación a un modelo SMT 29
3.6 Aplicación de las operaciones 31
3.7 Sumario 32

4 Evaluación de la propuesta **35**
4.1 Introducción 35
4.2 Plataforma de la experimentación 36
4.3 Comparación con otras herramientas 36
4.4 Evaluación de los grafos obtenidos 37
4.5 Resultados obtenidos en las operaciones 38
4.6 Amenazas a la validez 40
4.7 Sumario 41

5 Conclusiones y Trabajo Futuro **43**
5.1 Conclusiones 43
5.2 Trabajo Futuro 44

Bibliografía **47**

Índice de figuras

1.1 Ejemplo motivador 4

2.1 Modelo básico 12
2.2 Modelo con cardinalidad 14
2.3 Modelo extendido 15
2.4 Modelo de variabilidad ortogonal 15

3.1 Proceso soportado por Advisory. 26
3.2 Grafo de dependencias atribuido del ejemplo motivador 28

4.1 Cantidad de vulnerabilidades que detecta cada herramienta. 38
4.2 Dependencias, restricciones y vulnerabilidades detectados por proyecto. 39
4.3 Gráfico usando las operaciones de impacto máximo y mínimo por herramienta. 40

Índice de tablas

4.1 Cantidad de configuraciones por umbrales. 36

Capítulo 1

Introducción

1.1 Contexto

Los proyectos software normalmente delegan gran parte de la funcionalidad en librerías externas, lo que hace que las vulnerabilidades de éstas, puedan afectar al proyecto en desarrollo. En la actualidad, se identifican múltiples vulnerabilidades cada día [1] que deben ser conocidas y gestionadas rápidamente por los desarrolladores. Siendo conscientes de que las cadenas de ciberataques utilizadas por los atacantes para penetrar en los sistemas son cada vez más sofisticadas [2]. Así, los atacantes pueden hacer de versiones de dependencias con vulnerabilidades conocidas, como la reciente vulnerabilidad CVE-2021-44228[1] detectada en Log4j[2], que ha afectado al menos a 186,352 proyectos en el ecosistema Java [3] debido a la complejidad del análisis de sus dependencias. Por tanto, una mala configuración (según OWASP Top-10 vulnerabilidades) en un componente software (dependencia), puede utilizarse como punto de entrada (vector de ataque) para un atacante. Debido a la gran variedad de opciones de configuración de las dependencias es todo un reto analizar las posibles vulnerabilidades de un proyecto software [4][5][6].

Los sistemas de alta variabilidad (Variability-intensive System - VIS) son aquellos sistemas software que, para funcionar de manera correcta, deben gestionar y lidiar con un gran número de dependencias [7][8]. En la literatura encontramos proyectos con centenares de dependencias y opciones de configuración, como por ejemplo el Kernel de Linux con más de $2^{10.000}$ configuraciones distintas. Una configuración se define en este contexto como una combinación válida de versiones de las librerías y artefactos software de los que un proyecto es dependiente. Esta cantidad de

[1]https://nvd.nist.gov/vuln/detail/CVE-2021-44228
[2]https://logging.apache.org/log4j/2.x/

dependencias y librerías dificulta que los desarrolladores sean conscientes de qué vulnerabilidades afectan al software que están desarrollando y cómo tomar medidas para paliar los riesgos de seguridad.

Para ayudar a encontrar vulnerabilidades en las dependencias de un proyecto software encontramos soluciones tecnológicas como Snyk[3], OWASP Dependancy Check[4] o DependaBot de GitHub[5] que ayudan en la identificación de vulnerabilidades debido al uso de dependencias en un proyecto. No obstante, las herramientas y soluciones existentes en el mercado tienen ciertas limitaciones, por ejemplo, no permiten explorar el espacio completo de configuraciones existentes dada la complejidad del mismo [9]. Esta diferencia hace que pueda haber configuraciones que se estén usando y que sean vulnerables, pero no sean detectadas por estos sistemas. Normalmente, y hasta donde sabemos, las herramientas actuales se centran en el análisis de una única configuración dejando al lado, por ejemplo, los posibles entornos de despliegue, que podrían o no estar afectados por vulnerabilidades.

Dada la dificultad del análisis de dependencias de manera manual, la comunidad de líneas de productos software propuso el análisis automático de la variabilidad en los VIS, por ejemplo con técnicas como los AAFM (Automated Analysis of Feature Models) [10]. Los AAFM habilitan el razonamiento sobre los VIS mediante el uso de sistemas de inteligencia artificial o algoritmos ad-hoc, para extraer información relevante del conjunto de dependencias descritas en un VIS. Unido al análisis de variabilidad, han surgido aproximaciones que intentan analizar las vulnerabilidades de una línea de productos software para optimizar el conjunto de pruebas a realizar [11].

1.2 Motivación

Los proyectos de desarrollo software actuales se construyen sobre otras herramientas existentes, conocidas como dependencias. Una dependencia d_x puede tener asociado un conjunto valores de versión V_x válidas para el proyecto. Esto es, $d_x \mapsto V_x$. Para nuestra dependencia d_x, todas las configuraciones estarán definidas por un conjunto de pares de la forma $\{(d_x, v_1), \ldots, (d_x, v_n)\}$ con respecto al conjunto de versiones válidas. La combinatoria entre las diferentes versiones de las diferentes dependencias definen el espacio de configuración del proyecto completo. Si nuestro

[3]https://www.snyk.io
[4]https://owasp.org/www-project-dependency-check/
[5]https://github.com/dependabot

proyecto tuviera tres dependencias d_x, d_y, y d_z el espacio de configuraciones vendría definido de la siguiente manera:

$$\{d_x \mapsto V_x\} \times \{d_y \mapsto V_y\} \times \{d_y \mapsto V_y\} \tag{1.1}$$

Por ejemplo, supongamos que tenemos las dependencias d_x y d_y, tomando cada una un número diferente de versiones en el rango 1.0 a 2.0 incluidas, habiendo hasta n versiones intermedias; el espacio de configuraciones completo del proyecto se puede representar de la siguiente manera:

$$\{(d_x, 1.0) \ \ y \ \ (d_y, 1.0), \ \ \ldots, \ \ (d_x, 1.n) \ \ y \ \ (d_y, 1.n), \ \ \ldots, \ \ (d_x, 1.n) \ \ y \ \ (d_y, 2.0)\} \tag{1.2}$$

Este espacio de configuraciones permite al desarrollador elegir para su proyecto una determinada configuración que se adapte a su entorno. Podemos pensar todo esto para un entorno de despliegue o integración continuo (CI/CD), que dé servicios a clientes. Sin embargo, el espacio de configuraciones crece exponencialmente en función del número de versiones de cada dependencia, quedando para n dependencias:

$$proyecto_{EspacioConfig} = \prod_{i=1}^{n} |V_i| \tag{1.3}$$

Por ejemplo, en el caso de tener 4 dependencias con 10 versiones cada una, tendríamos teóricamente 10.000 configuraciones posibles. Esta cota superior sobre el número de configuraciones se puede reducir mediante distintas restricciones que se suelen especificar en un archivo dentro de los proyectos. Por ejemplo, la dependencia d_x debe ser mayor o igual que la versión 1.5, quedando de la siguiente manera $d_x \geq 1.5$, lo que implica que la elección de dicha versión continúe siendo amplia. Esto requiere un análisis complejo de qué versiones son o no son válidas. El hecho de tener un espacio de configuraciones elevado dificulta al desarrollador la elección de la óptima para sus necesidades.

Como ya hemos comentado, las dependencias pueden verse afectadas por vulnerabilidades conocidas. Actualmente el *Common Vulnerabilities and Exposures (CVE)* de Mitre[6] es el estándar de facto usado para representar la información de las vulnerabilidades, y es usada en casi todas las bases de datos como la de *National*

[6]http://cve.mitre.org/

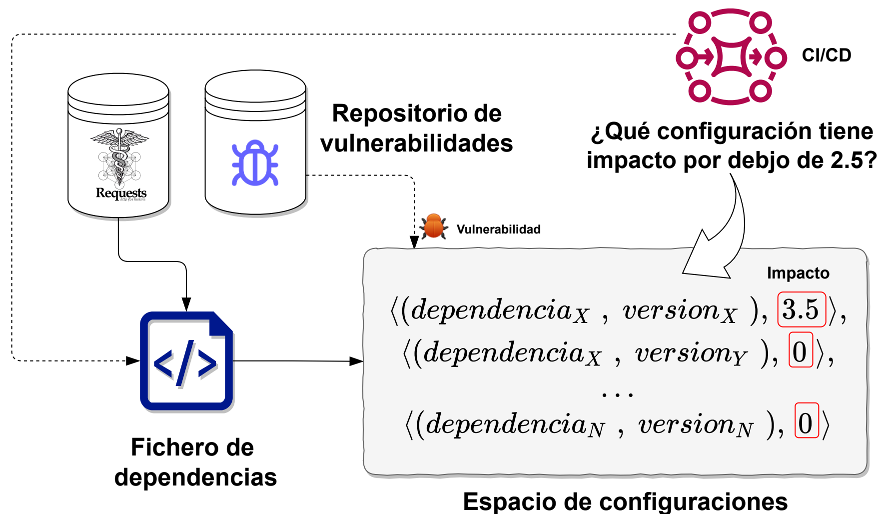

Fig. 1.1 Ejemplo motivador

Vulnerability Database (NVD) de NIST[7]. Generalmente, un CVE viene definido por un identificador de la forma CVE–$\langle year \rangle$–$\langle identificador \rangle$, por ejemplo, CVE-2022-27528. Además, cada CVE tiene asociado una descripción detallada, un impacto o severidad expresado mediante el Common Vulnerability Scoring System (CVSS)[8], y una lista de Common Platform Enumeration (CPE)[9] con características de las aplicaciones software, hardware o sistemas que se ven afectados por esta vulnerabilidad. Por ejemplo, la vulnerabilidad CVE-2021-43546, tiene un impacto CVSS de 4.3 sobre 10 (nivel medio), y uno de sus CPE es para la aplicación Mozilla Firefox, expresado de la siguiente manera: *cpe:2.3:a:mozilla:firefox:*:*:*:*:*:*:*:**.

Como hemos comentado antes, las vulnerabilidades tienen asociadas un nivel de impacto o severidad (CVSS) que hacen que debamos tenerlas en cuenta para evaluar la seguridad de las configuraciones seleccionadas en los proyectos. Con lo que a la complejidad de seleccionar una configuración, hay que añadirle la consideración de evaluar el impacto producido por las vulnerabilidades asociadas.

En la Fig. 1.1 tenemos un ejemplo donde un sistema CI/CD toma un fichero de dependencias que usa la dependencia de Request de Python. El paquete Requests[10] de llamadas HTTP, es usado en aproximadamente 1.200.000 repositorios y 65.000 paquetes[11], lo que da una idea de su repercusión sobre otros proyectos. Podemos

[7] http://nvd.nist.gov
[8] https://www.first.org/cvss/
[9] https://nvd.nist.gov/products/cpe
[10] https://docs.python-requests.org/
[11] https://github.com/psf/requests/network/dependents

comprobar que Requests a su vez depende (indirectamente) de urllib3 y flask. A partir de ahora, usaremos Request, urllib3 y flask, y asumiremos un fichero de dependencias que define las siguientes dependencias: $urllib3 \geq 1.21.1$ y $urllib3 < 1.27$, y $flask > 1.0$ y $flask < 2.0$. Si analizamos el espacio de configuraciones, podemos comprobar que hay un total de 29 versiones posibles para urllib3 y 9 para flask, lo que significa un total de 261 configuraciones de versiones posibles. Además, de entre las 216 configuraciones detectamos que al menos una vulnerabilidad está asociada a una de las dependencias. Es decir, si el desarrollador eligiera aleatoriamente una configuración de entre las 216, aproximadamente un 83% de las veces estaría asumiendo un impacto de seguridad en su configuración. Es decir, la configuración que eligiera podría tener una vulnerabilidad, y por tanto un impacto sobre su seguridad.

Una posible pregunta que se podría plantear cualquier desarrollador o un sistema CI/CD, es si podríamos asegurar minimizar el riesgo o impacto, seleccionando una configuración cuyo impacto de seguridad sea inferior a 2.5. Para esto, tendremos que analizar el espacio de configuraciones y las vulnerabilidades.

1.3 Descripción del problema

Como exponemos en la secciones 1.1 y 1.2 el análisis de las dependencias dentro de los proyectos de desarrollo software es una tarea demandante en tiempo. Debido a que el número de configuraciones puede aumentar enormemente con tan solo una decena de versiones en varias dependencias, una sola persona no puede abarcar el análisis de todo el espacio de configuraciones. A esto le añadimos que el desarrollador tendría no solo que indagar en las dependencias directas de su proyecto si no también en las dependencias indirectas del mismo es decir, dependencias de las dependencias directas.

Por lo cual es imposible para un desarrollador de forma manual configuración por configuración ver si las dependencias y sus versiones tienen vulnerabilidades, calcular un impacto para esas configuraciones y ver cual de ellas es la menos vulnerable; en un tiempo razonable. A esto le añadimos que no existe una única forma de rastrear las vulnerabilidades, ya que existen múltiples bases de datos de vulnerabilidades, haciendo más complicado el proceso manual.

Aunque este problema esta parcialmente resuelto por algunas de las herramientas mencionadas en 1.1, estas herramientas solo analizan la configuración con las versiones mas actuales de entre las permitidas por las restricciones de los ficheros de dependencias. Esto resulta problemático ya que no ofrece un análisis que cubra

el espacio de configuraciones de una manera más amplia resulta en la perdida de información que puede llevar al desarrollador a malas decisiones. Como ampliar el espacio de configuraciones de su herramienta a versiones más actuales sin necesidad, cuando el espacio de configuraciones puede disponer de una configuración que utilice alguna versión algo más antigua que no esté afectada por la vulnerabilidad. No aumentando así el número de versiones a utilizar u obligar al desarrollador a utilizar versiones más actuales que pueden ser mas inestables.

Por último mencionar, que no solo se trata de analizar el espacio de configuraciones, si no que más allá tratamos de razonar sobre el mismo. De forma que podamos ofrecer información diferente como por ejemplo, las 10 configuraciones menos vulnerables del espacio de configuraciones. Operación que de forma manual sería complejo realizar en un tiempo razonable, siendo un problema para el desarrollador.

1.4 Objetivos

El desarrollo de este trabajo tiene por objetivo principal el análisis y el razonamiento sobre el espacio completo de configuraciones de los proyectos de desarrollo software de forma que podamos proporcionar, una solución al problema descrito anteriormente. Para ello, presentamos una serie de objetivos más específicos que desgranan al objetivo principal:

- Modelar las dependencias y sus versiones, es decir, el espacio de configuraciones de un proyecto software, y representarlo mediante un grafo de dependencias. Extrayendo dicha información de los repositorios software de dichos proyectos.
- Atribuir ese espacio de configuración con información de seguridad, es decir, las vulnerabilidades asociadas a las dependencias y las versiones de esas dependencias. Obteniendo un grafo atribuido.
- Transformar dicho grafo atribuido en un modelo de teorías de satisfacibilidad módulo (Satisfiability Modulo Theories - SMT) al que poder razonar y analizar.
- Habilitar técnicas y operaciones que permitan razonar sobre el espacio de dependencias de un proyecto software teniendo en cuenta la información de seguridad relacionada con las vulnerabilidades (p.ej., no usar Log4j).

- Evaluar y verificar como se comporta nuestra solución con respecto a las soluciones existentes en el mercado.

1.5 Ámbito de investigación

Los dos campos principales a los que aporta nuestra investigación son la variabilidad y la ciberseguridad, relacionados a proyectos software y las dependencias que este va acumulando a lo largo de su construcción. Mejorando una serie de aspectos concretos como:

1. **Extracción de dependencias.** La extracción de información sobre las dependencias desde un repositorio software y la representación de esta misma en un grafo de dependencias.
2. **Modelización de la variabilidad de dependencias.** Debido a que, este grafo es una representación de todas las dependencias más sus posibles versiones, modelizamos un espacio de configuración sobre la variabilidad de estas dependencias.
3. **Detección de vulnerabilidades.** Al tomar en cuenta todo el espacio de configuración y no solo una parte del mismo, podemos detectar más vulnerabilidades en los proyectos analizados. No solo para una versión concreta de la dependencia, si no para todas las versiones posibles de la misma.
4. **Razonamiento sobre las dependencias.** Al utilizar algoritmos de razonamiento automático podemos razonar sobre este espacio de configuraciones una vez transformado en un modelo SMT. Aplicamos operaciones que nos aportan información sobre las posibles configuraciones de versiones como el impacto de seguridad de las mismas, las que tienen mas o menos impacto o si existe alguna con impacto cero.

1.6 Contribución

Como resultados de este trabajo de investigación y desarrollo se han realizado las siguientes publicaciones:

- **Antonio Germán Márquez Trujillo**, Ángel Jesús Varela-Vaca, José A. Galindo, María Teresa Gómez-López, and David Benavides. Advisory: Análisis de

vulnerabilidades en proyectos de desarrollo software. En las actas de las séptimas Jornadas Nacionales de Investigación en Ciberseguridad, JNIC 2022, Bilbao (País Vasco), Spain, Junio.

- **Antonio Germán Márquez Trujillo**, Ángel Jesús Varela-Vaca, and José A. Galindo. Advisory. Una herramienta para identificar los riesgos de seguridad. En las actas de la vigésimo sextas Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2022, Santiago de Compostela (Galicia), España, Septiembre.
- **Antonio Germán Márquez Trujillo**, Ángel Jesús Varela-Vaca, and José A. Galindo. Advisory. Advisory. Una propuesta de tesis para identificar los riesgos de seguridad. En las actas de la segundas Jornadas de Investigación Predoctoral en Ingeniería Informática, JIPII 2022, Puerto Real (Cádiz), España, Junio.
- **Antonio Germán Márquez Trujillo**, Ángel Jesús Varela-Vaca, José A. Galindo, María Teresa Gómez-López, and David Benavides. Advisory: Vulnerability analysis in software development projects. In Proceeedings of the 26nd International Systems and Software Product Lineconference, SPLC 2022, Graz, Austria, September.

Además de las anteriores contribuciones académicas, se ha construido un framework para el análisis de la seguridad en las dependencias de proyectos software. El siguiente artefacto como resultado de la implementación de Advisory:

- **Advisory**: Análisis de vulnerabilidades en proyectos de desarrollo software (Artefacto): https://zenodo.org/record/6483497

Capítulo 2

Estado del arte

2.1 Introducción

Cada vez con mayor frecuencia los proyectos de desarrollo software utilizan una mayor cantidad de dependencias, cada una con un rango de versiones en aumento constante. Convirtiéndose en un entorno de alta variabilidad donde es imposible calcular de forma manual la seguridad de las dependencias que usamos. Para esto varias solucionas actuales nos permiten analizar las versiones mas actuales de cada dependencia que se usa eliminando así la variabilidad, con la perdida de información que esto conlleva. El AAFM puede ayudar a solucionar este problema con un perdida menor de información. Haciendo uso de un grafo de dependencias como modelo de variabilidad y bases de datos de vulnerabilidades para atribuirlo con información relativa a la seguridad.

En este capitulo haremos un descripción del proceso de minado de datos en repositorios en la sección 2.2 y su aplicación para la extracción de grafos de dependencias 2.3.3. Luego describiremos todos los aspectos relacionados a la seguridad con los que atribuiremos el grafo y sus posibles problemas en la sección 2.4, de la misma forma describiremos los aspectos de la variabilidad y el AAFM tanto como sus problemas en la sección 2.3. Terminando con una breve recopilación de las soluciones existentes en la sección 2.5

2.2 Minado de datos en repositorios

El minado de datos de repositorios de software (Mining Software Repositories - MSR) es una tarea que cuenta cada vez con una demanda mayor, con una gran

variedad de técnicas adscritas a diferentes metodologías [12]. Siendo las siguientes cuatro de las más importantes:

- **Análisis de metadatos**: esta metodología usa los metadatos guardados en un repositorio software (e.g., logs de comandos, o ficheros de errores). Para estudiar o analizar el propio repositorio y aportar información a sus propietarios.
- **Análisis estático del código fuente**: este enfoque utiliza el análisis estático del código fuente para extraer hechos y otra información de las versiones de un sistema. Estos enfoques abarcan una amplia gama de técnicas disponibles para analizar, procesar y extraer datos del código fuente.
- **Diferenciación y análisis del código fuente**: se ha desarrollado métodos para expresar y derivar los cambios en el código fuente de una forma más "consciente", es decir, sintaxis y semántica. Para esto se utiliza la información del código fuente o los modelos del código fuente.
- **Métricas de software**: las diferentes métricas del software como el tamaño, el esfuerzo, el coste, la funcionalidad, la calidad, la complejidad, la eficiencia, la fiabilidad o la mantenibilidad. Pueden ser utilizadas en el contexto del MSR para poder calcularlas a partir del propio repositorio.
- **Visualización**: se basa en el uso de la visualización para facilitar la comprensión de la información extraída de un repositorio software. Este enfoque se basa en la agrupación de los datos extraídos y la presentación de los mismos.
- **Métodos de detección de clones**: la búsqueda de composiciones textuales, estructurales o semánticamente similares en el código fuente. En este contexto el MSR puede ser usado para la detección de estos clones para su posterior mitigación.
- **Minería de patrones frecuentes**: se utiliza el MSR para detectar cambios frecuentes en el código fuente. Estos cambios pueden ser secuenciales y en esos cambios se pueden detectar lo que se denomina como *patrones frecuentes*, es decir, entidades que cambian con frecuencia o siempre de una determinada forma.

De entre todas las enumeradas en la que nos centramos para la extracción de las dependencias de un proyecto software es en el **Análisis de metadatos** para analizar los ficheros de dependencias contenidos en el repositorio del proyecto.

Estos datos se pueden representar como sistemas de alta variabilidad, debido a la cantidad de dependencias que contiene un proyecto software y las versiones de cada dependencia. Estos sistemas de alta variabilidad los describimos en la sección siguiente dando una visión general sobre los mismos, su tipos de modelos y la capacidad del análisis automático sobre los mismos.

2.3 Variabilidad

2.3.1 Sistemas de alta variabilidad

Un VIS es aquel sistema software que necesita gestionar y lidiar con un gran número de características, para funcionar correctamente [7] [8]. Un ejemplo de estos tipos de sistemas que podemos encontrar en la literatura es el kernel de Linux, que cuenta con más de $2^{10.000}$ configuraciones diferentes. Estas configuraciones son combinaciones válidas de las características que componen el VIS.

En este caso concreto el proyecto software sería el sistema de alta variabilidad, y las dependencias con sus respectivas versiones serían las características del mismo. Entendemos entonces una configuración como un conjunto de dependencias cada una con una versión asociada. Estos sistemas de alta variabilidad se pueden modelar con distintas aproximaciones que explicaremos en la sección siguiente. Para modelar este tipo de sistemas se propone el uso de grafos dirigidos cíclicos, también llamados grafos de dependencias en este caso concreto

2.3.2 Modelos de variabilidad

Los sistemas de alta variabilidad se pueden representar en diferentes aproximaciones de modelado para un mejor análisis del mismo. No solo visual, también automático ya que existen herramientas que pueden analizar y razonar sobre los mismos. En esta sección daremos un vistazo a las principales aproximaciones para el modelado de estos sistemas y cual escogemos para el desarrollo de este trabajo.

Modelos de características

La representación mas básica de una linea de productos software es un modelo de características básico [13], está compuesto por características, que son componentes del sistema de alta variabilidad, y relaciones entre estos componentes que pueden ser de tres tipos:

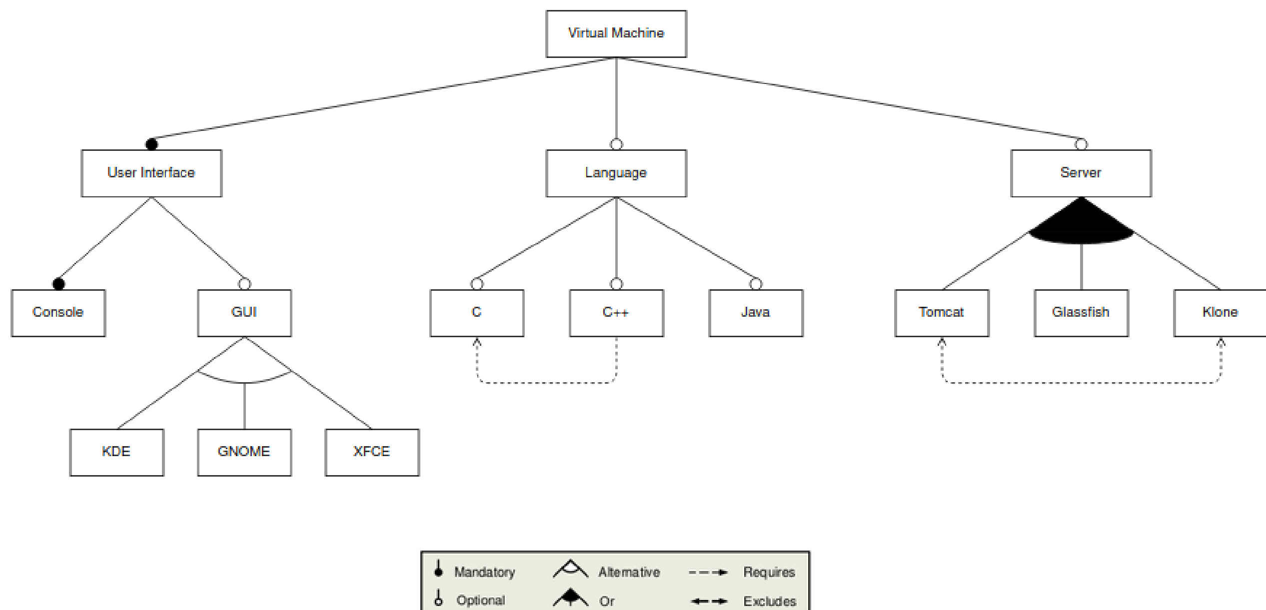

Fig. 2.1 Modelo básico

- Relaciones entre una característica padre y sus características hijas.
- Relaciones cruzadas en el árbol, de dos tipos, requerimiento o de exclusión, y que sirven como restricciones a la hora de representar la linea de productos software. Estas son de la forma: Si la característica A es incluida en el modelo la característica B debe ser también incluida (o excluida).
- También existen restricciones complejas de tipo implicación, que sirven para denotar operaciones lógicas en ellas de la forma: Si la característica A es incluida en el modelo implica que B y C deben estar incluidas también (o excluidas, o también B o C deben estar incluidas). O también operaciones lógicas simples pueden ser añadidas a las relaciones del modelo como: A y no B.

En la Figura 2.1 se pueden ver además las relaciones parentales que pueden darse entre características:

- **Obligatoria**: Una característica hija tiene una relación obligatoria con su característica padre sí, siempre que aparece la característica padre en un producto la característica hija tiene que ser seleccionada también. Por ejemplo, toda interfaz de usuario debe tener una consola.

- **Opcional**: Una característica hija tiene una relación opcional con su característica padre sí, siempre que aparece la característica padre en un producto la característica hija puede o no ser seleccionada. Por ejemplo, toda interfaz de usuario puede o no tener una interfaz gráfica de usuario (GUI).
- **Alternativa**: Un conjunto de características hijas tienen una relación alternativa con su característica padre sí, siempre que aparece la característica padre en un producto, solo una de esas características hijas podrá ser seleccionada. Por ejemplo, si se escoge que la maquina virtual tendrá GUI esta debe ser KDE, GNOME o XFCE, pero solo una de ellas.
- **Or**: Un conjunto de características hijas tienen una relación or con su característica padre sí, siempre que aparece la característica padre en un producto, una o mas de esas características puede ser seleccionada. Por ejemplo, si se escoge que la maquina virtual tenga servidor podrá tener una combinación entre Tomcat, Glassfish y Klone, seleccionando al menos siempre una.

También se muestran en la figura las relaciones cruzadas en el árbol, que sirven como restricciones:

- **Requerimiento**: Si la característica A requiere una característica B, B aparecerá en todos los productos en los que aparezca A. Por ejemplo, toda maquina con el lenguaje C++ requerirá que contenga también el lenguaje C.
- **Exclusión**: Si la característica A excluye a la característica B, entonces ningún producto contendrá a las dos características al mismo tiempo. Por ejemplo, ninguna maquina virtual puede tener un servidor Tomcat y Klone a la vez.

2.3.3 Cardinalidad

Algunos autores propusieron extender los modelos básicos a modelos con cardinalidad, como si fueran modelos tipo UML con multiplicidad [14] [15] (lo que vamos a llamar cardinalidades). Esto se hizo para generalizar las relaciones dentro del modelo independientemente del tipo que sean de una forma sencilla y práctica, es decir, gracias a esto tenemos una regla general para formalizar las relaciones en un modelo de características, como se puede ver en la Figura 2.2. Las cardinalidades pueden ser de dos tipos:

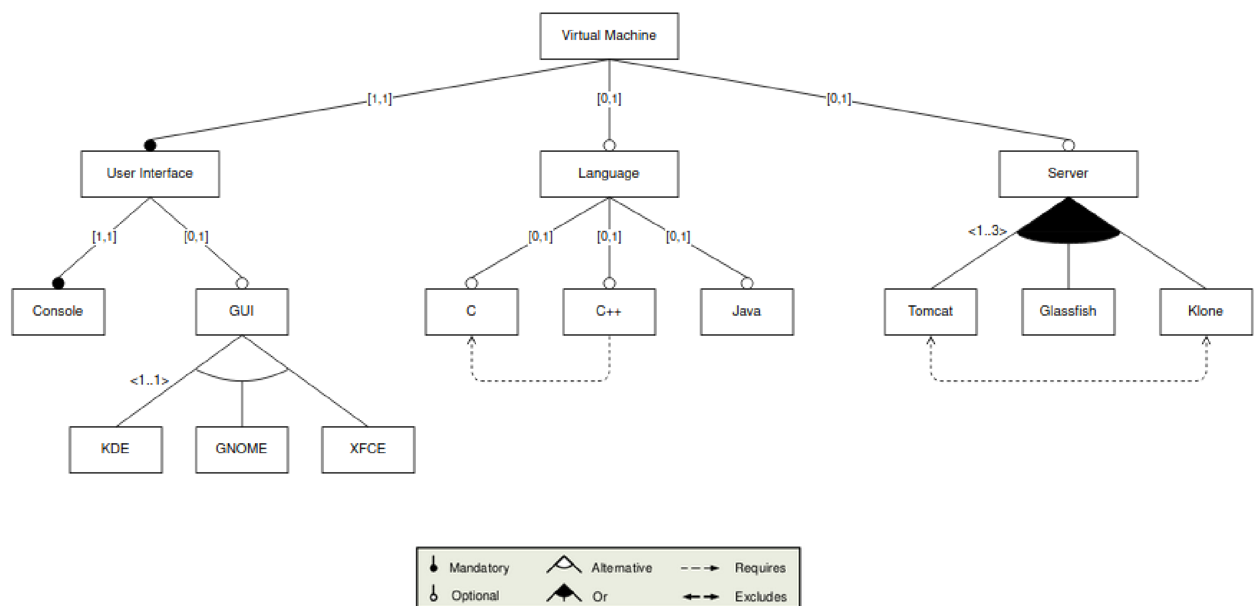

Fig. 2.2 Modelo con cardinalidad

- Cardinalidad característica: Esta cardinalidad viene definida por un intervalo del tipo [n..m] donde n es el limite inferior y m el limite superior, determinando el numero de instancias de la característica que puede ser parte del producto. Quedando las relaciones obligatorias definidas como [1,1] y las opcionales [0,1].

- Cardinalidad grupal: Esta cardinalidad viene definida por un intervalo del tipo $< n...m >$ donde n es el limite inferior y m el limite superior que determina el numero de características hijas que pueden ser parte del producto cuando la característica padre es seleccionada. Quedando las relaciones alternativas definidas como $< 1,1 >$ y las or como $< 1, m >$, siendo M el número de características hijas en la relación.

Modelos de características atribuidos

Un modelo de características básico o cardinal puede tener necesidades a la hora de almacenar información de las características, de esta forma nacieron los modelos atribuidos, los cuales pueden almacenar información de las características en base a atributos de las mismas como se puede ver en la Figura 2.3. Estos atributos serían considerados como características no funcionales [16], pero no hay una notación actual que describa como definirlos. Aún así la mayoría de las proposiciones están

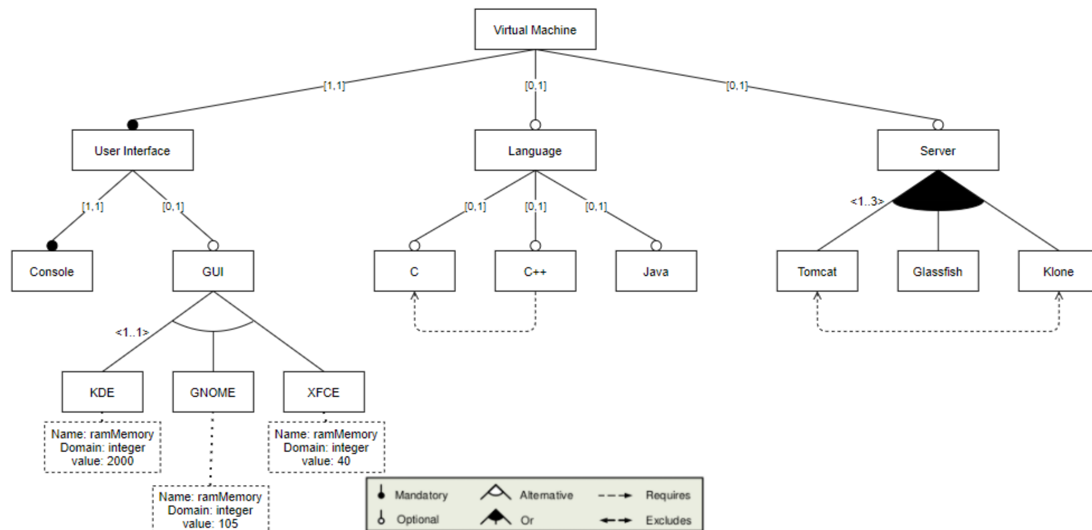

Fig. 2.3 Modelo extendido

de acuerdo en que deben contener al menos un nombre, un dominio y un valor. Por ejemplo, cada GUI de la máquina virtual puede ser corrida con un limite de memoria RAM, pues este límite podría ser un atributo de esas características. Esto además, puede dar la posibilidad a introducir restricciones mas complejas entre los atributos como, "Si el atributo memoryRam de la característica GNOME es menor que el valor de la RAM de la maquina virtual la característica GNOME no podrá ser parte del producto".

Modelos de variabilidad ortogonal

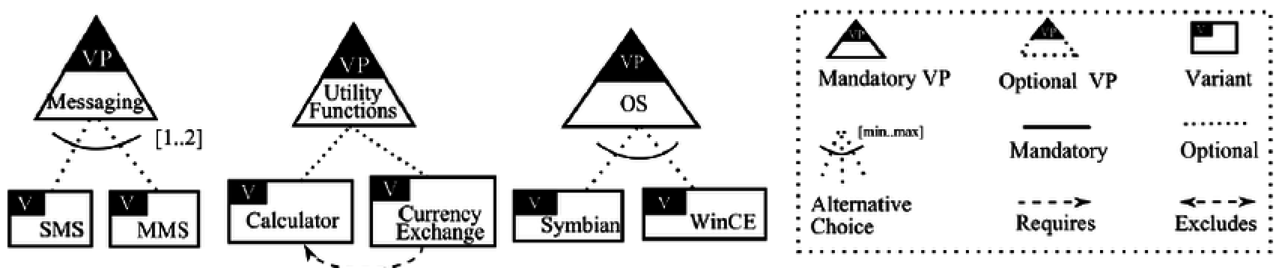

Fig. 2.4 Modelo de variabilidad ortogonal

Un modelo de variabilidad ortogonal es otra de las propuestas para representar una linea de productos software [17], estos también están compuestos por características, relaciones parentales y relacione cruzadas en el modelo. La diferencia está

en el uso de cardinalidad en las relaciones y las propiedades de las características. Tomando como ejemplo el modelo de la figura 2.4 podemos ver que las características pueden ser:

- **Variant Point (VP)**: Representa una variable dentro del modelo en el caso de la figura de ejemplo *Messaging* sería una variable obligatoria, es decir que siempre debe estar presente en las configuraciones que se deriven del modelo. Si es opcional, esta variable puede o no pertenecer a las configuraciones.
- **Variant (V)**: Representa las posibles instancias que puede tener una variable, en el caso de *Messaging* se puede instanciar como *SMS* o *MMS*.

Las relacione de estos modelos de variabilidad entre una variable y una instancia son totalmente iguales al de un modelo de características con la diferencia de que la relación alternativa se define con cardinalidades. Por lo que fuera como la relación alternativa de un modelo de características se define como **[1, 1]** y una or como **[1, m]**. Siendo m el número de características hijas de la relación o algún número intermedio.

Grafos de dependencias

Un grafo dirigido es aquel en el cual las aristas entre un vértice y otro tienen un sentido único, a diferencia de un grafo no dirigido donde las aristas no tienen un sentido definido siendo bidireccionales. Estos grafos además pueden ser cíclicos o acíclicos, es decir si permiten o no la formación de ciclos entres dos o mas vértices.

Estas propiedades son usadas para definir grafos de dependencias, donde el sentido de las aristas corre siempre desde los vértices dependientes a sus vértices dependencias, pudiendo esas dependencias tener otras dependencias, es decir, subdependencias. Si el grafo permite ciclos o no dependen del contexto en el que se trate, en el nuestro hemos decidido que así sea ya que es interesante poder detectar dependencias cíclicas.

Estos grafos de dependencias anteriormente han sido usados para modelar y representar gestores de paquetes completos, en literatura podemos encontrar ejemplos como el repositorio central de maven [18]. Pero nunca han sido tratados también como un sistema de alta variabilidad, es decir, como un sistema que tenga que lidiar con la variabilidad implícita en todas las características que lo componen. Estos grafos junto con los modelos de características atribuidos nos han servido como base para desarrollar un grafo de dependencias atribuido, el cual usamos

para este trabajo, con información relativa a la ciberseguridad que definimos en la sección 2.4, dotando así a las dependencias y sus versiones con vulnerabilidades asociadas. Para mas tarde poder analizarlo con las técnicas que se describen en la sección siguiente.

2.3.4 Análisis automático

Los modelos de variabilidad derivados de un VIS son difícilmente analizables manualmente debido a la alta variabilidad de los sistemas que representan, debido a esto la comunidad de líneas de productos software propuso el análisis automático de los modelos de variabilidad, como por ejemplo el AAFM [10] [19]. Este razonamiento permite el análisis sobre los modelos de variabilidad mediante el uso de sistemas de inteligencia artificial o algoritmos ad-hoc, para extraer información relevante del conjunto de características de un VIS. Incluso existiendo propuestas para el análisis de las vulnerabilidades de una linea de productos software para la optimización del conjunto de pruebas a realizar [11]. A continuación definimos las principales técnicas de análisis automático de modelos de variabilidad.

Constraint Satisfaction Problem

Esta técnica se basa en los problemas de satisfacción de restricciones (Constraint Satisfaction Problem - CSP), que consisten en un conjunto de variables, un conjunto finito de dominios y un conjunto de restricciones constriñendo el valor de las variables. Esto es a lo que se denomina la programación de restricciones, la cual es un conjunto de técnicas tales como algoritmos o heurísticas que tratan con CSPs. Estos son resueltos buscando valores para las variables en los cuales todas las restricciones son satisfechas. Como es deducible los resolutores CSP solo tratan con valores binarios 1 y 0 (true/false), pero también trabajan con valores numéricos como enteros o intervalos. Estos CSP aportan un conjunto mas rico de elementos de modelado y restricciones que los resolutores de lógica proposicional, pero son menos eficientes.

Boolean satisfiability problem

La lógica proposicional se define como un conjunto de variables cuyo valor es restringidos por un grupo de conectores lógicos como not, and, or, condicional y bicondicional.

Esta lógica puede ser analizada por resolutores basados en problemas de satisfabilidad booleana (también llamados SAT), que son paquetes que toman una formula proposicional y determinan si esta es satisfascible comprobando si hay una asignación de esa formula que la haga verdadera. Para ello estos resolutores necesitan que la formula de entra sea descrita en forma normal conjuntiva (Conjuntive Normal Form - CNF). Esto es, una representación de la formula proposicional donde solo puedan ser usados los conectores lógicos and, or y not. Esta técnica puede lidiar con grandes problemas en la mayoría de los casos para dar una solución a las operaciones propuestas anteriormente. Con la problemática de que se pierden elementos de modela y restricciones.

Binary Decision Diagrams

De manera parecida a la técnica anterior los diagramas de decisión binario (Binary Decision Diagrams - BDD) son resolutores que definidos en un paquete toman una formula proposicional, no necesariamente a diferencia de la técnica anterior, y la transforma en un representación en grafo. Permitiendo saber si la formula es satisfacible a la vez que provee una gran eficiencia a la hora de contar soluciones (configuraciones). El problema de está técnica es que paga su eficiencia con un desempeño en memoria exponencial que imposibilita el análisis de modelos que escalan a un gran tamaño en número de componentes.

Satisfiability Modulo Theories

Por último, existen los resolutores basados en las teorías de satisfacibilidad módulo (Satisfiability Modulo Theories - SMT) [20] [21]. Son resolutores extendidos a partir de los resolutores SAT pero usando las teorías módulo, que son una serie de reglas para definir con lógica proposicional un problema de satisfacibilidad, es decir, con estructuras de primer orden.

La combinación de ambas (SAT más teorías módulo) da lugar a los SMT, modelos de satisfacibilidad booleana basados en teorías módulo que permiten modelar problemas en lógica proposicional de primer orden, proporcionando una flexibilidad léxica característica de un CSP. Aprovechando por otro lado la eficiencia y rapidez de los resolutores SAT. Siendo este tipo de resolutores el que usamos a lo largo del trabajo por ofrecer un eficiencia, flexibilidad y ahorro de memoria.

2.4 Ciberseguridad

2.4.1 ¿Qué es?

La seguridad en la informática y la computación, también conocida como la ciberseguridad, es la protección de la infraestructura del hardware y del software, así como de la información contenida en la misma. Ya sea esta información estática o circulante por la misma. Su objetivo principal es minimizar los riesgos de la información y la infraestructura mediante la aplicación de normas como restricciones de acceso, autorizaciones, protocolos, perfiles de usuario, planes de emergencia y/o denegaciones. Podemos definir los objetos que tiene que defender como:

- **La infraestructura**: es una parte principal del sistema informático, ya que dota de la capacidad computacional y funcional para el funcionamiento correcto del mismo. No solo se trata de defenderlo de personas que les den un mal uso, también de robos, incendios o desastres naturales por ejemplo.

- **Los usuarios**: son las personas que hacen uso de los sistemas informáticos. Por lo cual se deben proteger estos sistemas para proteger a estos usuarios de otros malintencionados.

- **La información**: es el principal recurso de los sistemas informáticos, y como tal puede ser utilizado de forma maliciosa.

En el caso concreto de este trabajo nos centramos en la seguridad relativa a las dependencias de nuestro software, es decir, en proteger la información y los usuarios de nuestro sistema de software de terceros maliciosos, o que sean vulnerables y puedan aprovecharse maliciosamente.

2.4.2 La necesidad de la ciberseguridad

Tras la ciberseguridad se esconde la necesidad de protegernos de aquellas amenazas que suponen un riesgo para la integridad de nuestros datos o para los usuarios a los que ofrecemos nuestro servicio. Por esto los desarrolladores de software quieren mantener alejadas las vulnerabilidades de su software, aunque esto es una tarea difícil porque no puedes asegurar un código completamente. Por eso existe el análisis de riesgos que pretende identificar las zonas vulnerables de nuestro sistema para mitigar mientras más de ellas mejor. Y por otro lado, tener un plan que

mida el impacto de una vulnerabilidad en caso de que sea descubierta y explotada maliciosamente por un tercero, para determinar perdidas y un plan de actuación.

Muchos de estos planes para mitigar los riesgos de seguridad tienen en cuenta las bases de datos de vulnerabilidades, un arma para dar a conocer y así solucionar las vulnerabilidades que se descubren en sistemas software de forma que se minimice la posibilidad de que algún ente malicioso pueda encontrarla antes y explotarla a su favor. En la siguiente sección haremos un descripción de las bases de vulnerabilidades existentes.

2.4.3 Bases de datos de vulnerabilidades

Las bases de datos de vulnerabilidades son extensos repositorios de datos los cuales sirven para almacenar las vulnerabilidades conocidas, preservando su persistencia y el conocimiento de las mismas para poder evitarlas [22]. Actualmente encontramos tres tipos de estas bases de datos:

- **Bases de datos comerciales**: bases de datos pertenecientes a grandes empresas o que se dedican a encontrar vulnerabilidades para almacenarlas en sus bases de datos, obteniendo remuneración económica del que quiera saber su existencia.
- **Bases de datos open-source**: bases de datos open-source, estas suelen ser mantenidas y alimentadas por la comunidad. Un ejemplo de estas, por su gran relevancia pasada, era la Open Source Vulnerability Data Base (OSVDB) que actualmente sigue descontinuada.
- **Bases de datos gubernamentales**: bases de datos pertenecientes a un estado o gobierno concreto del mundo, tales como la NVD [23].

La mayoría de estas bases de datos tienen algo en común, lo que se conoce como el Common Vulnerabilities and Exposures (CVE) que explicaremos a continuación.

CVE, CPE y CVSS

La MITRE CVE es un esfuerzo comunitario por conformar una lista única de todas las vulnerabilidades conocidas, manteniendo así un registro persistente donde estas no puedan duplicarse. Cada vulnerabilidad en la lista debe estar acompañada por un identificador único con el siguiente formato **CVE-YYYY-NNNN**, donde *YYYY* indica el año y *NNNN* el número de vulnerabilidad. Estas vulnerabilidades son

descubiertas y listadas por todas las organizaciones que colaboran y se coordinan con el MITRE, informando así de cada vulnerabilidad hallada.

Cada CVE cuenta con una serie de información relativa a la seguridad como el CVE-ID, una descripción, posibles formas de mitigarla o explotarla, una lista de Common Platform Enumerations (CPEs) y un Common Vulnerability Scoring System (CVSS), que explicamos como:

- **CPE**: es un esquema de nomenclatura estructurado que permite comprobar una serie de información[1] vinculada a nombres de un sistema, software o paquete.
- **CVSS**: este el es sistema común de puntuación de vulnerabilidades, un marco abierto en el que se puntúa en un rango de 0 a 10 la severidad de la vulnerabilidad en base una serie de métricas temporales y ambientales, como la antigüedad del software o la cantidad de plataformas dependientes del mismo.

2.4.4 Inconsistencias

Aunque exista la lista del MITRE como esfuerzo común para centralizar las vulnerabilidades esto no elimina el problema de que actualmente haya inconsistencias entre las diferentes bases de datos. Y que por supuesto será algo a lidiar a la hora de intentar atribuir un grafo de dependencias con información relativa a la seguridad. Esto se da por una serie de motivos:

- Organizaciones que no colaboran con el MITRE en todos o en ningún caso, reservando vulnerabilidades que no están centralizadas.
- Vulnerabilidades que aún estando en el MITRE no han sido añadidas a todas las bases de datos.
- Siempre teniendo en cuenta que trabajamos con vulnerabilidades existentes, aquellas no descubiertas no pueden ser tomadas en cuenta. Y las recientemente descubiertas sufren el riesgo de no estar estandarizadas en todas las bases de datos.

Actualmente se están desarrollando propuestas para intentar mitigar estas inconsistencias [24].

[1] <cpe_version>:<part>:<vendor>:<product>:<version>:<update>:<edition>:<language>:<sw_edition>:<target_sw
:<target_hw>:<other>

2.5 Soluciones existentes

Tras lo anteriormente mencionado buscamos propuestas que trabajen en la misma línea que nosotros, y encontramos entre todas ellas dos en estado avanzado de desarrollo que ofrecen un servicio para la detección de vulnerabilidades en dependencias de proyectos de desarrollo software. Y que actualmente trabajan comercialmente, estas son:

- **Snyk**: es una herramienta de detección de vulnerabilidades en código fuente, dependencias, contenedores e infraestructuras como código. Permitiendo la corrección automática mediante pull requests a GitHub en caso de vulnerabilidades en código fuente o dependencias, modificando el fichero de dependencias correspondiente. Actualmente trabajan para cualquier tipo de proyecto independientemente del lenguaje, permitiendo también el uso de pipelines CI/CD en tu repositorio GitHub.

- **DependaBot**: es una herramienta integrada en GitHub que permite actualizar aquellas dependencias que sean vulnerables a versiones más recientes. Sus funciones son más limitadas que las de Snyk ya que solo trabaja con dependencias. Aún así se puede configurar un pipeline CI/CD con el mismo y trabaja con todo tipo de proyectos software alojados en GitHub independientemente del lenguaje.

Estas herramientas cumplen con parte de la motivación expuesta en 1 el problema de ambas es que solo analizan la versión latest de las dependencias, de las versiones que son especificadas en el fichero de dependencias correspondiente. El no contemplar la variabilidad en las versiones conlleva a una perdida de información que por ejemplo, induce a correcciones erróneas. Como recomendar versiones todavía más recientes que rompan la compatibilidad del sistema, en vez de recomendar una que esté dentro del rango y que no rompe la compatibilidad del sistema.

2.6 Sumario

El estado del arte que describimos en este capítulo dan una visión general del contexto y estado actual de los objetivos que pretendemos alcanzar, dando un vistazo por todas las tecnologías que utilizamos como el MSR, los grafos de dependencias, los VIS y los términos relacionados con la ciberseguridad. Además de aquellos

aspectos que no han sido definidos aún en la literatura y que desarrollaremos en este trabajo.

Por último, repasamos herramientas actuales que podemos definir como competidoras. Que como vemos son mejorables y creemos que nuestro esfuerzo dará un paso mas allá en el estado del arte, disminuyendo la perdida de información.

Capítulo 3

Propuesta

3.1 Introducción

Tras la revisión del estado del arte y las soluciones existentes en el mercado actualmente. Nuestra propuesta se basa en primer lugar, en el desarrollo de un framework que permita el extracción de grafos de dependencias extraídos de un repositorio software mediante el minado de repositorios software. Que incluya las dependencias y además las versiones de estas, modelando así todo el sistema de variabilidad especificados en los ficheros de dependencias. Para poder razonar sobre el espacio de configuraciones que define el modelo.

Para eso desarrollamos Advisory, como framework que primero extrae el grafo de dependencias (modelo de variabilidad). Luego lo atribuye, desde bases de datos de vulnerabilidades, con información relativa a la seguridad de las dependencias y las versiones extraídas (modelo de variabilidad atribuido). Y por último, tiene la capacidad de transformarlo a un modelo de satisfacibilidad booleana para poder aplicar operaciones de análisis y razonamiento sobre el mismo, que aporten información relevante sobre la seguridad de las dependencias del proyecto software.

3.2 Solución propuesta

La Fig. 3.1 muestra una visión general del proceso soportado por Advisory. El proceso se divide en cuatro componentes principales: a) Extracción del grafo de dependencias, por ejemplo, de un fichero requirements.txt de un proyecto Python, extraemos su grafo de dependencias usando la información de GitHub[1]; b) Atribuir

[1]https://docs.github.com/es/graphql

el grafo con información relativa a las vulnerabilidades desde la base de datos de vulnerabilidades de NVD del NIST; c) Codificar la información en un modelo formal basado en SMT solver [25], que nos permite razonar sobre las dependencias y sus vulnerabilidades, y finalmente; d) Aplicar un conjunto de operaciones que nos facilite el análisis de la información de dependencias y sus vulnerabilidades.

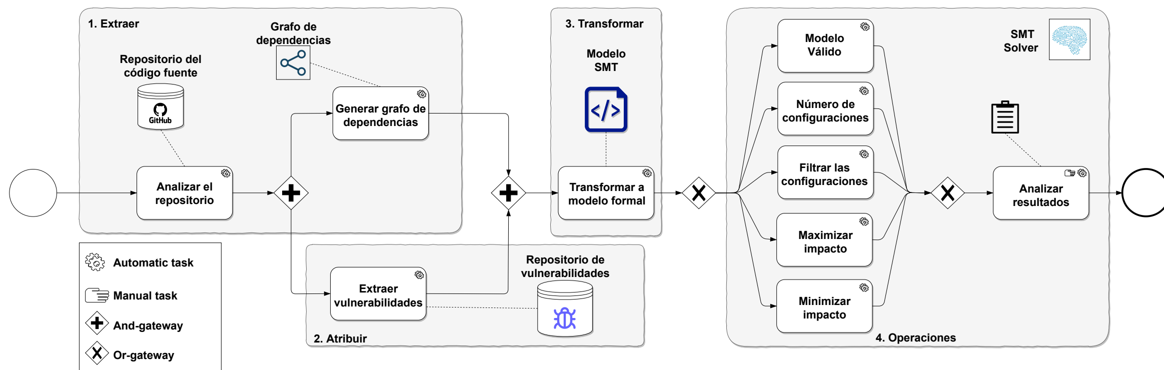

Fig. 3.1 Proceso soportado por Advisory.

De manera resumida y basándonos en el ejemplo motivador de la Sección 1.2, partimos desde los ficheros de dependencias del repositorio (p.ej. requirements.txt de un proyecto Python), extraemos el grafo de dependencias usando la información de GitHub[2]. Posteriormente atribuimos el grafo con información relativa a la seguridad desde la base de datos de vulnerabilidades de NVD del NIST, transformamos dicha información a un modelo formal, y analizamos dicho modelo mediante un SMT solver [25]. Finalmente, sobre este modelo podemos aplicar diferente operaciones. Estos cuatro componentes combinados proporcionan la solución al análisis de seguridad de un espacio de configuración sobre un proyecto software.

A continuación, describiremos los componentes de manera pormenorizada siempre basándonos o usando como soporte el ejemplo motivador de la Sección 1.2.

3.3 Extracción del grafo de dependencias

En el primer componente (Extraer) construimos el grafo de dependencias de un proyecto software alojado en un repositorio. Este proceso consiste en los siguientes pasos: 1) obtenemos la información desde el repositorio; 2) filtramos las versiones validas para esas dependencias; y 3) construimos nodo a nodo las dependencias del grafo. Como se describe con más detalle a continuación:

[2] https://docs.github.com/es/graphql

3.3.1 Minado de la información

En primer lugar, obtenemos la información sobre las dependencias desde el repositorio de código del proyecto software, extrayéndola de manera automática, por ejemplo, requirements.txt o septup.py. De estos archivos extraemos el nombre de la dependencia y la URL para acceder a ella. En caso de que las dependencias tengan otras sub-dependencias (indirectas), se realizará un proceso recursivo, extrayendo las restricciones sobre los valores de versiones que puede tomar éstas. Actualmente Advisory es capaz de obtener esta información de repositorios alojados en GitHub y de naturaleza Python. Para ello, se apoya en la API GraphQL de GitHub que nos permite hacer llamadas usando el lenguaje GraphQL[3]. Para el caso del ejemplo motivador de la sección anterior, se ha realizado una única llamada al repositorio que contiene los ficheros con las dependencias, en este caso de urllib3 y flask.

3.3.2 Filtrado de las versiones

Después utilizaremos las restricciones para filtrar las versiones válidas. Primero extraemos la información de todas versiones disponibles de cada dependencia. Luego las filtramos una a una y escogemos las que satisfagan las restricciones del fichero de dependencias. Para el ejemplo, estamos trabajando con proyectos de naturaleza Python, por lo que utilizamos el Python Package Index (PyPI)[4] para extraer todas versiones dichas dependencias, para luego filtrarlas según las restricciones del fichero de dependencias. Para las del ejemplo motivador, obtenemos las siguientes versiones: flask={1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4} y urllib3={1.21.1, 1.22, 1.23, 1.24, 1.24.1, 1.24.2, 1.24.3, 1.25, 1.25.1, 1.25.10, 1.25.11, 1.25.2, 1.25.3, 1.25.4, 1.25.5, 1.25.6, 1.25.7, 1.25.8, 1.25.9, 1.26.0, 1.26.1, 1.26.2, 1.26.3, 1.26.4, 1.26.5, 1.26.6, 1.26.7, 1.26.8, 1.26.9}.

3.3.3 Construcción nodo a nodo

Por último, con esta información y las versiones filtradas construimos un nuevo nodo del grafo por cada dependencia y para cada versión, y agregamos los arcos dirigidos para relacionar cada nodo de dependencia con los de sus versión, además de incluir otras relaciones con nodos padres, e hijos (si fuera necesario). Así lo haremos con todas las dependencias extraídas. Nótese que dada la más que probable explosión

[3]https://graphql.org/
[4]https://pypi.org/

combinatoria de dependencias, será necesario especificar un nivel de profundidad del grafo. Empezando por la raíz que será nuestro proyecto (Requests para ejemplo) que tendrá profundidad 0, realizamos una llamada que construirá tanto el nodo raíz como todos los nodos de profundidad 1, que serán las dependencias de nuestro proyecto (o dependencias directas). Sí definimos que nuestro grafo debe tener profundidad 2, posteriormente haremos una llamada por cada nodo de profundidad 1 para construir sus dependencias, que serán subdependencias o dependencias indirectas de nuestro proyecto. El grafo a nivel 1 del ejemplo motivador quedaría como en la Fig. 3.2. Notar que en la figura se han añadido información de las vulnerabilidades que se detallará a continuación.

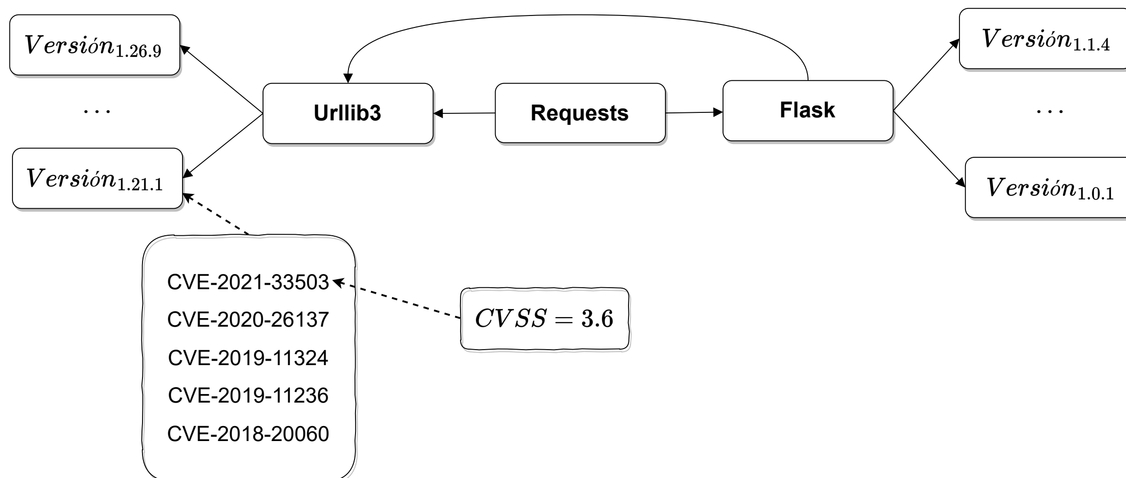

Fig. 3.2 Grafo de dependencias atribuido del ejemplo motivador

3.4 Atribución del grafo

Ahora describiremos como se atribuyen los nodos del grafo, es decir las dependencias, con información relativa a las vulnerabilidades (CVE). Para este proceso actualmente utilizamos la API de la base de datos de NVD del NIST. En primer lugar, utilizaremos el nombre de la dependencia, como clave para encontrar esas vulnerabilidades a la hora de realizar las búsquedas. De estas búsquedas extraemos todos los CVE que incluyen en alguno de sus CPE el nombre dicha dependencia. En el caso de flask y urllib3, NVD nos devuelve 2 y 8 CVE respectivamente asociados a esas dependencias[5].

[5]Estos datos son los obtenidos en el momento de redactar este artículo.

Estos CVEs podrán asociarse a una versión o no incluidos de nuestro grafo, por lo que tendremos que analizar, si alguna de las versiones de las dependencias está incluida en algún CPE de estos CVE, ya que de lo contrario ese CVE no debería ser asignado a ninguna versión de nuestro grafo. Por ejemplo, para flask ninguno de los 2 CVE detectados será asociado porque no coincide con alguna versión contemplada en sus CPEs. Además, para urllib3, 7 de 8 CVEs si son asociados a las versiones válidas del ejemplo motivador. Esto lo hacemos así porque hacer peticiones individuales por cada versión ralentizaría el proceso de atribución debido a que cada consulta es una petición a la API de NVD y consumiría demasiado tiempo.

De esta forma relacionamos las versiones de las dependencias con los CVE que hemos extraído y que les afectan de forma directa. Además, vamos a almacenar para cada CVE el valor de CVSS (impacto), así como el conjunto de métricas de su vector de ataque. Por ejemplo, podemos ver los CVE asociados para la dependencia 1.21.1 de urllib3 y el impacto del CVSS de uno de ellos en la Fig. 3.2. Este impacto también viene con el CVE, y será fundamental para determinar el impacto del espacio de configuraciones.

3.5 Transformación a un modelo SMT

Por último, el grafo atribuido lo transformaremos a un modelo Satisfiability Modulo Theories (SMT) [25]. Como hemos mencionado en la sección 2.3.4 escogemos este tipo de resolutores por su expresividad léxica que nos ayuda con los enteros para las versiones y el impacto, con una eficiencia de un SAT y sin la alta demanda de memoria de un BDD. Un modelo SMT es un modelo formal de satisfacción de restricciones generalizado (SAT) que permite usar fórmulas más complejas que implican números reales, enteros y/o diversas estructuras de datos como listas, matrices, vectores de bits y cadenas. Para nuestro aproximación, un SMT vendrá definido mediante la tupla: $\langle D, V, DC, Vul, f_d(D, V), f_t(I) \rangle$. Donde:

- D es el conjunto de nodos del grafo, y representará el conjunto de variables del modelo. Para una dependencia $d_i \in D$ si $d_i = 0$ significa que no es seleccionado para el análisis, y si $d_i = 1$, sí es seleccionados. Si escogemos flask y urllib3, ambas dependencias deben aparecer en el modelo SMT como variables.

- V es el conjunto de valores de versiones de cada dependencia escogida del grafo. La dependencia sólo podrá tomar los valores de las versiones que tiene

disponible en el grafo. Como mencionamos, si se escogen flask y urllib3, estas deben tomar como valor alguna de las dependencias válidas que ya vimos en la sección anterior.

- *DC* representa las restricciones que aplicamos a las versiones de cada dependencia del grafo. En este caso, al tratarse de variables de tipo numérico podemos hacer uso de operaciones lógicas de rango. Por ejemplo, para el caso de urllib3 del ejemplo motivador, $urllib3 \geq 1.21.1$ y $urllib3 < 1.27$.

- *Vul* es el conjunto de valores de impacto, es decir, el impacto definido en cada CVSS asociado a un CVE.

- $f_d(D, V)$ es una función que permite calcular el impacto para cada dependencia escogida del grafo en función de la configuración de versiones que tome el resolutor. En nuestra caso, vamos a considerar todos los CVE con sus impactos (CVSS), para lo que debemos agregar esta información por cada nodo. Para ello, podemos utilizar diferentes funciones que calculen el impacto de la dependencia agregando los impactos, por ejemplo, podemos usar la media, la mediana o la moda. Si usáramos la media para determinar el impacto de la dependencia d_x en la versión v_s que tiene n vulnerabilidades (CVE), implica que si la dependencia toma esa versión, quedaría tal que para cada dependencia el calculo del impacto sería:

$$Impacto_{(d_x,v_s)} = \frac{\sum_{i=1}^{n} CVSS_{CVE_i}}{n} \tag{3.1}$$

  Por ejemplo, si la dependencia urllib3 tomase la versión 1.21.1, el impacto se calcularía de la siguiente manera:

$$\frac{CVSS_{CVE-2021-33503} + \ldots + CVSS_{CVE-2018-20060}}{5} \tag{3.2}$$

- $f_t(I)$ es una función que calcula el impacto total de un proyecto agregando el impacto de todas las dependencias, es decir, si nuestro proyecto tuviera para m dependencias cada una con sus versiones quedaría como sigue:

$$Impacto_{total} = \frac{\sum_{i=1,j=1}^{m,s} Impacto_{(d_i,v_j)}}{m} \tag{3.3}$$

  Al igual que la anterior función, podemos utilizar diferentes funciones que calculen el impacto de la dependencia como la media, la mediana o la moda.

Para el ejemplo motivador, usando una media, el impacto total se quedaría como sigue:

$$\frac{Impacto_{urllib3} + Impacto_{flask}}{2} \tag{3.4}$$

Para aplicar esta transformación en la versión actual de Advisory hemos usado modelos para el resolutor Z3[6].

3.6 Aplicación de las operaciones

Una vez construimos el modelo SMT a partir del grafo, podemos aplicar operaciones de razonamiento. De entre ellas, las actualmente implementadas por Advisory son:

- **Modelo válido:** Conocer si el modelo puede encontrar alguna solución que satisfaga todas las restricciones y operaciones creadas. Esta operación devolvería un booleano *True* o *False* indicando que el proyecto es válido o no. El ejemplo motivador es válido dado que existe al menos una configuración, por ejemplo, la compuesta por {urllib3 = 1.21.1, flask = 1.0.1}.

- **Número de configuraciones:** si el modelo es válido, extraer el número de configuraciones posibles satisfactorias. Nótese que esta operación no es viable ejecutarla en proyectos con muchas dependencias siempre que usemos SMT solvers. En el ejemplo de la Sección 1.2 existen 261 configuraciones. El número máximo de configuraciones que Advisory puede devolver es el representable en un entero de Z3 en Python definido por la función *sys.maxsize()*, un total de 9.223372036854775807 configuraciones.

- **Análisis de las configuraciones:** si las anteriores operaciones no nos aportan la suficiente información sobre el impacto de la seguridad de nuestras dependencias. Hemos desarrollado las siguientes operaciones que permiten refinar el análisis de una configuración. Estas operaciones son los siguientes: a) **Filtrar configuraciones por un umbral mínimo y un umbral máximo**, que nos permite crear un rango de impacto total sobre las configuraciones. Por ejemplo, obtener todas las configuraciones cuyo impacto esté entre 0 y 1.5. Si no se especifican, de forma predeterminada el mínimo se asigna a 0 y el máximo se asigna a 10, que son los valores máximos y mínimo de impacto para una

[6] https://github.com/Z3Prover/z3

vulnerabilidad según CVSS. Por ejemplo, en el caso del ejemplo motivador, si fijamos el umbral máximo a 0 obtendríamos un total de 45 configuraciones. Esta operación devuelve un conjunto de configuraciones; b) **Minimizar o Maximizar** si queremos sacar las configuraciones con menor impacto o las configuraciones con mayor impacto, podemos aplicar una de las dos operaciones de optimización para el impacto. Por ejemplo, en el caso del ejemplo motivador, podríamos maximizar el impacto para obtener la configuración $\{urllib3 = 1.22, flask = 1.1.3\}$ con impacto máximo de 1.83. Estas operaciones devuelven un conjunto de configuraciones; y c) **Limitar exploración.** Estas tres operaciones cuentan con un parámetro para limitar el número de configuraciones que pueden devolver Advisory. Actualmente, analizar el número de configuraciones válidas para problemas de satisfacibilidad, se convierte en casos en los que haya mucha combinatoria, en un problema no determinista en tiempo (NP-completo). Por lo que necesitamos limitar el número de configuraciones que queremos recibir. En otras palabras, dada a imposibilidad de poder enumerar todas las configuraciones existentes que cumplan un criterio de seguridad, permitimos limitar el número de soluciones a un valor especificado por el usuario. Por ejemplo, podríamos obtener 3 configuraciones que cumplan con el umbral máximo de 2.5, o minimizar el impacto y devolver las 3 configuraciones con menor impacto.

Una vez que el resolutor resuelve el modelo junto con la operación requerida, éste arrojará los resultados en términos de lógica proposicional, es decir, a nivel de asignación de valores a cada una de las variables y Advisory se encarga entonces de interpretar los resultados y presentarlos como dependencias y versiones a usar para cumplir los objetivos marcados por el usuario. Nótese que las distintas funciones de optimización están implementadas en lógica proposicional dentro del propio solver (Z3), lo que permite reducir el espacio de búsqueda a la vez que definimos las condiciones del umbral, las funciones de maximización y minimización, y finalmente el número de configuraciones a devolver.

3.7 Sumario

En este capitulo hemos descrito los componentes que forman nuestra solución para dar una solución al problema expuesto en 1.3, el análisis de vulnerabilidades en las dependencias de proyectos de desarrollo software. Estos componentes son la extracción del grafo de dependencias, la atribución del mismo con información

relativa a la seguridad, su transformación a un modelo SMT y el razonamiento que realizamos sobre el mismo.

La extracción del grafo que pasa por el minado de la información del repositorio de código, el filtrado de las versiones que pasan por las restricciones del fichero de dependencias analizado y la construcción nodo a nodo del grafo desde la raíz que es el proyecto que analizamos hasta el nivel de profundidad de dependencias deseado. Luego atribuimos este grafo con información relativa a la seguridad a las dependencias y sus versiones.

Por último transformamos el grafo en un modelo SMT y así poder razonar sobre el mismo. El razonamiento que podemos realizar sobre el grafo de dependencias son las operaciones de modelo valido y número de productos que son las más básicas. Y luego, las operaciones de filtro, maximización y minimización que siendo más complejas pueden aportar una información de mayor utilidad al desarrollador.

Capítulo 4

Evaluación de la propuesta

4.1 Introducción

Para la solución propuesta en el capítulo 3 presentaremos en este capitulo una serie de experimentos con la finalidad de probar el funcionamiento de Advisory. Donde trataremos de probar los datos que extrae nuestra herramienta, las operaciones que desarrollamos para razonar sobre el espacio de configuraciones y una comparativa con las herramientas actuales que hacen algo parecido a nosotros.

La realización de la experimentación necesita que seleccionemos un conjunto representativo de pruebas con la limitación de que deben ser proyectos Python. Para la elección de los proyectos, se ha usado la lista de los 10 proyectos más usados según la web oficial de proyectos más descargados de PyPI[1]. Las características de vulnerabilidades identificadas y del número de configuraciones de los mismos se presenta en la Tabla 4.1. En este caso mostramos los resultados del número teórico de configuraciones totales, y el número de configuraciones alcanzables para umbrales de impacto 0, (0-2.5], (2.5,10]. Estos datos serán analizados más adelante en los experimentos.

Advisory y los experimentos descritos en este trabajo están disponibles como código abierto para la comunidad en Zenodo[2]. El cual es un repositorio abierto, donde los investigadores pueden subir sus investigaciones, así como conjuntos de datos, artefactos software o cualquier archivo digital relacionado con algún campo de la investigación. Acuñándose un objeto digital persistente (DOI) que lo identifique.

[1]https://pypistats.org/top
[2]https://doi.org/10.5281/zenodo.6479353

Table 4.1 Cantidad de configuraciones por umbrales.

Proyecto	Total Configs.	Impacto		
		$= 0$	> 0 y ≤ 2.5	≥ 2.5
boto3	$1.88 \cdot 10^6$	+1000	+1000	0
urllib3	$1.35 \cdot 10^{13}$	0	+1000	0
botocore	$2.37 \cdot 10^7$	+1000	+1000	0
s3transfer	$5.46 \cdot 10^5$	0	+1000	0
requests	$2.02 \cdot 10^{19}$	+1000	+1000	0
six	151	151	0	0
typing	$7.04 \cdot 10^7$	+1000	0	0
dateutil	$1.05 \cdot 10^{22}$	0	+1000	0
aws-cli	$5.21 \cdot 10^8$	+1000	+1000	0
charset-normalizer	$2.79 \cdot 10^{12}$	+1000	+1000	0
click	$1.4 \cdot 10^7$	+1000	+1000	0
numpy	$5.56 \cdot 10^{23}$	+1000	+1000	0
cryptography	86,580	+1000	+1000	0
packaging	$2.7 \cdot 10^5$	+1000	0	0

4.2 Plataforma de la experimentación

El entorno de experimentación en los que realizamos los experimentos, respecto al apartado hardware son una máquina con CPU Intel(R) Core(TM) i7-10510U (1.80GHz hasta 4,90GHz, 8MB caché, 4 núcleos), una tarjeta gráfica integrada Intel UHD Graphics, una memoria RAM de 16GB 2666MHz DDR4-SDRAM y un almacenamiento de 750GB NVMe PCIe SSD. Y respecto al apartado software son un entorno Ubuntu 20.04.4 LTS como sistema operativo, y Python versión 3.9.12 como lenguaje de programación.

4.3 Comparación con otras herramientas

En el mercado ya existe una serie de herramientas que detectan vulnerabilidades en proyectos software y sus dependencias, por eso es interesante posicionar el rendimiento de Advisory con respecto a ellas. En concreto analizaremos Snyk y DependanBot de GitHub dada su popularidad, uso, y soporte de análisis de

proyectos tipo Python. Se ha descartado OWAS Dependacy Check porque ésta se basa en proyectos de naturaleza Java y Advisory sólo soporta actualmente proyectos tipo Python.

En este experimento intentaremos verificar si Advisory puede detectar más vulnerabilidades (cuantificadas como número de CVE detectados) en el análisis de las vulnerabilidades de las dependencias de un proyecto que estas. Es decir, si somos capaces de producir más información sobre las vulnerabilidades de un proyecto que otras opciones. Esto ayudaría a los desarrolladores a tener una información más ampliada de las posibles vulnerabilidades que pueden afectar a sus proyectos. Para poder compararnos minimizando los sesgos, configuraremos nuestra herramienta para que sólo busque en el primer nivel de profundidad (dependencias directas) del grafo.

Hipótesis: Advisory detecta más vulnerabilidades que las soluciones alternativas para los proyectos analizados. En nuestra hipótesis nula es que nuestra herramienta puede detectar más vulnerabilidades en las dependencias de los proyectos software más usados.

Resultados del experimento 1. Los resultados que obtenidos para cada proyecto analizado se puede ver en la Fig. 4.1. Vemos que Snyk detecta más vulnerabilidades que DependaBot para todos los proyectos, pero además Advisory consigue detectar más vulnerabilidades que Snyk para todos los proyectos, excepto para el proyecto *cryptography*, sólo 1 más. Este último caso, parece ocurrir debido a las diferencias entres las bases de datos de vulnerabilidades usadas por Snyk y Advisory o posibles inconsistencias [26] entre las base de datos de Snyk y NVD. Otro posible motivo es que Synk, por defecto, busca en más de un nivel de dependencias (dependencias indirectas) las vulnerabilidades, mientras que Advisory sólo se ha ejecutado en este experimento configurado para explorar el primer nivel de dependencias.

4.4 Evaluación de los grafos obtenidos

En el segundo experimento mostramos los datos procedentes de la extracción de los grafos y la atribución de los mismos. Probamos así el funcionamiento de Advisory, en las etapas de extracción de la información descritas en la Sección 3.3.

Hipótesis: Advisory es capaz de atribuir con información de seguridad las dependencias que extrae de los proyectos. Nuestra hipótesis es que con Advisory extraemos y atribuimos una cantidad de dependencias, restricciones y vulnerabilidades

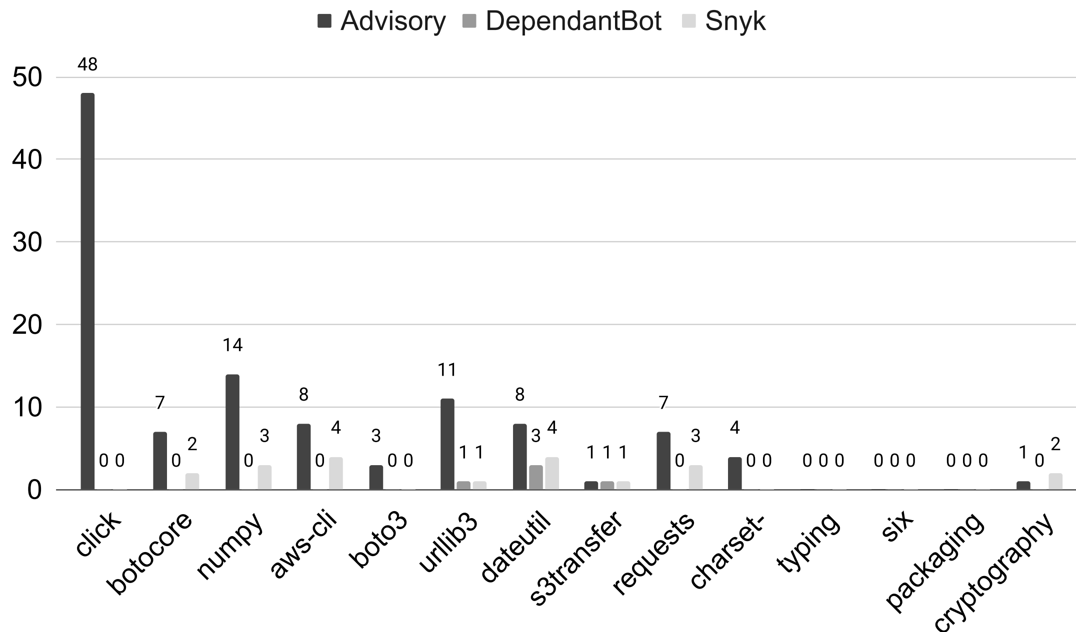

Fig. 4.1 Cantidad de vulnerabilidades que detecta cada herramienta.

coherente entre ellos, es decir, que mientras más dependencias encontramos más restricciones somos capaces de extraer y más vulnerabilidades atribuimos.

Resultados del experimento 2. Realizamos la extracción del grafo y la atribución sobre los proyectos, con los resultados de la Fig. 4.2 y la Tabla 4.1. Como vemos, en todos los casos, se aprecia una relación entre el número de dependencias detectadas y el número de restricciones y CVEs. Mientras más dependencias extraemos, más restricciones y mientras más dependencias más posibilidades de tener vulnerabilidades. Pero a la vez a más restricciones, menor número de versiones, lo que reduce el número de vulnerabilidades asociadas. En proyectos como *botocore* o *aws-cli* donde el número de restricciones crece más que las dependencias, el espacio de configuraciones es más pequeño y por lo tanto es atribuido con menos vulnerabilidades. Esto no pasa en los proyectos como *click* o *numpy* donde las dependencias y el número de restricciones se mantiene equivalente.

4.5 Resultados obtenidos en las operaciones

En el tercer experimento analizamos las operaciones mencionadas en la Sección 3. Para ello, probamos el funcionamiento de la solución tanto en el conteo de

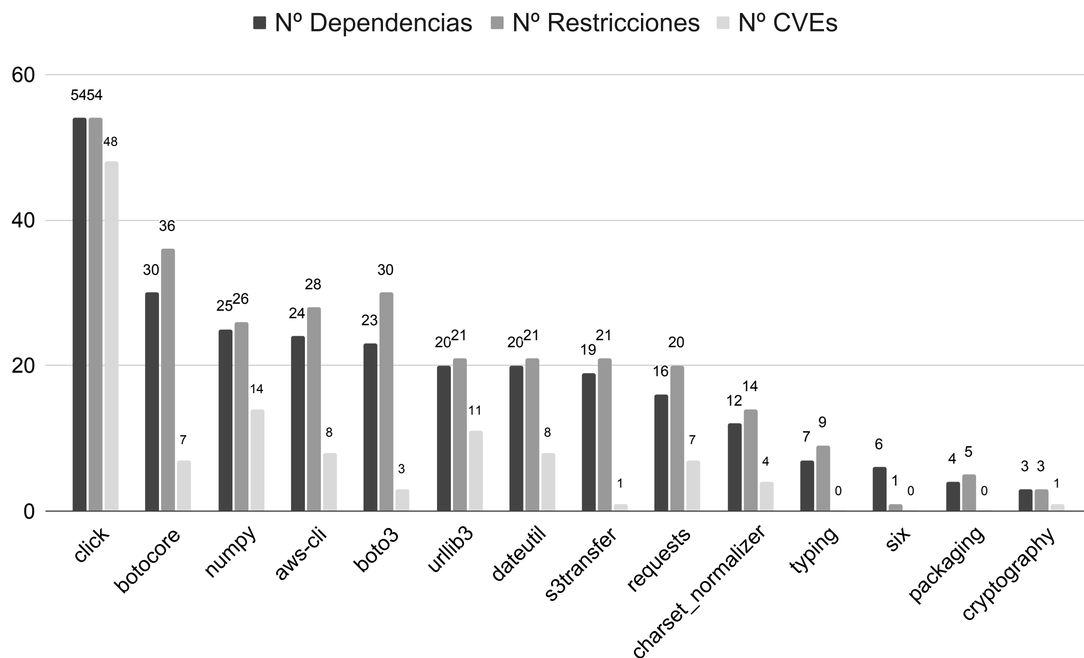

Fig. 4.2 Dependencias, restricciones y vulnerabilidades detectados por proyecto.

configuraciones que cumplan ciertas condiciones como para la identificación del impacto máximo y mínimo. Para ello, ejecutamos las operaciones de optimización sobre los proyectos y reportamos el valor medio del impacto detectado.

Hipótesis: Nuestras operaciones procesan la suficiente información relativa a la seguridad y aportan información al desarrollador. Nuestra hipótesis pretende dar a entender que las operaciones desarrolladas son herramientas útiles para desarrollar proyectos software seguros.

Resultados del experimento 3. El resultado esperado si podemos detectar si nuestro proyecto tiene configuraciones vulnerables o no. Y además, en caso de tener configuraciones vulnerables cuáles son las que tiene el impacto máximo y mínimo de entre todas. Teniendo en cuenta la problemática del número de configuraciones que tiene cada herramienta de la Tabla 4.1. Comenzamos aplicando las dos operaciones maximización y minimización del impacto, para extraer la configuración con mayor impacto y la configuración con menor impacto. Como vemos en la Fig. 4.3, sólo 3 de las 14 herramientas analizadas no cuentan con configuraciones vulnerables, 8 de ellas cuentan con configuraciones vulnerables pero también con configuraciones sin vulnerabilidades, y otras 3 de ellas no cuentan con configuraciones sin vulnerabilidades.

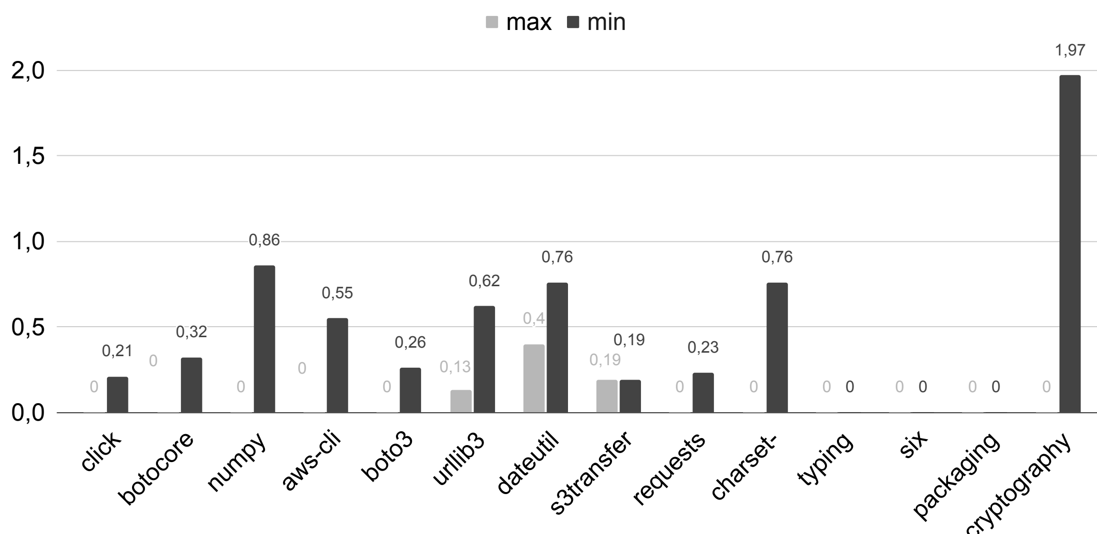

Fig. 4.3 Gráfico usando las operaciones de impacto máximo y mínimo por herramienta.

Realizando la operación de filtro con umbrales sobre estas herramientas, extraemos los datos que vemos en la Tabla 4.1. Podemos observar que existen herramientas, como *urllib3* de la que no se puede obtener una configuración con impacto 0. También podemos ver casos donde hemos acotado el total de configuraciones entorno a un rango, como es el caso de *boto3* o *botocore*, donde todas sus configuraciones están entre 0 y 2.5 de impacto, o el de *packaging* y *six* donde todas sus configuraciones están en impacto 0. Estos datos nos permiten ver y analizar de un simple vistazo a modo de cuadro de mando el conjunto de configuraciones, comprobando qué conjunto de configuraciones están afectados por vulnerabilidades o no.

4.6 Amenazas a la validez

Validez externa: Los datos de los experimentos son realistas, ya que son copias exactas de los repositorios de las herramientas analizadas. Sin embargo, para la extracción de los datos del grafo y la atribución de vulnerabilidades se han utilizado APIs que pueden contener errores externos a nosotros, o que pueden dejar de funcionar, así como inconsistencias entre los datos que obtenemos. La amenazas que nos afectan a la que nos es enfrentamos es sobre ***validez de la populación***, ya que la cantidad de vulnerabilidades atribuidas a un grafo pueden no ser todas las existentes en el momento del análisis. Además, entre diferentes bases de datos

pueden existir inconsistencias. Para reducir esta amenaza en el pensamos nutrirnos de más de una bases de datos, de forma que, si una sufre una inconsistencia pueda ser corregida por el resto.

Validez interna: Los recursos de CPU requeridos para analizar el espacio de configuraciones de un grafo depende del número de versiones y restricciones que se aplican a cada dependencia. Sin embargo, para minimizar estos efectos, hemos introducido parámetros para elegir la profundidad deseada o el número límite de configuraciones a devolvemos. Así controlamos aquellos proyectos con un coste computacional muy elevado. Por otro lado, no estamos exentos de cometer errores en nuestra herramienta que invaliden los resultados de la misma. Por tanto, las amenazas que nos afectan son: 1) ***Regresión estadística***: En el caso del uso de la media como factor de agregación de los diferentes impactos de las vulnerabilidades, hace que los casos extremos tengan demasiada influencia en el cálculo del impacto agregado; y 2) ***Instrumentación***: Las herramientas escogidas para el análisis han sido elegidas siendo las más descargadas a lo largo del presente mes, y podría originar que con el paso del tiempo hayan cambiado..

4.7 Sumario

Como mencionamos en 1.2 actualmente hay una preocupación incremental por las vulnerabilidades que afectan a las dependencias de nuestro software, eso unido a la variabilidad que existen en estas dependencias han motivado nuestra solución. El objetivo de Advisory es que el trabajo de los desarrolladores a la hora de escoger versiones de dependencias que están libres de vulnerabilidades. Esta evaluación tiene el objetivo de validar los objetivos de nuestra solución de cara a suplir el problema expuesto.

El análisis de un espacio de configuraciones con una variabilidad enorme no tiene una solución viable conocida en la actualidad. Pero nuestra aproximación permite un análisis más extenso del que se podría hacer a mano y del que actualmente hacen las herramientas competidoras comparadas con Advisory. Esto es debido a que estas no hacen un análisis del espacio completo de configuraciones, perdiendo información sobre las dependencias de un proyecto software.

En lo respectivo a las vulnerabilidades que conseguimos detectar con nuestra solución, describimos como esta es coherente al numero de dependencias y restricciones de versiones que minamos desde los repositorios. Por otro lado, demostramos que las operaciones implementadas cumplen con su objetivo y aporta información

relevante sobre el estado de la seguridad de las dependencias de nuestros proyectos de desarrollo software. Probando el funcionamiento de las operación de filtrado de configuraciones según umbrales, además de las de maximización y minimización del impacto de seguridad de las configuraciones, observando como se comporta Advisory en herramientas reales que se usan en una gran cantidad de proyectos de desarrollo software actuales.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1 Conclusiones

El análisis de vulnerabilidades en proyectos software resulta en una tarea compleja y crucial debido al elevado número de posibilidades del espacio de configuraciones definido por sus ficheros de dependencias. Además, es un problema real para los desarrolladores a priori pueden escoger una configuración de dependencias cuyo impacto se desconoce o que esté libre de vulnerabilidades. Ya que de forma manual es difícil analizar este espacio de configuraciones.

Advisory resuelve este problema mediante el análisis de este espacio de configuraciones. Primero construyendo un grafo de dependencias atribuido con información de las vulnerabilidades y luego analizando las configuraciones mediante un modelo formal basado en SMT al cual transformamos desde el grafo. Esto permite dar al desarrollador obtener información relevante sobre las vulnerabilidades de las configuraciones analizadas. Hemos comprobado en entornos reales, que somos una solución competitiva y que puede aportar a la comunidad información relevante sobre la seguridad de sus proyectos de desarrollo software.

En resumen, hemos aprendido las siguientes lecciones importantes:

1. **Acercamiento entre variabilidad y seguridad.** Podemos analizar la variabilidad que se encuentra en las dependencias de los proyectos software y extraer información relativa a su seguridad.

2. **La necesidad de unas dependencias seguras.** Tal como se comenta en el ejemplo motivante 1.2 y con casos recientes como Log4J actualmente se necesita saber cuales de nuestras dependencias son seguras para mantener un correcto

funcionamiento de nuestro sistema software. Y que este no se pueda ver comprometido por software de terceros.

3. **Inconsistencias en las bases de datos de vulnerabilidades.** La necesidad de tener que incorporar múltiples bases de datos para evitar inconsistencias que pueden originarse al indexar una vulnerabilidad en una base de datos y en otras no.

4. **Imposibilidad de analizar el espacio completo de configuraciones.** El análisis de espacio de configuraciones muy elevados hace imposible para un resolutor analizar todas las posibles configuraciones. Lo que resulta un problema abierto para el análisis completo de la seguridad de una herramienta con una gran cantidad de dependencias y versiones. Una forma de mitigar esto seria aplicar técnicas de reducción de la variabilidad.

5. **Cada vez se usan más dependencias.** En sintonía con lo anterior, cada vez los proyecto son más dependientes y en número de dependencias que se usan es mayor, aumentando así la variabilidad de dependencias y versiones de los gestores de paquetes. Esto dificultará el análisis de la seguridad de dependencias, por lo que se necesitarán soluciones mejores al problema.

6. **Extensión de nuestra herramienta.** Nuestra herramienta solo trabaja para un gestor de paquetes concreto y unas operaciones aún escasas. Sería interesante expandir los gestores de paquetes, las operaciones que realizamos, la usabilidad de la herramienta mediante una interfaz de usuario y otras funciones.

Además, tenemos ciertas limitaciones y aspectos mejorables que por la falta de tiempo no hemos podido cubrir que se mencionan en la sección siguiente de trabajos futuros.

5.2 Trabajo Futuro

El desarrollo de Advisory nos lleva a la solución inicial que describimos en la sección 3, pero dicha solución tiene una serie de trabajos futuros a lo largo de la tesis doctoral que plantean extender la herramienta para dar un mayor soporte al propósito del análisis de vulnerabilidades en proyectos de desarrollo software. Estos son:

- **Desarrollar nuevas operaciones**: Implementaremos nuevas operaciones que den a la persona que use la herramienta información relevante a la seguridad

de su proyecto. Una de estas operaciones futuras es analizar la deuda técnica de las configuraciones de dependencias de nuestro proyecto o por ejemplo recomendar el uso de dependencias en base a lo vulnerables que sean. Esto incluye el uso de nuevos sistemas de IA para extraer información relevante.

- **Añadir nuevos gestores**: Actualmente nuestra herramienta trabaja con el gestor de paquetes PyPI, ideamos implementar en el futuro nuevos gestores de paquetes tales como Maven para proyectos Java, NPM para proyectos JavaScript o Composer para proyectos PhP.

- **Añadir nuevas bases de datos**: Hasta el momento realizamos búsquedas de vulnerabilidades la *National Vulnerability Database (NVD)*[1] del NIST perteneciente al gobierno de los EE.UU, de cara al futuro implementaremos otras bases de datos de otros gobiernos como la *Japan Vulnerability Notes (JVN)*[2] del gobierno japonés o la *Chinese National Vulnerability Database (CNVD)*[3] del gobierno de China. Para incrementar así la cantidad de vulnerabilidades que podemos detectar y mitigar las inconsistencias entre bases de datos.

- **Desarrollar una interfaz de usuario**: En el futuro daremos soporte a los usuarios para que puedan utilizar las funcionalidades que implementaremos de forma web. Esta además podría integrar un CI/CD para la gestión de la integración continua que lance una serie de pruebas de vulnerabilidades en los repositorios de los proyectos. O incluso pull requests a los repositorios de software modificando los ficheros de dependencias para eliminar versiones vulnerables.

- **Análisis del código fuente**: Aumentaremos las funcionalidades de la herramienta para analizar las vulnerabilidades mas allá de las dependencias, es decir en el código fuente de los proyectos software. Analizaremos la variabilidad de las diferentes soluciones, para una vulnerabilidad en código y sugeriremos la mas adecuada respecto a unas métricas.

- **Reducción de la variabilidad**: En la comunidad de lineas de productos es sabido que la alta variabilidad de un dominio es problemática [27]. Debido a que hace imposible el analizar todo el espacio completo de configuraciones, por eso se ha diseñado un límite de devolución de configuraciones para las

[1]https://nvd.nist.gov/
[2]https://jvndb.jvn.jp/en/
[3]https://www.cnvd.org.cn/

operaciones existentes. Implementaremos otros métodos que poden el espacio de configuraciones con las restricciones que el usuario desea, por ejemplo las 3 versiones más recientes por dependencia.

- **Grafos completos**: Usaremos una base de datos que almacene los grafos de dependencias y más allá acumule los resultados de búsqueda de los usuarios para generar grafos cada vez más completos es decir, intentaremos modelar los diferentes repositorios de herramientas de cada lenguaje de programación.
- **Herramienta de análisis de grafos**: Seria interesante que implementáramos herramientas que analizaran los grafos en busca de errores o malas prácticas como dependencias circulares, o simplemente brindar operaciones sobre grafos como caminos mínimos si fueran interesantes para alguna operación de análisis. Para eso haremos uso de nuevas IA para análisis de grafos como las Graph Neural Networks (GNN).
- **Incompatibilidad entre ficheros**: Cada fichero de dependencias puede representar un entorno diferente (producción, desarrollo, pruebas, etc). Implementaremos la capacidad de analizar si estos ficheros son compatibles entre si para determinar si los diferentes entornos en los cuales trabajan los desarrolladores tienen incompatibilidades o problemas entre sí.

Bibliografía

[1] J. R. Jones, "Estimating software vulnerabilities," *IEEE Security Privacy*, vol. 5, no. 4, pp. 28–32, 2007.

[2] T. Yadav and A. M. Rao, "Technical aspects of cyber kill chain," in *Security in Computing and Communications*, pp. 438–452, 2015.

[3] W. Hu, Y. Wang, X. Liu, J. Sun, Q. Gao, and Y. Huang, "Open source software vulnerability propagation analysis algorithm based on knowledge graph," in *2019 IEEE SmartCloud*, pp. 121–127, 2019.

[4] S. Dass and A. S. Namin, "Vulnerability coverage for adequacy security testing," in *35th Annual ACM Symposium on Applied Computing*, SAC '20, (New York, NY, USA), p. 540–543, ACM, 2020.

[5] P. Murthy and R. Shilpa, "Vulnerability coverage criteria for security testing of web applications," in *2018 ICACCI*, pp. 489–494, 2018.

[6] S. M. Perez, V. Cosentino, and J. Cabot, "Model-based analysis of java EE web security misconfigurations," *Comput. Lang. Syst. Struct.*, vol. 49, pp. 36–61, 2017.

[7] J. A. G. Duarte, *Evolution, testing and configuration of variability systems intensive.* PhD thesis, University of Rennes 1, France, 2015.

[8] J. A. Galindo, D. Benavides, and S. Segura, "Debian packages repositories as software product line models. towards automated analysis," in *ACOTA, Belgium, September, 2010*, vol. 688, pp. 29–34, CEUR-WS.org, 2010.

[9] R. Heradio, D. Fernández-Amorós, J. A. Galindo, D. Benavides, and D. S. Batory, "Uniform and scalable sampling of highly configurable systems," *Empir. Softw. Eng.*, vol. 27, no. 2, p. 44, 2022.

[10] J. A. Galindo, D. Benavides, P. Trinidad, A.-M. Gutiérrez-Fernández, and A. Ruiz-Cortés, "Automated analysis of feature models: Quo vadis?," *Computing*, vol. 101, no. 5, pp. 387–433, 2019.

[11] Á. J. Varela-Vaca, R. M. Gasca, J. A. Carmona-Fombella, and M. T. G. López, "AMADEUS: towards the automated security testing," in *24th ACM SPLC '20, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pp. 11:1–11:12, ACM, 2020.

[12] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of software maintenance and evolution: Research and practice*, vol. 19, no. 2, pp. 77–131, 2007.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[14] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software process: Improvement and practice*, vol. 10, no. 1, pp. 7–29, 2005.

[15] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, "Extending feature diagrams with uml multiplicities," in *6th World Conference on Integrated Design & Process Technology (IDPT2002)*, vol. 23, pp. 1–7, 2002.

[16] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "Form: A feature-; oriented reuse method with domain-; specific reference architectures," *Annals of software engineering*, vol. 5, no. 1, pp. 143–168, 1998.

[17] K. P. G. Böckle and F. J. van der Linde, "Software product line engineering: Fundations, principles and techniques," in *Springer–Verlag*, 2005.

[18] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: A temporal graph-based representation of maven central," *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, 2019.

[19] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[20] L. d. Moura, B. Dutertre, and N. Shankar, "A tutorial on satisfiability modulo theories," in *International Conference on Computer Aided Verification*, pp. 20–36, Springer, 2007.

[21] S. Ranise and C. Tinelli, "Satisfiability modulo theories," *Trends and Controversies-IEEE Intelligent Systems Magazine*, vol. 21, no. 6, pp. 71–81, 2006.

[22] A. Tripathi and U. K. Singh, "Taxonomic analysis of classification schemes in vulnerability databases," in *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pp. 686–691, IEEE, 2011.

[23] H. Booth, D. Rike, G. A. Witte, *et al.*, "The national vulnerability database (nvd): Overview," 2013.

[24] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX security symposium (USENIX Security 19)*, pp. 869–885, 2019.

[25] P. Arcaini, A. Gargantini, and P. Vavassori, "Generating tests for detecting faults in feature models," in *2015 IEEE 8th ICST*, pp. 1–10, IEEE, 2015.

[26] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empirical Softw. Engg.*, vol. 21, p. 2268–2297, dec 2016.

[27] J. Bosch, "Towards a new digital business operating system: Speed, data, ecosystems, and empowerment (keynote)," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 2–2, 2018.