Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Mejorando taint analysis para JavaScript con grandes cantidades de código

Licenciatura en Ciencias de la Computación
Tesis de Licenciatura

Pablo Balbi

Director: Diego Garbervetsky

Buenos Aires, 2023

MEJORANDO TAINT ANALYSIS PARA JAVASCRIPT CON GRANDES CANTIDADES DE CÓDIGO

En la actualidad, las técnicas de análisis estático de código son una herramienta muy usada para detectar, de manera temprana, vulnerabilidades de seguridad durante etapas tempranas en el proceso de desarrollo. Taint analysis es un tipo particular de análisis, con prestaciones que lo caracterizan como un gran candidato para detectar fallas muy comunes en aplicaciones web, como son los ataques de inyección. Si bien las técnicas para llevar a cabo este tipo de análisis están ampliamente estudiadas, el mismo depende de una serie de especificaciones que indican qué elementos de un programa podrían estar involucrados en una vulnerabilidad.

En esta tesis se presenta una técnica de inferencia de especificaciones de taint analysis, para el lenguaje JavaScript. La implementación de la misma surge de combinar un método ya existente basado en aprendizaje automático, con el motor de análisis estático CodeQL. La técnica presentada parte un grafo que modela la propagación de datos a lo largo de un programa, y construye un modelo de optimización lineal que resuelve el problema de inferencia de manera escalable. Las especificaciones producidas se expresan en una representación que permite generalizar un fragmento de código, para así poder reconocer casos similares en otros programas.

Además, se presenta una nueva metodología de evaluación que no requiere supervisión alguna, a fin de cuantificar el potencial de la técnica para inferir nuevas especificaciones. Mediante esta metodología, se evalúa el procedimiento presentado en el trabajo, sobre un conjunto de alrededor de setecientos programas afectados por cuatro clases diferentes de vulnerabilidades de seguridad, alcanzando valores de recall cercanos al 80 %.

Palabras Clave: Análisis Estático, Taint Analysis, Aprendizaje Automático, Programación Lineal, Seguridad de la Información.

IMPROVING TAINT ANALYSIS SPECIFICACIONS FOR JAVASCRIPT WITH BIG CODE

Nowadays, static code analysis techniques are widely used tools for detecting security vulnerabilities early in the development stage. Taint analysis is a specific type of static code analysis commonly used to detect security-related problems in web applications, such as injection attacks. While the techniques for conducting taint analysis are well-studied, they depend on a set of specifications that indicate which program elements could be involved in a vulnerability.

This thesis presents a technique, Jeldon, for inferring the specifications required for taint analysis in JavaScript. The implementation combines an existing machine learning-based approach with the static analysis engine CodeQL. Our approach starts with a graph that models data propagation throughout a program, then, it constructs a linear optimization model which is capable of solving the inference problem required to build the specifications in a scalable manner. The produced specifications are expressed in a representation that allows the generalization of a fragment of code, enabling the analyzer to recognize similar patterns in other programs.

Additionally, we present a new methodology for quantitatively evaluating the Jeldon technique. This new methodology allows us to measure how well it performs inferring new specifications, without requiring any supervision. Using this methodology, we evaluated Jeldon on approximately 700 programs affected by four different security vulnerability kinds, achieving a recall of around 80 %.

Keywords: Static Program Analysis, Taint Analysis, Machine Learning, Linear Programming, Information Security.

AGRADECIMIENTOS

Gracias a Vac[1] y a mis viejos, sin quienes esta tesis se habría hecho esperar mucho más. Y doble agradecimiento a mi vieja, quien se dedicó a hacer decenas de revisiones, dar consejo, siempre sin dejar de alentarme. Espero que, a la vez, haya aprendido a valorar el programa analysis.

Gracias a mis amigos de la facultad, de los laburos, del colegio y de la vida... por acompañarme, bancarme y hacer amenos estos ya casi ocho años.

Gracias a Diego, a quien puedo llamar amigo, por acompañarme durante estos tres años de idas y vueltas, inclusive con una pandemia en el medio. Sin él, esta habría sido una tarea imposible.

Gracias a toda la gente del Lafhis, por darme un lugarcito durante estos últimos meses donde trabajar, conocer sobre investigación, y su día a día.

[1] Ana, mi hermana.

Índice general

1. INTRODUCCIÓN

JavaScript, desarrollado por NetScape, es uno de los lenguajes de programación más usados en la actualidad. Según StackOverflow Developer Survey [15], encuesta que se realiza anualmente y es completada por unos noventa mil desarrolladores en todo el mundo, JavaScript es la tecnología más usada en el desarrollo de software, de acuerdo a lo referido por más de la mitad de los encuestados[1]. Este lenguaje se originó en el año 1995, a comienzos de la conocida *dot-com bubble*. Fue originariamente diseñado con el propósito de ser un lenguaje de scripting para navegadores web pero, a lo largo del tiempo, fue mutando y creciendo. Hoy en día, JavaScript es un lenguaje multi-paradigma, usado tanto en aplicaciones web, de escritorio, y de servidor. En este último caso de uso, las aplicaciones del lado del servidor, fueron popularizadas con Node.js[2].

El ecosistema de Node.js provee un gestor de dependencias, llamado npm[3], que tiene disponibles alrededor de 2,5 millones de bibliotecas de código [4]. Comparado con gestores o repositorios de bibliotecas como Maven (Java) o PyPI (Python) que tienen una cantidad de alrededor de 500 mil bibliotecas, npm es de mayor magnitud. En promedio, una aplicación de Node.js depende de más de 50 dependencias tanto directas como transitivas. Tal nivel de dependencia de código externo complejiza la tarea de análisis de código, debido a que el tamaño en bruto del programa aumenta con cada dependencia que se agrega, además de los casos en que ciertas bibliotecas están implementadas en un lenguaje de programación completamente diferente (por ejemplo bibliotecas que acceden a funciones nativas por medio de bindings).

Una aplicación implementada en JavaScript puede ser catalogada en dos grandes categorías: *client-side*, donde su ambiente de ejecución es un runtime del lado del cliente o usuario, por ejemplo un navegador web; o *server-side*, donde la aplicación corre en un runtime como Node.js de manera remota al usuario, e interactúa con él por medio de un protocolo común como HTTP. Existen tipos de vulnerabilidades que afectan a cada una de estas dos variantes de ejecución, aunque normalmente se considera que el grado de severidad suele ser mayor en aplicaciones del lado del servidor. Esto se debe a que el runtime sobre el cual son ejecutadas las aplicaciones client-side suele estar situado dentro de un sandbox[5], mecanismo que restringe las interacciones que la aplicación puede tener con el sistema operativo. En cambio, las aplicaciones server-side suelen tener contacto directo con el sistema operativo o el sistema de archivos del host donde se ejecutan. Esto da lugar a una categoría de vulnerabilidades conocida como **injection attacks**, clasificada en tercer lugar según el conocido ranking de clases de vulnerabilidades OWASP Top 10 [1]. Un ataque de inyección o *injection attack* ocurre, por ejemplo, cuando datos provistos por un usuario alcanzan puntos críticos dentro de un programa sin ser sanitizados, donde se escribe un archivo, se ejecuta una consulta contra una base de datos o se ejecuta un

[1] Más exactamente, según el 63,61 %.

[2] Node.js es uno de los runtime de JavaScript más conocidos y usado para aplicaciones desarrolladas en JavaScript del lado del servidor. Fue diseñado partiendo del runtime open-source Google V8.

[3] npm

[4] Modulecounts

[5] En seguridad informática, el aislamiento de procesos o *sandboxing*, es un mecanismo de seguridad para separar o aislar programas en ejecución, normalmente para mitigar los fallos del sistema operativo donde son ejecutados y para prevenir propagación en las vulnerabilidades del software.

fragmento de código enviado por el usuario.

El análisis de programas o *program analysis* es una técnica que estudia programas que toman a otro programa como entrada, y producen cierta información o validación acerca del mismo. Es un hecho habitual utilizar program analysis para la detección temprana de errores, antes que una aplicación dada sea expuesta ante sus usuarios finales. Existen dos variantes de program analysis que se diferencian en la manera en que el analizador interactúa con el programa bajo análisis: estática y dinámica. En el primer caso, el analizador razona sobre el programa sin que este sea ejecutado, mientras que en el segundo caso, el programa de entrada es ejecutado, y el analizador trabaja mediante instrumentación o interactuando con una versión modificada del runtime. Esta tesis se apoya en el análisis estático, ya que este permite sobre-aproximar el comportamiento de un programa a través de todas sus posibles ejecuciones, pudiendo proveer garantías sobre cierta clase de errores. A diferencia de este, el análisis dinámico solo puede razonar sobre ejecuciones concretas, por lo que no puede proporcionar el mismo tipo de garantías.

Existe a la vez un tipo de análisis estático de código denominado **taint analysis**, que es utilizado para detectar de manera temprana distintos tipos de vulnerabilidades, como los ataques de inyección antes mencionados. El mismo busca descubrir una fuente de datos no segura (**sources**), donde se puede originar un dato que termine en un elemento crítico del programa (**sinks**). Por otro lado, si en alguno de estos flujos de información desde un source a un sink existe algún tipo de sanitización de los datos provenientes de un usuario (**sanitizers**), estos son considerados como seguros.

Una implementación de taint analysis consta de dos partes principales: el motor del análisis de código, encargado de realizar la ejecución abstracta del programa donde se busca detectar si existe o no algún flujo de datos no sanitizado, y una serie de especificaciones (**taint specifications**) que indican qué elementos o términos de un programa deben ser considerados como fuentes de datos no seguros, sanitizadores o puntos críticos (sources, sanitizers y sinks, respectivamente). En la práctica, estas especificaciones son por lo común escritas manualmente, por medio del estudio de documentación o código fuente de los programas. Este comportamiento constituye una tarea laboriosa y propensa a errores, y, por consiguiente, conlleva a que la falta de cobertura de casos, o falsos positivos sea un fenómeno común.

El uso de estas técnicas de análisis de código se volvió más accesible y masivo en los últimos años. Plataformas como Snyk o CodeQL proveen herramientas que permiten usar analizadores de código estático modernos de manera muy simple, ya sea en un entorno de desarrollo local, o en plataformas de integración continua. Para ilustrar este tipo de integraciones, la figura 1.1 muestra una alerta detectada por CodeQL, y visible para el desarrollador dentro de la plataforma de gestión de cambios de código Github. CodeQL es un motor de análisis de código estático que no solo permite ejecutar diferentes análisis como taint analysis, sino también desarrollarlos mediante un lenguaje de consultas declarativo llamado llamado QL.

1.1. Aportes

Esta tesis se enfoca en la problemática de generación automática de especificaciones de taint analysis. Para ello, se basa, particularmente, en el trabajo realizado en Seldon [3], que presenta una técnica de inferencia de estas especificaciones a partir de un corpus de código open source, y un conjunto pequeño de especificaciones iniciales.

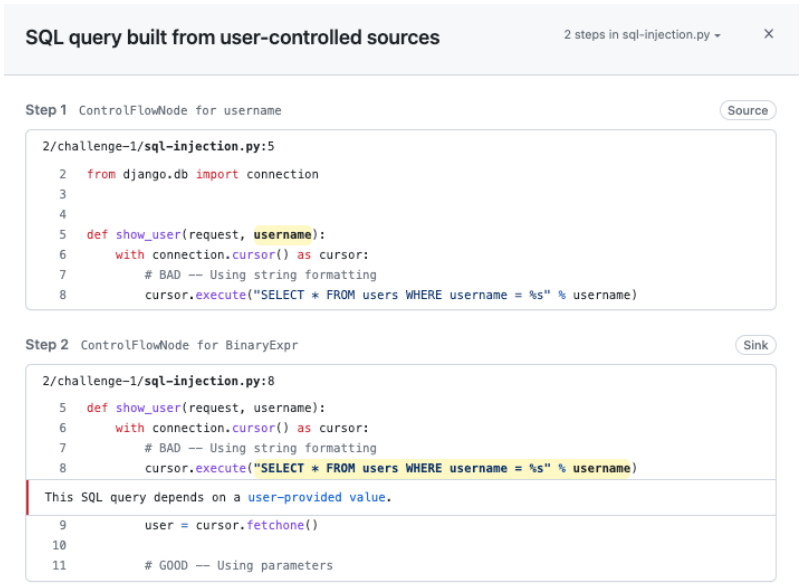

Fig. 1.1: CodeQL usado en la plataforma de integración continua Github Actions. Fuente.

Este trabajo brinda diversos aportes que pueden a la vez ser insumo de gran utilidad:

- Se implementa la técnica presentada en Seldon para JavaScript, con el objetivo de mejorar las especificaciones de taint analysis en CodeQL
- Se presenta una metodología de evaluación no supervisada. Esta nos permite evaluar cuantitativamente la técnica, sin necesidad de corroborar una a una, manualmente, la validez de las especificaciones inferidas
- Una evaluación cuantitativa de la técnica por medio de la metodología antes mencionada

Algunas secciones de lo presentado en este trabajo de tesis constituyen partes de un proyecto más amplio, y han sido antes publicadas en *ICSE-SEIP '22: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* [4]. Dicha publicación se enfoca principalmente en una metodología que, con intervención de un experto, puede mejorar y filtrar de manera inteligente las inferencias obtenidas por medio de la técnica descripta en esta tesis.

1.2. Metodología

La técnica presentada en este trabajo fue desarrollada manteniendo la formulación original del problema de inferencia. Como los lenguajes de programación bajo estudio son de cierta similitud aunque diferentes, la técnica original fue adaptada para su correcto funcionamiento en el nuevo lenguaje estudiado. Esto fue documentado, y, toda vez que fue posible, fue evaluado empíricamente. Una vez que la técnica fue implementada en su totalidad, se aplicó una metodología de evaluación para medir la eficacia de manera cuantitativa. Con dicha metodología en curso, se inició un proceso iterativo de prueba de mejoras a nuestra propia implementación, con el propósito de que todos los cambios se vieran reflejados en los resultados de la evaluación empírica.

1.3. Organización general del trabajo

Este trabajo está organizado en dos partes principales y sus anexos.

La primera parte, está compuesta por los capítulos 2 y 3. El capítulo 2 introduce el marco conceptual, explica las tecnologías, conceptos e ideas centrales del desarrollo de la técnica. Luego, el capítulo 3 presenta y explica el trabajo realizado en Seldon [3], ya que nuestra investigación se apoya en el mismo, con el fin de poder referenciar y diferenciar sus partes a través de nuestra implementación. Dicha sección también explicita cuestiones conceptuales del trabajo original.

La segunda parte presenta los resultados, y está compuesta por los capítulos 4, 5 y 6. En el capítulo 4 se explican los pasos llevados a cabo durante la aplicación de nuestra técnica. La sección conserva una estructura similar a la del capítulo previo, a fin de facilitar la comparación de las partes en ambas implementaciones. Una vez explicado el trabajo original y nuestra implementación, en el capítulo 5 se describe cómo se evalúa si la técnica puede o no ser utilizada para inferir nuevas especificaciones de taint analysis. Además, este capítulo presenta una metodología de evaluación que permite corroborar el buen funcionamiento de la técnica de manera no supervisada. Por último, el capítulo 6 resume las ideas principales del trabajo, y las relaciona con otros en la misma línea de investigación. Finalmente, a modo de cierre, se sintetizan los aspectos más relevantes del trabajo, sus aportes al estado del arte, y se presentan algunas ideas sobre proyectos ulteriores, que complementen estos hallazgos, y a los que consideramos futuras líneas de investigación.

En último lugar, el anexo 7 presenta algunos detalles de implementación de la técnica, relacionados con CodeQL.

2. MARCO TEÓRICO

2.1. JavaScript

JavaScript es un lenguaje de programación interpretado o compilado justo-a-tiempo (just-in-time), con funciones de alto orden, prototipado (con soporte de clases agregado en los últimos años), multi-paradigma, single-threaded y con un sistema de tipos dinámico. Entre los diferentes paradigmas de programación, puede ser considerado como orientado a objetos, imperativo y funcional.

JavaScript surgió en 1995 como un lenguaje para scripting de navegadores web, y hoy en día se usa tanto en entornos dentro del navegador como fuera del mismo. Un runtime muy conocido para este último es NodeJS.

Según un reporte anual realizado por Github [6], JavaScript es el lenguaje más usado actualmente, y consecutivamente desde el año 2014.

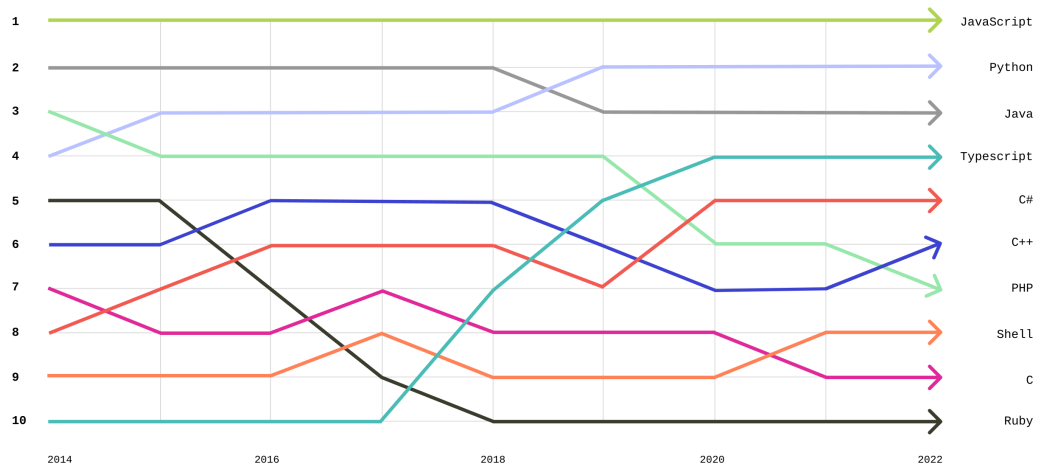

Fig. 2.1: Lenguajes más usados a través del tiempo, según el reporte GitHub Octoverse 2022 [6]

Una de las prácticas actuales de desarrollo de software que permite el trabajo eficiente, es la componibilidad de diferentes programas para formar uno. La forma más común de lograr esta estrategia de construcción de software, es mediante el uso de bibliotecas de código, que un programa *importa*, y consume sus funcionalidades a partir de una interfaz bien definida (API, o Application Software Interface). Esta práctica es muy común en proyectos desarrollados en JavaScript, tanto que, en promedio, un proyecto depende de mil otros módulos, y no es raro encontrar árboles de dependencias con dos mil nodos[1].

Para darse una idea de lo fácil que es desarrollar un proyecto con un árbol de dependencia medianamente grande, el snippet de código JavaScript en la figura 1 implementa un servidor HTTP extremadamente simple, que es solamente capaz de responder *hello world*.

Al consultar al gestor de bibliotecas de NodeJS npm cuántas dependencias, incluyendo

[1] `https://blog.npmjs.org/post/180868064080/this-year-in-javascript-2018-in-review-and-npms.html`

Listing 1 Implementación del *hola mundo* de los servicios web HTTP en NodeJS. Disponible en Github.

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

transitivas, tiene el ejemplo anterior, estas son 58, solo para una implementar un "hola mundo" y con 11 líneas de código.

JavaScript se caracteriza por ser un caso complejo, interesante y a la vez un gran desafío para el estudio de análisis estático de código, ya que presenta cualidades tales como cantidad variable de argumentos en una función, funciones de alto orden, introspección de tipos de datos (también conocido como *reflexión*), creación y evaluación dinámica de texto como código mediante el operador `eval`, entre otras características. La tesis "Static Analysis for NodeJS" [14] realiza un análisis más exhaustivo de estas.

Más aún, si tomamos en cuenta la heterogeneidad en los ambientes de ejecución donde las aplicaciones desarrolladas en JavaScript suelen ser ejecutadas, surge una complejidad extra: Por ejemplo, los programas desarrollados para navegadores web dependen del motor de ejecución que el navegador implemente, y hacen uso del DOM[2], una API que el navegador provee para interactuar con la interfaz gráfica que el sitio web presenta al usuario. La dificultad con esta API, que es usada regularmente en **todo** sitio que presenta una funcionalidad interactiva, es que se encuentra implementada de manera nativa por el navegador, por lo que analizarla estáticamente, es necesario tener un modelo del comportamiento de la misma.

2.2. Taint Analysis

Debido al amplio uso de JavaScript en diversas aplicaciones y ambientes de ejecución, las aplicaciones desarrolladas en este lenguaje suelen ser un objetivo muy comúnmente elegido por usuarios maliciosos. La figure 2.2 muestra como JavaScript lleva la mayor cantidad de vulnerabilidades (o CVEs[3]) reportadas en comparación con otros lenguajes.

Una forma de detectar estas vulnerabilidades es mediante **taint analysis**. El mismo es un análisis de código que puede detectar flujos de datos caracterizados como maliciosos por un experto en seguridad, o para ciertos criterios de categorización automáticos. Estos enfoques funcionan realizando el seguimiento de información sensible o **tainted** a través de un programa, empezando de un conjunto de fuentes o **sources** pre-definidos. Cuando

[2] `https://developer.mozilla.org/en-US/docs/Glossary/DOM`

[3] CVEs, Common Vulnerabilites and Exposures, es un consorcio que busca definir, encontrar y catalogar, de manera abierta, vulnerabilidades de software.

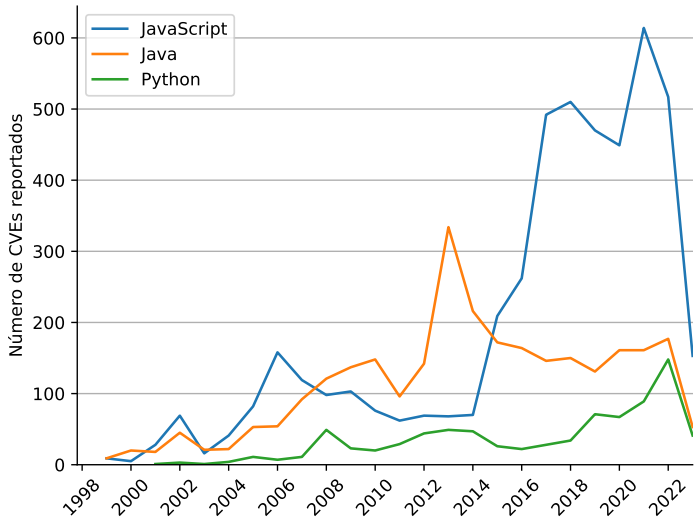

Fig. 2.2: CVEs reportados por año con la palabra clave Java (2633), Python (776) y Javascript (4548). Calculados en Google Collab.

uno de estos sources produce un valor sensible, se realiza un seguimiento del dato buscando si el mismo alcanza un sumidero o **sink**.

Taint analysis puede ser diseñado de forma tal que pueda ser ejecutado como un análisis de código tanto estático como dinámico. Una característica a resaltar de la versión dinámica, es que al ser evaluado mediante la ejecución del código a analizar, es necesario que las entradas del programa usadas para ejecutar el programa bajo estudio, deban ser lo suficientemente completas para lograr una buena cobertura. Caso contrario, el análisis estaría *cubriendo* solo una porción del programa. Por otro lado, los análisis estáticos no sufren de estos problemas ya que no ejecutan el programa a estudiar, sino que lo evalúan abstractamente. Esto hace que, aunque la cobertura sea total, esta pueda ser imprecisa.

Listing 2 Ejemplo de código mostrando una vulnerabilidad de NoSQL injection

```
1 app.get('/search', async (req, res) => {
2   let title = req.query.title;
3   let data = await db.collection('movies').findOne({
4     $where: `this.title == "${title}"`,
5   })
6   res.json({
7     data: data,
8   });
9 });
```

En el snippet de código 2 se muestra un ejemplo de una vulnerabilidad de NoSQL Injection. El extracto implementa una API mediante la cual un usuario puede buscar dentro de la colección *movies* una película, por su título. En la línea 2 se puede ver que el título de la película a buscar es recibido mediante los query parameter de la URL del request,

luego utilizada para armar la consulta NoSQL. Si un usuario solicita la película titulada `"Volver al futuro"` luego la API responderá la película si la misma se encuentra disponible. Pero, si un usuario malicioso interactúa con nuestra aplicación, podría enviarnos el string `'" || "" == "'` el cual hace que la query resultante sea equivalente a la siguiente reducción:

```
let query = {$where: `this.title == "${title}"`}
query = {$where: `this.title == "" || "" == ""`}
query = {$where: `this.title == "" || true`}
query = {$where: `true`}
```

De esta forma, la query resultante devolvería todos los elementos guardados en la colección. Mediante un string formado para este propósito, un atacante puede tomar ventaja del hecho de que nuestra aplicación no realiza ningún tipo de validación o sanitización sobre los pedidos enviados por usuarios externos, haciendo que la aplicación se comporte en formas no esperadas por el desarrollador. Si analizáramos este código por medio de un taint analysis, `req.query.title` en la línea 2 sería un source, y el argumento `query` en la líneas 3-5, dentro de `db.collection(name).findOne(query)`, sería un sink.

2.3. CodeQL

CodeQL [5] es un motor de análisis de código desarrollado originalmente por Semmle[4] y luego adquirido por Github. Es usado muy ampliamente a través de la plataforma de desarrollo provista por Github, en los sistemas de integración continua, y también localmente por desarrolladores y expertos en seguridad informática. En CodeQL el código es tratado como datos. Las vulnerabilidades de seguridad, bugs, y otro tipo de errores son modelados como consultas o queries, que pueden ser ejecutadas sobre, bases de datos extraídas del código fuente de programas. Estas bases de datos contienen información que puede ser consultable, una vez extraída de un programa, implementado en un solo lenguaje, y en un instante en el tiempo. La base de datos contiene una representación del código a diferentes niveles de abstracción, tales como a nivel de abstract syntax tree, a nivel del grafo de data flow y control flow. Esto permite que una consulta no solo exprese restricciones o condiciones sobre un único nivel de abstracción, sino muchos e interrelacionados entre sí.

Además del motor de análisis de código, las consultas son descriptas en un lenguaje diseñado para tal fin, llamado QL [13]. El mismo es un lenguaje de consulta, declarativo y orientado a objetos, optimizado para realizar consultas sobre las bases de datos antes descriptas. Inspirado en Datalog, cualquier operación descripta en QL es una operación lógica entre conjuntos de datos. Por otro lado, la orientación a objetos de QL permite diseñar consultas de manera modular, con ocultamiento de información, lo cual le permite al ecosistema mejorar mucho la reusabilidad de código y hace muy común el desarrollo de consultas reutilizables en forma de bibliotecas [5].

Los siguientes ejemplos muestran consultas en QL a niveles diferentes de abstracción. En primer lugar, la siguiente consulta busca, en el AST del programa cuya base de datos

[4] `https://en.wikipedia.org/wiki/Semmle`

[5] Para ver un ejemplo de bibliotecas de consultas, CodeQL mantiene en su repositorio una colección de módulos de JavaScript modelados en QL de forma de poder predicar sobre primitivas de los mismos, por ejemplo sobre el manejo del DOM en aplicaciones que son ejecutadas en el navegador web.

estamos consultando, los condicionales cuya rama positiva esté vacía, es decir, que no contenga ninguna sentencia de código.

```
from IfStmt ifstmt, BlockStmt block
where ifstmt.getThen() = block and block.isEmpty()
select ifstmt, "if redundante"
```

Podemos luego subir más en el nivel de abstracción y predicar sobre el control flow graph (CFG). La siguiente consulta busca nodos dentro del CFG que no son alcanzables. Es decir, partiendo del punto de entrada del programa, v_{entry}, busca nodos v parte del CFG, tales que, no existe un camino entre v_{entry} y v.

```
from ControlFlowNode n
where n.isUnreachable()
select n, "nodo no alcanzable en el CFG"
```

Más aún, podemos realizar una consulta sobre el flujo de datos dentro del programa. El siguiente ejemplo busca un nodo que sea parte del grafo de dataflow (`DataFlow::Node`), de tal modo que haya un camino desde el mismo, hasta el primer argumento de una llamada a función, cuya estructura sintáctica sea parecida a `require("fs").readFile`. En este ejemplo, podemos notar una característica interesante del lenguaje: El mismo nos permite predicar sobre la clausura transitiva de una relación, como se puede ver en la línea 3 con `source.getASuccesor*()`.

```
1 from DataFlow::Node source, DataFlow::CallNode readFile
2 where
3     DataFlow::moduleMember("fs", "readFile").getACall() = readFile and
4     source.getASuccessor*() = readFile.getArgument(0)
5 select source, "valor que puede ser usado para leer un archivo"
```

El ejemplo anterior muestra una consulta que predica sobre dataflow local a una función, es decir, que las relaciones de flujo de datos tomadas en cuenta, sean parte de una misma función. Este sabor de dataflow también es conocido como intra-procedural. CodeQL permite también realizar consultas a nivel inter-procedural, es decir, a través de llamadas a función. Para ello se hace uso de la faceta orientada a objetos del lenguaje, definiendo en una *configuración*, cómo son los elementos que queremos encontrar a lo largo del flujo. El siguiente snippet muestra la misma consulta del ejemplo anterior, pero de manera inter-procedural. Esto se define indicando tan solo en dos predicados cuál es la forma que tienen los nodos que queremos considerar como sources, `isSource`, y sinks, `isSink`.

```
// Definición de la configuración
class FileReadConfiguration extends TaintTracking::Configuration {
  FileReadConfiguration() { this = "FileReadConfiguration" }

  // Este predicado caracteriza los sources en los que estamos interesados
  override predicate isSource(DataFlow::Node source) {
    exists(DataFlow::PropRead requestURL |
      requestURL.getPropertyName() = "url"
```

```
      and source = requestURL
    )
  }

  // Este predicado caracteriza los sinks en los que estamos interesados
  override predicate isSink(DataFlow::Node sink) {
    DataFlow::moduleMember("fs", "readFile")
        .getACall()
        .getArgument(0) = sink
  }
}

// Consulta
from FileReadConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

Dataflow restringe los casos donde se considera que hay "flujo" entre dos elementos de un programa. Este toma en cuenta solamente casos donde un valor se mueve a lo largo de diferentes asignaciones, llamadas a función, o dentro de objetos, pero sin sufrir mutaciones. El siguiente fragmento de código ejemplifica un caso donde *dataflow* no consideraría la asignación en la línea 3 en su grafo subyacente.

```
1 app.get('/transact', async (req, res) => {
2   let token = req.query.token; // asignación tenida en cuenta
3   let token2 = attachUserID(token); // asignación ignorada
4   // ...
5 });
```

En el ejemplo, se muestra una implementación de un *handler* de pedidos HTTP. Los parámetros de la función anónima, cuya declaración arranca en la línea 1, son `req` un request HTTP, y `res` un manejador que permite enviar una respuesta al pedido. La información recibida por medio del request, `req`, es comúnmente considerada como sensible o *tainted*, ya que proviene de un agente externo al programa. En la línea 2 podemos notar que se asigna a la variable `token` un parámetro extraído de la URL[6] llamado "token". Luego, la siguiente asignación almacena el valor resultante de la llamada a la función `attachUserID` a la variable `token2`. Esta llamada a función involucra a la variable `token`, que sabemos tiene un valor potencialmente peligroso. Sin embargo, en el caso en que la función llamada realice alguna modificación sobre el valor proveniente de su argumento, dataflow descartaría esta asignación de su grafo, a pesar de que el valor de `token2` provenga de un dato marcado. Estos casos son tenidos en cuenta por una variable de dataflow conocida como **taint tracking**.

[6] También conocidos como query parameters.

3. TRABAJO PREVIO: SELDON

Antes de adentrarnos en los detalles de implementación de nuestra técnica, es necesario mencionar e introducir a Seldon [3], trabajo en el cual se apoya la técnica presentada en esta tesis.

Primero, definimos una terminología que será usada a lo largo de este trabajo.

Definición 3.0.1 (Término de programa)**.** Un *término de programa*, o solamente *término*, también conocidos en la literatura como *program element*, hace referencia a un nodo dentro del grafo de control flow de un programa. Refiere a elementos que forman parte del mismo, tales como una asignación, una llamada a función, la guarda de un condicional, entre otros.

Seldon presenta un técnica semi-supervisada de inferencia de taint specifications para bibliotecas de código parcialmente o no modeladas, cuya idea principal es plantear el problema de inferencia, como uno de programación lineal. Este proceso de inferencia parte de un conjunto de programas $\mathcal{D}$, y un pequeño conjunto de términos $\mathcal{A}_M$ que poseen anotaciones que indican qué rol cumplen dentro de taint analysis. Seldon infiere especificaciones sobre el conjunto restante de términos no categorizados $\mathcal{A}_U$, el cual es mucho más grande. Este proceso de inferencia está compuesto de cuatro etapas, las cuales analizaremos más en profundidad:

1. Capturar de cada programa en $\mathcal{D}$, información de cómo fluyen los datos (*information flow*), en forma de un grafo al que llamaremos *propagation graph*.
2. Traducir los vértices del grafo, de términos de programa a una representación independiente al programa del cual pertenecen.
3. A partir del propagation graph, construir un sistema lineal que modele el problema de inferencia de especificaciones de taint analysis.
4. Resolver el modelo lineal, encontrando roles de taint analysis a los términos en $\mathcal{A}_U$.

Las siguientes secciones explican en detalle cada una de las etapas que componen Seldon. Mientras que este último fue implementado y evaluado para Python, nuestro trabajo adapta la técnica para JavaScript, como se describe en el capítulo 4.

3.1. Propagation Graph

El *propagation graph* es un grafo extraído de un programa, que representa cómo fluye la información entre diferentes términos. Es utilizado para construir el modelo de optimización lineal, que define el problema de inferencia. La siguiente definición presenta cómo esta conformado este grafo.

Definición 3.1.1 (Propagation graph)**.** Un **propagation graph**, o **grafo de propagación**, es un grafo dirigido $G = \langle V, E \rangle$, donde los los vértices en V son **términos relevantes** y un eje $(e_1 \to e_2) \in E$ indica que hay **flujo de información** del término e_1 hacia e_2. La definición de término relevante y qué es considerado como flujo de información será

diferente entre Seldon y nuestra implementación, por lo que será especificado más adelante en las secciones correspondientes.

Una noción de nomenclatura que queremos destacar es que, en la publicación original, los términos de un programa considerados como relevantes son llamados *eventos*. En esta tesis no haremos uso de la misma terminología, y utilizaremos solamente el concepto de *término*.

Seldon trata el problema de inferencia de taint specifications para Python, un lenguaje con caraterísticas similares a JavaScript a nivel de program analysis, como fue visto en la sección 2.1. En la definición 3.1.1 definimos los vértices del propagation graph como términos *relevantes*. En la publicación original, un término es considerado como relevante si es de alguno de los siguientes tipos:

- Llamadas a función
- Lectura de atributos de un objeto
- Argumentos en la definición de una función

Una vez capturados los términos relevantes que definen el conjunto de vértices del grafo V, los ejes del grafo, encargados de capturar el flujo de información, se generan de acuerdo a las siguiente reglas:

1. En una llamada a función, si el cuerpo de la función llamada puede ser determinado estáticamente, se hace un *inlining* del mismo.
2. Si la función pertenece a una biblioteca externa, y asumiendo que la función tiene n argumentos y retorna un valor v_{return}, se agrega un eje $(arg_i \rightarrow v_{return}) \forall i \in \{1, ..., n\}$. Esto es una sobre-aproximación del grafo de information flow verdadero.
3. Mediante **points-to analysis**, se completa el grafo de forma que si dos vértices $a, b \mid a \in PointsTo(b)$, se agrega un eje $(b \rightarrow a)$.

Points-to analysis o **pointer analysis** es un tipo de análisis estático de código que busca determinar información sobre qué valores es asignada una variable. Una vez obtenida, esta información puede ser interpretada con un mapa estático del **heap** o memoria de un programa. Como el heap es la estructura principal de memoria global de un programa, points-to analysis es uno de los tipos de análisis fundacionales para construir análisis inter-procedurales.

El algoritmo de points-to utilizado en Seldon es una variante de Andersen [2], construido de forma tal que tiene las siguientes características: es flow-sensitive, field-sensitive y context-sensitive con un contexto acotado de hasta 8 llamadas a función. No es un objetivo de esta tesis definir formalmente dichas propiedades de pointer-analysis. Las mismas se encuentran bien descriptas en [16]. Aunque para facilitar una idea al lector sobre las capacidades del análisis utilizado en Seldon, presentamos a continuación algunos conceptos fundamentales de cada propiedad:

- Flow sensitivity: El análisis toma en cuenta estructuras de control de flujo.

- Field sensitivity: El análisis puede distinguir entre diferentes atributos de un objeto. En caso contrario, si el análisis carece de field sensitivity, asignaciones o lecturas a atributos de un objeto serán consideradas como si fueran al objeto receptor, y no a "partes" del mismo.
- Context sensitivity: El análisis modela, de alguna forma, el contexto donde ocurren asignaciones. Esto le permite diferenciar, por ejemplo, una asignación que ocurre dentro de una llamada a función, en dos contextos de llamadas diferentes.

Es interesante observar que no se consideran relevantes a los *property write*, tales como la escritura atributos de un objeto, ya que, de acuerdo a la publicación, estos raramente actúan como sinks, y no pueden ser considerados como sources o sanitizers. Sin embargo, el algoritmo de points-to es field-sensitive. Esto quiere decir que el mismo diferencia operaciones STORE o LOAD en los atributos de un objeto, en vez de agregarlas como si el objeto no tuviera atributos [16]. De esta manera, la relación de $PointsTo$ captura implícitamente el flujo de información de las escrituras a atributos de un objeto.

Una vez construido el grafo G_i para cada programa $P_i \in \mathcal{D}$, Seldon utiliza para la construcción del modelo el grafo $G = \langle \cup V_i, \cup E_i \rangle$ resultante de la unión del propagation graph de cada programa. Es interesante notar que el conjunto de vértices de cada grafo V_i contiene términos pertenecientes a diferentes programas, y por lo tanto, son disjuntos par en par. Es decir, $V_i \cap V_j = \emptyset$ con $i \neq j$.

La sección siguiente introduce una forma de expresar términos de programa de manera que, una vez construido el modelo lineal, dos términos que en nuestro dominio de taint analysis deberían ser el mismo vértice, tengan igual representación y, por consiguiente, la misma variable dentro del modelo.

3.2. Representación programa-independiente

Al inferir nuevas especificaciones, es necesario tener una representación que permita identificar un mismo término a través de diferentes programas, ya que de otra forma, un término cuyo rol fue inferido, se vería restringido al programa del cuál surgió durante la construcción del modelo.

Para dar un ejemplo, veamos el snippet de código 3, y supongamos que Seldon infiere que el primer argumento de la llamada a función en la línea 6, `findOne(query)`, es un sink. Para que esta nueva especificación sea útil, es necesario expresarla con una notación mediante la cual esta pueda ser reconocida en otros programas, ya que de utilizar, por ejemplo, la ubicación dentro del archivo fuente, esta tendría una única ocurrencia, la de 3.

A este tipo de representaciones, que pueden expresar un término t de manera lo suficientemente genérica para poder identificar diferentes, pero "parecidas", ocurrencias del mismo, las designaremos **representación programa independiente**.

Definición 3.2.1 (Representación programa independiente)**.** Formalizando, una representación programa independiente es una manera de expresar términos de programa, definida por una función $\theta : \mathcal{T} \to \mathcal{R}$, donde $\mathcal{T}$ es el universo de de términos de programa. La función toma un término t proveniente de un programa ρ, y construye una representación $\theta(t) = r$. Esta representación r es lo suficientemente genérica para que si otro programa $p^\star$ tiene un término $t^\star$, el cual tiene ciertas características en común con t, luego valga $\theta(t^\star) = r^\star = r = \theta(t)$. De esta forma θ nos permite relacionar términos "parecidos" entre programas diferentes.

Listing 3 Ejemplo simplificado del programa ethereumclassic/explorer, perteneciente a routes/contract.js

```
1  const mongoose = require('mongoose');
2  const Contract = mongoose.model('Contract');
3
4  exports.findContract = function (address, res) {
5    const query = { address };
6    const contractFind = Contract.findOne(query).lean(true);
7    contractFind.exec((err, doc) => {
8      // ...
9    });
10 };
```

Para esto, la publicación original define una función $Rep(t)$, la cual produce todas las posibles representaciones que un término t podría tener. Como tanto Python (lenguaje estudiado por Seldon) y JavaScript no son lenguajes estáticamente tipados, es difícil determinar estáticamente la signatura de llamadas a función, y por lo tanto, la función en sí. Por ello, $Rep(t)$ captura un conjunto de expresiones que representan a t, y no una sola.

Tomando como ejemplo el supuesto sink inferido por Seldon en la línea 6 de 3, éste podría expresarse de la siguiente manera:

```
findContract(param address, param res).Contract.findOne(param query)
```

Por otro lado, la representación podría agregar un mayor o menor grado de contexto, pudiendo capturar más o menos términos de otros programas que no sean el/los usados durante el proceso de inferencia:

1. `Contract.findOne(param query)`
2. `findOne(param query)`

Estas otras dos presentaciones son determinadas (1) ignorando que la llamada a `findOne` se encuentra dentro de del cuerpo de una función `findContract` , y (2) además omitiendo que la función sospechada es miembro de un objeto `Contract` .

3.3. Construcción del modelo

Habiendo construído el propagation graph, debemos confeccionar un modelo que nos permita resolver el problema de inferencia. La idea clave detrás de Seldon, que le da la capacidad de aplicar esta técnica de apredizaje a escala, es formular el problema de inferencia como uno de optimización lineal. Esto permite que se pueda resolver de manera eficiente y escalable mediante el uso de solvers.

3.3.1. Intuición

Para expresar el problema de inferencia como un modelo de optimización lineal, es necesario definir los siguientes componentes: un conjunto de variables, parte de las cuales

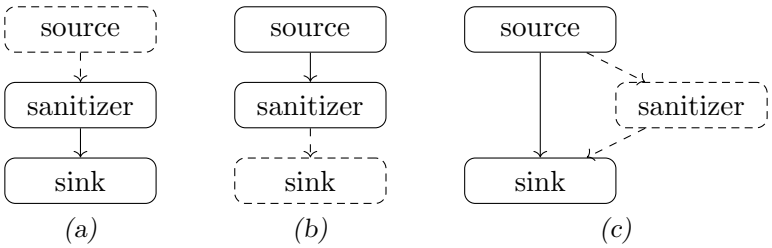

Fig. 3.1: Intuición con la cual son construidas las restricciones del modelo de optimización lineal en [3]

tendrán un valor conocido, y otra parte serán incógnitas; una serie de restricciones que predican sobre el valor de las variables, y cómo se relacionan entre sí; y por último, una función objetivo.

En el caso de Seldon, la función objetivo será definida en la sección 3.4. Esta sección se centra en explicar la intuición detrás de las variables y restricciones entre ellas.

La idea detrás del modelo implementado en Seldon, consiste en predicar sobre patrones comunes acerca de cómo se comporta el flujo de información entre diferentes términos de un programa. Por ejemplo, supongamos que tenemos tres términos a, b y c, y sabemos que dentro de nuestro programa la información fluye en el siguiente sentido: $a \to b \to c$. Supongamos que sabemos que c es un sink, por ejemplo, el argumento de una llamada a una función de escritura de archivos, que controla la ruta del mismo. Además, supongamos que b es una llamada a una función que sanitiza la ruta de un archivo, no permitiendo valores que un potencial atacante querría usar como `"/etc/passwords"`. Si este fuera el caso, entonces debe ser que a, o algún otro término previo a este, debe de ser una posible fuente de valores externos al programa, razón por la cual la función de sanitización fue añadida.

Este tipo de patrones que dados dos términos con un rol conocido, relacionados con un tercero del cual no se conoce el rol, es el utilizado por Seldon para caracterizar el problema de inferencia.

En 3.1 se ilustra cada una de los tres tipos de restricción. Cada columna representa un caso diferente sobre cómo debería ser el flujo de datos entre tres vértices del grafo, no necesariamente consecutivos, sabiendo que dos tienen un rol asignado, y uno es desconocido. A continuación, veremos más en profundidad uno de estos casos.

La figura 3.1a nos dice, coloquialmente, que si en un programa existe un sanitizer, que tiene flujo de información hacia un sink, es muy probable que el primero esté sanitizando los valores producidos por un source. Luego, las restricciones producidas con esta intuición dicen que si un vértice del grafo de propagación v_1 tiene flujo de información hacia otro v_2, es decir, son vecinos o existe un camino entre ellos, si clasificamos v_1 y v_2 como sanitizer y sink respectivamente, luego debemos asignar el rol de source a algún término v con flujo de información hacia v_1.

De manera análoga, la figura 3.1b indica que si en el grafo existen dos vértices v_1 y v_2, vecinos o con un camino entre ellos, y se les asigna un rol de source y sanitizer respectivamente, luego debemos asignar el rol de sink a algún vértice v tal que sea vecino de v_2, o exista un camino entre ellos.

Por último, la figura 3.1c completa las tres intuiciones, predicando sobre el mismo caso para un source y un sink conocidos, donde, en el medio, debería existir un sanitizer.

3.3.2. Variables

Como primer paso para formalizar el modelo planteado en Seldon, es necesario contar con una forma de traducir los vértices del grafo usados en las restricciones antes descriptas coloquialmente. Para ello, se utiliza la representación descripta en la sección 3.2. Dado un vértice $v \in V$, y sea n una posible representación $n = Rep(v)$, se introducirán en el modelo tres variables n^{src}, n^{san} y n^{snk} que representan una puntuación que indica qué tan probable es que v sea un source, sanitizer o sink. Una vez optimizado el modelo, el valor resultante en cada una de estas variables no será verdaderamente un valor de probabilidad, sino que, en la publicación original, se agrega una restricción al modelo que permita *interpretar* estos valores como probabilidades:

$$0 \leq (n_r)^{rol} \leq 1 \quad \forall r \in Rep(v), rol \in \{src, san, snk\} \tag{3.1}$$

No todas nuestras variables son incógnitas del sistema, ya que una porción de los términos del programa, $\mathcal{A}_M$, está previamente anotada. Por eso, se agregan restricciones que asignan un puntaje con valor 1 a los términos cuyo rol es conocido. Si consideramos a los elementos del conjunto de términos anotados $\mathcal{A}_M$ como tuplas $\langle t, r \rangle$, con t un término, y r un rol cuyo valor puede ser source, sink o sanitizer, podemos definir las restricciones sobre los términos cuyo rol es conocido de la siguiente manera:

$$\begin{cases} (n)^r = 1 \\ (n)^t = 0 \quad \forall t \in \{src, san, snk\} \setminus \{r\} \end{cases} \quad \forall \langle v, r \rangle \in \mathcal{A}_M \mid n = Rep(v) \tag{3.2}$$

3.3.3. Formalización de las restricciones

Habiendo definido las variables del modelo, nos hace falta formalizar las restricciones descriptas en la sección 3.3.1. Tomemos la restricción presentada en la figura 3.1a. Recapitulando, esta restricción nos dice que si un vértice v_{san} cuyo rol podría ser sanitizer, tiene flujo de información hacia un potencial sink v_{snk}, entonces alguno de los vértices anteriores a v_{san} debería tener un rol de source. Podemos primero formalizar esta relación en lógica de primer orden[1]:

$$\begin{aligned} \textsc{Sanitizer}(v_{san}) \land \textsc{Sink}(v_{snk}) \land v_{san} \rightarrow^* v_{snk} \Rightarrow \\ \exists v \in V \mid v \rightarrow^* v_{san} \land \textsc{Source}(v) \end{aligned} \tag{3.3}$$

Para introducir la forma en que Seldon expresa esta restricción en su formulación lineal, podemos tomar en cuenta el ejemplo presentado en la figura 3.2. La figura muestra un pequeño propagation graph que ilustra un caso de la intuición antes discutida. El vértice v_{san}, cuya representación programa independiente es $n_{san} = Rep(v_{san})$, sería asignado el rol de sanitizer. Luego v_{sink}, con sink como rol asignado, tiene una representación n_{sink}. De manera análoga, los vértices v_1, v_2, v_3 tienen representaciones n_1, n_2, n_3, y serían los potenciales sources según lo expresado en la intuición. Por último, el vértice no etiquetado es un término que sabemos no puede actuar como source, por lo que procedemos a ignorarlo. Seldon expresa esta restricción explicando que la suma de los puntajes de todos los potenciales sources deberá ser mayor o igual a la suma del puntaje de v_{san} y v_{snk}. De esta forma, la formulación lineal para el ejemplo será:

[1] La operación $\rightarrow^*$ es la clausura transitiva de la relación de adyacentes en un grafo, y SOURCE, SANITIZER y SINK predican sobre el rol asignado a un vértice.

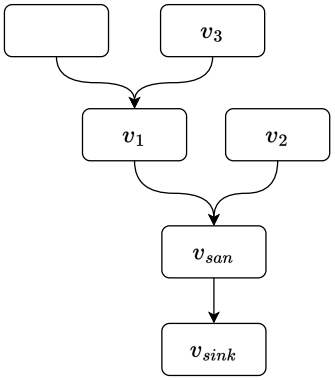

Fig. 3.2: Ejemplo de propagation graph usado para ilustrar la restricción construida a partir de la figura 3.1a.

$$(n_{san})^{san} + (n_{sink})^{snk} \leq (n_1)^{src} + (n_2)^{src} + (n_3)^{src} + C \tag{3.4}$$

Donde C es una constante fija en todo el sistema. Si fuera el caso que $C = 1$ y restringiéramos el valor de todas las variables $n_\star$ a $\{0, 1\}$, luego la ecuación 3.4 captura la intuición presentada en la figura 3.1a. Primero, si $(n_{san})^{san} = (n_{sink})^{snk} = 1$, caso en que v_{san} y v_{sink} son asignados los roles de sanitizer y sink respectivamente, por lo menos una de las variables entre $(n_2)^{src}, (n_2)^{src}, (n_3)^{src}$ debe ser 1 para satisfacer la ecuación. De esta forma, se expresaría el caso en que alguno de los vértices antecesores debe ser un source. Por otro lado, la ecuación se satisface trivialmente si $(n_{san})^{san}$ o $(n_{sink})^{snk}$ valen 0.

Si generalizamos la ecuación antes presentada, la restricción puede ser expresada mediante la siguiente inecuación:

$$(n_{san})^{san} + (n_{snk})^{snk} \leq \sum_{i=1}^{k} n_i^{src} + C \tag{3.5}$$

La constante C, configurada con un valor fijo para todo el sistema, fue experimentalmente definida con el valor 0,75 en [3]. Las variables n_{san}, n_{snk} y n_i son las representaciones[2] obtenidas mediante la función *Rep* de v_{san}, v_{snk} y los potenciales sources v_i respectivamente.

De manera similar, si tomamos la restricción planteada en la figura 3.1b, buscaremos que la suma de las puntuaciones de un source v_{src} y un sanitizer v_{san}, que cumplen $v_{src} \rightarrow^* v_{san}$, deberá ser menor o igual a la suma los puntajes de los potenciales sinks $v_1, ..., v_k$ donde $v_{san} \rightarrow^* v_i, i \in \{1, ..., k\}$. Esto puede ser formalizado con la siguiente fórmula:

$$(n_{src})^{src} + (n_{san})^{san} \leq \sum_{i=1}^{k} n_i^{snk} + C \tag{3.6}$$

Por último, la restricción correspondiente a la figura 3.1c se expresa siguiendo una idea similar:

[2] Como fue explicado en las sección 3.2, un término t puede tener múltiples representaciones con especificidad variable. En Seldon, para un término v se agregan en cada restricción de flujo un subconjunto de las representaciones, elegido teniendo en cuenta la cantidad de ocurrencias de cada una en el programa al que pertenecen. Luego, para reducir estas a una única variable para cada término t, se toma la media de todas sus correspondientes. Se hace caso omiso de estos detalles en la explicación ya esta estrategia, llamada *backoff variables* en la publicación original, no es usada en este trabajo.

$$(n_{source})^{src} + (n_{sink})^{snk} \leq \sum_{i=1}^{k} n_i^{san} + C \tag{3.7}$$

3.4. ¿Cómo se resuelve el modelo?

Podemos clasificar las restricciones antes definidas en 3 conjuntos, lo cuál nos permitirá hacer referencia a ellas más fácilmente:

1. C^{flow}: Restricciones inspiradas en 3.3.1: 3.5, 3.6 y 3.7.
2. C^{var}: Restricciones sobre qué valores pueden tomar las variables del sistema: 3.1.
3. C^{known}: Restricciones creadas a partir de términos de programa ya anotados $\mathcal{A}_M$: 3.2.

Debemos tener en cuenta que existen casos donde un programa en $\mathcal{D}$ no necesariamente cumple las restricciones relatadas en 3.3.1, lo cual volvería nuestro sistema de restricciones insatisfacible en muchos casos. Por ello, debemos relajar las restricciones impuestas en C^{flow}, de forma que incluyan posibles errores. En un sistema lineal, relajar una restricción consiste en incorporar a la misma una variable, parte de la función objetivo, que captura el error en la restricción. Por último, en cuanto a C^{var} y C^{known} estas seguirán siendo restricciones *duras*, es decir, no relajadas.

Relajaremos cada restricción c en C^{flow}, tomando como ejemplo la restricción descripta en 3.5, y asumiendo que c es la i-ésima restricción en C^{flow}:

$$(n_{san})^{san} + (n_{snk})^{snk} \leq \sum_{j=1}^{k} n_j^{src} + C$$

Agregamos a la restricción original un término de error $\epsilon_i \geq 0$, el cuál es único para la i-ésima restricción

$$(n_{san})^{san} + (n_{snk})^{snk} \leq \sum_{j=1}^{k} n_j^{src} + C + \epsilon_i$$

$$(n_{san})^{san} + (n_{snk})^{snk} - \sum_{j=1}^{k} n_j^{src} - C \leq \epsilon_i \tag{3.8}$$

Luego, con las restricciones en C^{flow} escritas como $L_i \leq \epsilon_i$ (con L_i el lado izquierdo de la i-ésima restricción en 3.8, y ϵ_i su correspondiente término de error), y con el objetivo de minimizar el error ϵ_i con respecto a las restricciones de flujo originales (calculado como $\sum_i \epsilon_i$), **podemos expresar el problema de optimización de la siguiente manera**:

$$min\left(\sum_{i=1}^{M} \epsilon_i + \lambda \sum_{r \in \mathcal{R}} ((n_r)^{src} + (n_r)^{san} + (n_r)^{snk})\right) \tag{3.9}$$

$$\text{sujeto a } 0 \leq (n_r)^{rol} \leq 1 \; \forall n_r \text{ y} \tag{3.10}$$

$$0 \leq \epsilon_i \; \forall i \in \{1, ..., M\} \text{ y} \tag{3.11}$$

$$\begin{cases} (n)^r = 1 \\ (n)^t = 0 \quad \forall t \in \{src, san, snk\} \setminus \{r\} \end{cases} \quad \forall \langle v, r \rangle \in \mathcal{A}_M \mid n = Rep(v) \tag{3.12}$$

$$\text{y cada una de las } M \text{ restricciones } c_i \text{ en } C^{flow} \tag{3.13}$$

donde M es la cantidad de restricciones en C^{flow}, $\mathcal{R}$ es el conjunto de todas las representaciones posibles usadas en el sistema (o el conjunto de variables, espacio que también hacemos referencia con $\forall n_r$), y λ un parámetro de regularización que penaliza si una gran cantidad de términos son asignados puntuaciones altas. Esto último, expresado en otras palabras, quiere decir que penalizamos posibles soluciones donde muchos términos tienen un rol asignado, con un nivel alto de confianza.

De esta forma, el problema queda definido por la función objetivo, descripta en 3.9; sujeta a las restricciones duras que surgen de la definición de las variables del sistema (C^{var} y C^{known}), 3.10 y 3.12 respectivamente; y la versión relajada de las restricciones de flujo en C^{flow} junto con las restricciones sobre sus términos de error, 3.13 y 3.11.

3.5. Solver

La publicación original no provee muchos detalles acerca de la metodología utilizada para optimizar el modelo, más allá de cuál es el solver utilizado y el algoritmo subyacente. Seldon utiliza un solver que es parte de la biblioteca de deep learning TensorFlow[3] llamado Adam Optimizer[4]. Este utiliza un método de optimización conocido como *projected gradient descent*, aplicado al algoritmo de *stochastic gradient descent* Adam [10].

3.6. Aplicar lo aprendido

Una vez terminado el proceso de inferencia, Seldon utiliza las especificaciones inferidas en un motor de taint analysis construido a partir del propagation graph definido en 3.1.

Coloquialmente, encontrar alertas que serían reportadas por un análisis de propagación de taint, consiste en buscar dos términos, un source y un sink, y encontrar un camino entre ellos donde no haya un sanitizer en el medio. La primera puede ser resuelta tomando el conjunto $\mathcal{A}_M$, y los roles inferidos $\mathcal{A}_I$, los cuales nos permiten encontrar sources, sinks y sanitizers dentro de un programa. Luego, dado un source v_{src} y un sink v_{snk} solo resta por encontrar, utilizando el grafo G, si existe un camino entre ambos que no contenga un vértice v_{san} el cual sea un sanitizer.

[3] Tensorflow.
[4] Adam Optimizer.

4. JELDON: IMPLEMENTACIÓN DE SELDON PARA JAVASCRIPT

Tal como fue mencionado, el aporte principal de este trabajo es una implementación de la técnica desarrollada en Seldon [3], aplicada para el lenguaje JavaScript, a la cual nos referiremos de ahora en más como JELDON [1]. Esta, parte de la idea principal de Seldon, que es formular el problema de inferencia de especificaciones de taint analysis como un modelo de optimización lineal (también nos referiremos a este tipo de modelos como de programación lineal, o LP). Esta idea permite escalar el problema de inferencia a grandes conjuntos de programas, de manera semi-supervisada, y partiendo de un conjunto de términos anotados $\mathcal{A}_M$ relativamente pequeño.

En Seldon se utiliza como conjunto de datos anotados, un conjunto de términos de programa $\mathcal{A}_M$ anotados con su correspondiente rol de taint analysis. A su vez, el resultado del proceso de inferencia utiliza la misma representación. Esto último es así ya que el motor de análisis utilizado en la publicación funciona a partir del mismo propagation graph construido para el proceso de inferencia. Teniendo las especificaciones iniciales e inferidas, $\mathcal{A}_M$ y $\mathcal{A}_I$ respectivamente, se puede saber si un vértice del grafo v tiene un rol r, mediante $v^r \in \mathcal{A}_M \cup \mathcal{A}_I$.

En nuestra técnica, el motor de análisis utilizado, junto con las especificaciones iniciales e inferidas, es CodeQL. Este es impleado también para la construcción del propagation graph. Debido a esto, nuestras especificaciones iniciales no serán un conjunto de términos anotados como en Seldon, sino un conjunto de consultas de CodeQL, al que llamaremos $\mathcal{Q}_M$. Una vez terminado el proceso de inferencia, obtendremos un conjunto de términos anotados $\mathcal{A}_I$, que para ser utilizados en un análisis deberán ser traducidos a un nuevo conjunto de consultas, $\mathcal{Q}_I$. La utilización de conjuntos de consultas como forma de denotar especificaciones de taint análisis, se debe principalmente a dos motivos: como fue mencionado en 2.3, CodeQL es una herramienta de análisis de código muy usada en la industria, ya que es desarrollada y aplicada en la principal plataforma de código versionado, Github. Por otro lado, además del motor de análisis de código, CodeQL es una biblioteca de consultas[2] que permiten identificar todo tipo de vulnerabilidades, desarrollada sobre su propio lenguaje de consultas QL. Esto nos permite utilizar las mismas como especificaciones iniciales, así como también como oráculo en la metodología de evaluación. Más aún, la expresividad del lenguaje proveída por la herramienta nos permite extenderla, pudiendo así agregar las funcionalidades necesarias para que la misma interactúe con el resto del proceso de inferencia.

En nuestro caso en particular, las consultas utilizadas como especificaciones iniciales siempre estarán en el contexto de una vulnerabilidad en particular. Para dar un ejemplo, la siguiente consulta NosqlInjectionQuery.qll describe, primero indicándole al motor que el contexto elegido es de taint analysis, qué forma tienen los sources, sinks y sanitizers para vulnerabilidades del tipo *NoSQL injection*. Este tipo de vulnerabilidades ya vistas en el ejemplo de código 2, consiste en que, un dato inseguro ingresado por un usuario, puede alcanzar un punto en el programa, sin haber sido sanitizado, donde se realiza una consulta a una base de datos NoSQL.

[1] **J**avaScript S**eldon**

[2] El motor de análisis estático de código, junto con la biblioteca de consultas, es conocido como CodeQL, el proyecto. Las bibliotecas de consultas están disponibles en Github.

4.1. Arquitectura

La técnica implementada en esta tesis parte de un conjunto de programas en JavaScript $\mathcal{D}$, y un conjunto de consultas $\mathcal{Q}_M$ que modelan taint analysis para una vulnerabilidad en particular. Como en Seldon, las etapas principales consisten en construir un propagation graph 3.1.1, generar un modelo de optimización lineal a partir de éste, que contenga las intuiciones presentadas en 3.3.1, y optimizar el modelo que resuelve el problema de inferencia.

En Seldon, estas tres etapas construyen un solo modelo $\mathcal{M}$ que, al ser optimizado, genera un único conjunto de términos anotados $\mathcal{A}_I$, las especificaciones inferidas. Una de las principales diferencias de nuestra técnica es que Jeldon trata el problema de inferencia individualmente para cada programa $\rho \in \mathcal{D}$, construyendo un grafo de menor tamaño G_ρ, traduciéndolo a un modelo $\mathcal{M}_\rho$, optimizándolo y así obteniendo un conjunto resultado $\mathcal{A}_I^\rho$. Luego, los resultados parciales son combinados para así obtener un único conjunto resultado $\mathcal{A}_I$.

Por otro lado, nuestra técnica utiliza una representación programa independiente diferente. El caso está pensado para representar especialmente, con mucha claridad, términos donde el programa bajo estudio interactúa con bibliotecas de código externo. Además, posee un elemento que le permite reducir la especificidad de una expresión, haciendo que la misma pueda abarcar más términos de programa. Esto será explicado más en detalle en la sección 4.2.

El siguiente diagrama muestra las diferentes etapas de la técnica y cómo interactúan entre sí, sus entradas y salidas, resultados parciales y finales.

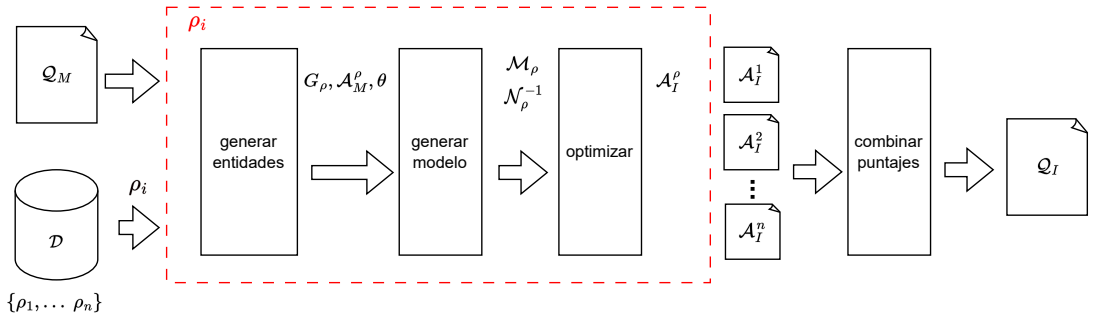

Fig. 4.1: Diagrama de arquitectura de Jeldon .

Cada etapa del proceso de inferencia consume un resultado parcial de la etapa anterior, y produce otro que será consumido por la etapa siguiente. Como fue mencionado antes, Jeldon realiza el proceso de inferencia de manera individual sobre cada programa en el conjunto de entrada. El rectángulo rojo muestra las etapas que procesan un programa ρ en particular. Estas producen como resultado parcial un conjunto de términos anotados $\mathcal{A}_I^\rho$. La última etapa **combinar puntajes** toma todos los resultados parciales y los combina aplicando un procedimiento que definiremos más adelante, obteniendo finalmente un nuevo conjunto de consultas $\mathcal{Q}_I$.

En este capítulo, el primer apartado explica en detalle la representación usada para los términos, de manera de independizarlos del programa al que pertenecen. Luego, se explicará en qué consiste cada etapa de la técnica, así como sus diferencias con sus correspondientes en Seldon. Para poder enfocarnos en una etapa en particular, definimos la

interfaz entre cada etapa, ya que cada una de estas debe recibir una o más entradas y comunicar un resultado intermedio a la etapa siguiente.

1. GENERAR ENTIDADES: Esta etapa parte del conjunto de consultas inicial $\mathcal{Q}_M$ y un programa $\rho \in \mathcal{D}$. La misma produce tres resultados: el propagation graph G_ρ, un conjunto de términos $\mathcal{A}^\rho_M$ de términos anotados, y una función θ que nos permite convertir un término de programa t a su representación programa independiente $Repr(t)$.

2. GENERAR MODELO: Construye el modelo de programación lineal en el formato esperado por el solver, al que llamaremos $\mathcal{M}_\rho$. Además, como esta etapa realiza la asignación de términos en su representación independiente a variables, construye la inversa de esta asignación $\mathcal{N}^{-1}_\rho$.

3. OPTIMIZAR: La etapa consiste de dos partes, una primera que toma el modelo construido en la etapa anterior $\mathcal{M}_\rho$ y lo optimiza, y otra que traduce el resultado del solver a un conjunto de términos anotados $\mathcal{A}^\rho_I$. Este último es el resultado final de las primeras tres etapas.

4. COMBINAR PUNTAJES: Última etapa y única que "tiene en cuenta" el conjunto entero de programas $\mathcal{D}$. Esta recibe como entrada cada uno de los conjuntos de términos anotados inferidos $\mathcal{A}^1_I, ..., \mathcal{A}^n_I$. La etapa los reduce a un solo conjunto de consultas $\mathcal{Q}_I$, el cual será el resultado final del proceso de inferencia.

Por último, se explicará cómo, habiendo obtenido el conjunto de consultas inferidas $\mathcal{Q}_I$, se puede utilizar junto con el inicial para mejorar taint analysis.

4.2. Representación orientada a código externo

Como fue explicado en 2.1, los programas implementados en JavaScript tienden a ser altamente modulares. Consecuentemente, un gran número de las llamadas a función suele ser a bibliotecas externas. Como la herramienta que utilizamos para construir el grafo de propagación es inter-procedural, este puede capturar el flujo de una punta del programa a la otra, registrando correctamente llamadas a función siempre y cuando la implementación sea local. En cambio, flujos que cruzan la frontera del programa con sus dependencias no son capturados. Por ejemplo, una llamada a una función que forma parte de una biblioteca externa no es tenida en cuenta por la herramienta, o es sobreaproximada. Por esto, los términos del programa donde un flujo de ejecución cruza la frontera entre el código fuente del programa y sus dependencias, resultan un objetivo interesante para que nuestra representación caracterice de manera eficiente. De otra manera, estos suelen ser asignados una sobreaproximación de su rol a nivel de propagación de taint, por ejemplo, que siempre lo propagan.

En TASER [17] se introduce una representación programa independiente 3.2.1 llamada *access paths*. La misma resulta muy útil para caracterizar lo que en la publicación se denomina *puntos de contacto*, es decir, puntos de entrada y salida a una biblioteca externa en el programa estudiado.

Definición 4.2.1 (Punto de contacto)**.** Un punto de contacto es un término de un programa donde se produce una interacción con una biblioteca de código externo. Por ejemplo,

estos pueden ser una llamada a una función que está definida en una biblioteca importada por el programa (**punto de salida**). También pueden ser instanciaciones de un tipo de datos importado, o *callbacks*[3], mediante las cuales, valores producidos por código externo al programa ingresan a él (**puntos de entrada**).

En el snippet 4.2 podemos ver dos ejemplos de puntos de contacto. En primer lugar, en la línea 6, podemos ver que dentro de la llamada a función `app.get(path, handler)`, el argumento `handler` es una función por medio de la cual, una biblioteca externa utilizada para crear una aplicación web, puede enviarnos los pedidos recibidos. Esto es lo que llamamos un **punto de entrada**. Luego, en la línea 7, se puede observar el primer y único argumento de la llamada a la función `findOne`, que proviene de una biblioteca para realizar consultas a una base de datos NoSQL. Esto es lo que llamamos un **punto de salida**.

```
1 const mongoose = require("mongoose");
2 const app = require("express");
3
4 let db = mongoose.connect(...);
5
6 app.get('/search', async (req, res) => {
7   let data = await db.collection('movies').findOne(req.query.query)
8   // ...
9 });
```

Fig. 4.2: Versión simplificada de 2 para ilustrar diferentes *access paths*.

Esta representación permite caracterizar puntos de contacto usando información acerca de su contexto, de manera que sean transportables a otros programas que consumen las mismas bibliotecas. Veamos cómo se pueden expresar los dos ejemplos anteriores como access paths. Primero, podemos utilizar el caso del punto de entrada para explicar cómo se lee una expresión de access paths:

```
(parameter 0 (parameter 1 (member get (root express))))
           4           3           2           1
```

Los access paths son un conjunto de expresiones anidadas, leídas de adentro, hacia afuera. El caso anterior se interpreta de la siguiente manera:

1. El objeto que surge de *importar* la biblioteca "express", `require("express")`.
2. El atributo `get` de 1.
3. El segundo parámetro[4] de 2, que resulta ser una función.

[3] Las *callbacks* son funciones usadas como argumento en una llamada a función. Mediante estas, la función llamada puede comunicar el resultado al context llamador. Son un patrón muy utilizado en JavaScript por dos motivos: primero, como el runtime funciona con un modelo de ejecución de un solo hilo, muchas operaciones que requieren una cantidad considerable de tiempo de cómputo o entrada/salida, utilizan callbacks para comunicar el resultado al contexto que las invocan; y por otro lado, debido al soporte de funciones de alto orden. Este patrón resulta muy conocido debido a su uso en exceso, ocasión que dio origen al concepto *callback hell*, donde existen decenas de callbacks anidadas.

[4] El índice de los parámetros de función comienza en 0.

4. El primer parámetro de 3, que resulta ser una función.

Luego, considerando el ejemplo del punto de salida, la representación tiene mayor longitud debido a que más información de contexto es necesaria para caracterizar este término.

```
(parameter 0 (member findOne (return (member collection (return
↪  (member connect (root mongoose)))))))
```

Fig. 4.3: Access path de un punto de salida con demasiado detalle.

Esta representación captura cómo está conformado el término, y de dónde proviene cada elemento que forma parte del mismo. Como se busca lograr una caracterización completa, la representación describe cada detalle, aunque esto podría no ser lo deseado. Este es un caso donde la representación puede resultar **demasiado específica**. Como nuestro objetivo final es poder inferir nuevos sinks, sources o sanitizers, queremos que las representaciones utilizadas generalicen apropiadamente un elemento particular de un programa.

Para ello, nuestra técnica utiliza una representación basada en access paths, con el agregado de un nuevo término sintáctico para poder generalizar, el símbolo $*$. Este puede ser utilizado reemplazando cualquier término interior de una representación, actuando como un *comodín*. De esta forma, si se quieren omitir detalles como en el ejemplo anterior, de dónde proviene el objeto que contiene la función `findOne`, se puede reemplazar todo el término por $*$. Realizado el reemplazo, esta representación no especifica de dónde proviene `findOne`, ya que solo predica sobre el argumento de la misma y su nombre. Al mismo tiempo, esto nos permite variar fácilmente el nivel de especificidad de una representación particular. Los siguientes dos ejemplos parten de la representación 4.3, y reemplazan dos sub-términos diferentes por $*$, de forma que, interpretados semánticamente, quiere decir que un resultado restringe mucho más las ocurrencias de términos de programa que podrían ser asociados, en comparación con la otra.

1. `(parameter 0 (member findOne (return (member collection *)))`: Se reemplazan los términos sintácticos que indican que la función `collection` proviene de una llamada a una función de conexión a una base de datos; parte de la biblioteca *mongoose*. De esta manera, la representación podría cubrir casos donde la conexión se encuentra dentro de una variable, el atributo de un objeto, o está abstraída dentro de otra función.

2. `(parameter 0 (member findOne *)`: Este segundo ejemplo reemplaza casi la totalidad del término, solo predicando sobre el primer argumento de la llamada a una función *findOne*. Si el nombre de la función es lo suficientemente específico, los términos capturados por esta representación deberían ser los pretendidos.

En esta sección extendimos una representación existente, especialmente útil para caracterizar puntos dentro de un programa donde se cruza la frontera de lo definido localmente con bibliotecas externas, agregando la capacidad que permite reducir una representación dada a otra menos específica. La siguiente sección presenta formalmente cómo se construyen estas expresiones, generadas por una función a la que llamaremos *Repr*.

4.2.1. Definición formal

Usando la extensión de access paths presentada en la sección anterior, la gramática 4.1 formaliza cómo están construidas las expresiones de esta representación. En las siguientes secciones, haremos uso de una función que nos permite traducir un término de programa t, a su representación programa independiente r, llamada *Repr*. La representación generada tiene las siguientes características:

- Basada en access points.
- Se agrega el elemento sintáctico $*$, usado para generalizar una representación más específica, a una menos.
- Se agrega el término *global*, usado para referirse variables globales como `document` o `window`, cruciales en aplicaciones desarrolladas para navegadores web.

$\mathcal{E}$	::=	$(\mathcal{A})$	
$\mathcal{A}$	::=	*global*	*Objeto global*
	|	*root M*	*`import` M, con M biblioteca de JavaScript*
	|	*parameter D* $\mathcal{R}$	*Argumento D de la función* $\mathcal{R}$
	|	*member N* $\mathcal{R}$	*Acceso al atributo N de* $\mathcal{R}$
	|	*instance* $\mathcal{R}$	*Llamada a `new` de* $\mathcal{R}$
	|	*return* $\mathcal{R}$	*`return` en el cuerpo de una función*
$\mathcal{R}$	::=	$*$	*Comodín*
	|	$\mathcal{E}$	*Caso recursivo*

Gramática 4.1: Definición formal de cómo está construida una representación generada por *Repr*.

4.3. Construcción del propagation graph

Como fue mencionado en la introducción de este capítulo, la construcción del propagation graph fue implementada utilizando CodeQL como herramienta que nos permite extraer información de un programa ρ. En 2.3 se menciona que el lenguaje de consultas QL nos permite predicar en diferentes niveles de abstracción, como data flow y control flow. También mencionamos que QL sigue la semántica declarativa de Datalog, por lo que una consulta en este lenguaje es equivalente a un predicado lógico. Esto da lugar a que en las siguientes secciones se puedan expresar las consultas implementadas para extraer el grafo, el conjunto de términos cuyo rol es conocido, y θ como si las mismas fueran relaciones y consultas de Datalog. Seguiremos este enfoque a pesar de la declaratividad que presenta el lenguaje QL, para permitir que, un lector no familiarizado con él, pueda entender las ideas detrás de la técnica. Por el mismo motivo, la traducción de las consultas omitirá detalles específicos de QL, como tipos de datos, nombres de funciones utilizadas, entre otros, para así centrarse en las relaciones expresadas por las mismas.

En la mayor parte de los casos de la implementación, cuando predicamos sobre términos de programa, lo haremos sobre un nodo dentro del grafo de data flow, representado en CodeQL por el tipo de datos `DataFlow::Node`. Más allá de situarnos dentro del grafo de data flow, el lenguaje nos permite pasar de un nivel de abstracción a otro como lo muestra la figura 4.4. Es necesario notar que, cuando ocurre un cambio de un nivel de mayor

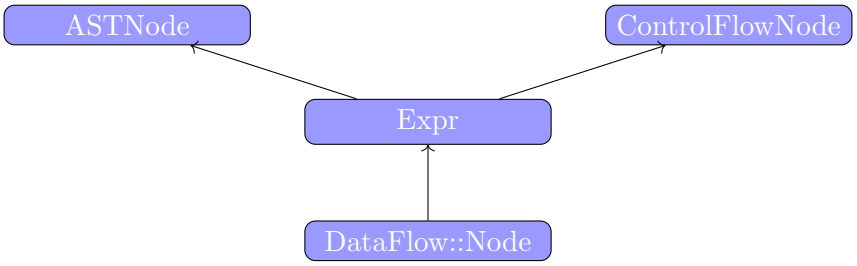

Fig. 4.4: Diagrama de clases mostrando como están relacionados los tipos de datos a los que podríamos referirnos por **términos** de programa. Podemos ver que hay una correspondencia entre `DataFlow::Node` y `Expr`, pero un nodo de tipo `Expr` puede significar tanto un elemento del grafo de control flow (`ControlFlowNode`) o del árbol de parseo (`ASTNode`).

(data flow por ejemplo) a menor (control flow) abstracción, la/s variable/s afectada/s puede/n sufrir cambios de especificidad en cuanto a qué representan, ya que pueden existir elementos que no son propios en el nivel de menor abstracción.

Como fue explicado en la definición de propagation graph 3.1.1, este es un grafo donde el conjunto de vértices V es un conjunto de términos de programa que consideramos *relevantes*, y el conjunto E de ejes modela el flujo de información dentro del programa. Explicaremos primero qué términos consideramos como relevantes, qué tipo de relaciones entre términos consideramos como flujo de información, cómo obtener el conjunto de términos conocidos y, por último, hablaremos de cómo construir θ.

4.3.1. Datalog como lenguaje de consulta

Datalog es un lenguaje de programación lógico y declarativo usado principalmente como un lenguaje de consultas sobre bases de datos deductivas, en el cual está inspirado QL. En este trabajo utilizaremos Datalog para ilustrar las consultas implementadas en QL de forma que un lector ajeno a esta tecnología pueda entender la técnica. Siempre que hablemos de una consulta o un conjunto de ellas en Datalog, tendremos en cuenta que existe una base de datos $\mathcal{B}$ sobre la cual será realizada la consulta. Esta sección introduce Datalog, haciendo hincapié en las características del lenguaje que son relevantes para el uso que se le dará en esta tesis, como medio didáctico.

Un programa en Datalog consiste de una serie de *hechos*, los cuales son considerados verdaderos y forman la base de datos $\mathcal{B}$, y una serie de reglas mediante las cuales se pueden deducir nuevos hechos. Los primeros se expresan como $R(\vec{x})$, donde $\vec{x}$ es un vector de variables.

Las reglas son expresiones con la siguiente forma

$$R(\vec{x}) \leftarrow R_1(\vec{x_1}), ..., R_n(\vec{x_n}) \tag{4.1}$$

Donde cada expresión de la forma $R_i(\vec{x_i})$ es llamada átomo. Esto hace que un hecho sea un átomo, el lado izquierdo de las reglas llamado *cabeza* también, y el lado derecho o *cuerpo* de la regla, un conjunto de átomos.

Podemos interpretar una regla como si la misma fuera un predicado en lógica de primer orden (LPO). En el siguiente ejemplo, ambos predicados, uno en Datalog y otro en lógica de primer orden tienen el mismo significado.

$$\begin{aligned} \rho &= \mathrm{T}(x, y) \leftarrow \mathrm{T}(x, z),\ \mathrm{R}(z, y). \\ \phi_\rho &= \forall x, y, z(\mathrm{T}(x, z) \wedge \mathrm{R}(z, y) \rightarrow \mathrm{T}(x, y)) \end{aligned} \tag{4.2}$$

Otra característica, es que pueden existir múltiples reglas con el mismo átomo C en la cabeza. Lógicamente, múltiples reglas con el mismo átomo en la cabeza son tratadas como una disyunción entre estas, ya que si la conjunción de los elementos del cuerpo de alguna de todas es verdadera, luego el átomo de la cabeza también. En función de ser sintéticos, introduciremos en la notación de Datalog utilizada disyunciones en el cuerpo de una regla. Estas se expresan de la siguiente manera:

$$R(\vec{x}) \leftarrow R_1(\vec{x_1}); ...; R_n(\vec{x_n}) \tag{4.3}$$

Y es equivalente en LPO a

$$\forall x, y, z(R_1(\vec{x_1}) \vee ... \vee R_n(\vec{x_n}) \rightarrow R(\vec{x})) \tag{4.4}$$

También utilizaremos una mezcla de conjunciones y disyunciones en el cuerpo de diversas reglas, por el mismo motivo de síntesis. El siguiente ejemplo ilustra una regla donde el cuerpo tiene un mezcla de conjunciones y disyunciones entre átomos, y su equivalente en LPO:

$$\begin{aligned} &\textsc{Relevante}(x) \leftarrow \mathrm{A}(x),\ (\mathrm{B}(x);\ \mathrm{C}(x)). \\ &\forall x(\mathrm{A}(x) \wedge (\mathrm{B}(x) \vee \mathrm{C}(x)) \rightarrow \textsc{Relevante}(x)) \end{aligned} \tag{4.5}$$

4.3.2. Términos relevantes

Seldon define como términos relevantes a los siguientes tipos:

- Llamadas a función
- Lectura de atributos de un objeto
- Argumentos en la definición de una función

Esta definición acota el universo de términos de un programa, pero puede llegar a ser demasiado abarcativa, lo que llevaría a tener mayor cantidad de vértices en el grafo resultante, y por consiguiente, un modelo más grande. Como QL nos permite predicar sobre un programa en diferentes niveles de abstracción, e incluye primitivas en el lenguaje para poder realizar consultas sobre construcciones de más alto nivel fácilmente (imports, o funciones de alto orden, por ejemplo), nuestra definición de término relevante intenta, desde un principio, acotar el conjunto de vértices del grafo a aquellos que podrían ser potenciales sources, sanitizers o sinks. Más aún, como fue mencionado en la introducción de este capítulo, Jeldon busca en especial realizar inferencias sobre términos que se encuentran en una interacción entre el programa bajo estudio y bibliotecas externas, los llamados puntos de contacto 4.2.1. Por ello, nuestra definición de término relevante tiene en cuenta estas ideas para así obtener un grafo cuyos vértices son candidatos ser puntos de contacto, y tener un rol de taint analysis.

Definiremos que un término es relevante en nuestra técnica mediante la siguiente regla:

$$\text{Relevante}(t) \leftarrow \text{CandidatoASource}(t);\ \text{CandidatoASanitizer}(t);\ \text{CandidatoASink}(t). \tag{4.6}$$

Para definir las reglas de cada una de las relaciones CandidatoA*, hace falta caracterizar qué propiedades tiene que cumplir un término para ser considerado como punto de contacto. Para ello, vamos a definir dos reglas que determinan si un término t es un punto de entrada (PuntoDeEntrada) o salida (PuntoDeSalida).

$$\begin{aligned}
&\text{PuntoDeEntrada}(t) \leftarrow \text{EsImport}(t).\\
&\text{PuntoDeEntrada}(t) \leftarrow\\
&\quad \text{PuntoDeEntrada}(b),\ \text{Relacionados}(b,t).\\
\star\ &\text{PuntoDeEntrada}(t) \leftarrow\\
&\quad \text{PuntoDeSalida}(b),\ \text{LlamadaAFunción}(b),\ \text{EsArgumentoOBase}(b,t).\\
&\text{PuntoDeSalida}(t) \leftarrow\\
&\quad \text{PuntoDeSalida}(b),\ \text{Relacionados}(b,t).\\
\star\star\ &\text{PuntoDeSalida}(t) \leftarrow\\
&\quad \text{PuntoDeEntrada}(b),\ \text{LlamadaAFunción}(b),\ \text{EsArgumentoOBase}(b,t).
\end{aligned} \tag{4.7}$$

Tanto la definición de PuntoDeEntrada como PuntoDeSalida son bastante declarativas. La primera presenta, en primer término, un caso base que indica que, cualquier término que incluye código externo al programa es un punto de entrada. Luego, ambas relaciones definen recursivamente que si otro término b es un punto de entrada o salida, y tanto t y b están "relacionados", luego t también lo es. Relacionados(b,t) es cierto si alguna de las siguientes condiciones se cumple entre ambos términos:

1. t es un atributo de b
2. t es una instanciación de b, mediante el operador `new`
3. b es una llamada a función, y t es el valor resultado
4. b es una promesa[5], y t es el valor al que b eventualmente resuelve

Son particularmente interesantes la tercera regla de PuntoDeEntrada ($\star$) y la segunda de PuntoDeSalida ($\star\star$). Podemos entender cómo ambas colaboran entre sí para caracterizar casos complejos como es el uso de *callbacks*. Una *callback* es una función que es usada como parámetro en una llamada a función, para que la función invocada comunique algún evento al contexto llamador. Consideremos el siguiente snippet de código que muestra el uso de una biblioteca para interactuar con MySQL, un motor de bases de datos muy conocido. La misma provee un método `execute` que acepta como primer parámetro una consulta, y como segundo, una función. Este segundo parámetro cumple el rol de callback, por medio del cual se comunica el resultado en caso en que la consulta sea exitosa, o información de error en caso contrario.

[5] Tipo de datos `Promise` definido dentro del modelo de computo asíncrono en JavaScript. Para más detalles ver Promise API.

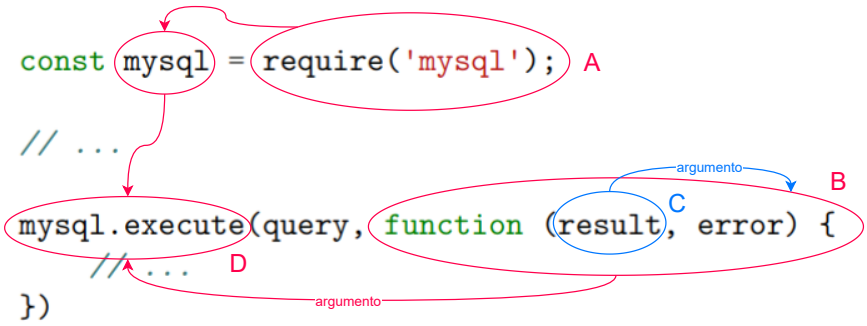

Fig. 4.5: Snippet de código JavaScript que muestra como interactúan diferentes puntos de contacto al usar callbacks.

Si seguimos los términos anotados como A, B y C en la figura 4.5, y las reglas definidas en 4.7, podemos deducir que una callback cumple un rol de punto de entrada:

1. EsImport(A), que implica PuntoDeEntrada(A).
2. Como A es un punto de entrada, y el término D es la lectura del atributo `execute` de A, luego PuntoDeEntrada(D).
3. Como D es un punto de entrada y una llamada a función, y B es parámetro de D, luego PuntoDeSalida(B).
4. Por último, como B es un punto de salida, y una posible llamada a función, y C es un argumento de dicha función, luego vale que PuntoDeEntrada(C).

Por último, las definiciones mencionadas anteriormente, y más adelante, utilizan algunos predicados auxiliares, para caracterizar términos comunes. Las siguientes relaciones predican sobre qué tipo de nodo, dentro del grafo de data flow, es t:

- EsImport(t): t incluye dentro del programa código de una biblioteca externa. Dos ejemplos de esto son `require("express")` or `import * from "express"`.
- EsArgumentoOBase(b, t): Es cierto si t es un argumento de la llamada a función b, o la llamada a función tiene sigue la estructura `t.b(...)` (es decir que t es la base de la llamada).
- LlamadaAFunción(t): t es una llamada a función. Por ejemplo, t podría ser una llamada a una biblioteca externa `mysql.query(...)`, una llamada a una función local al programa `sanitzeRequest(req)` o una llamada a una función anónima `((req) => req.header)(req1)`. Esta regla también cubre casos en los que t no es la invocación de una función, sino su declaración, como en el ejemplo visto sobre *callbacks*.
- EsArgumento(t): t es un argumento dentro de una llamada a función.
- EsReturn(t): t es el valor resultante de una llamada a función, o el valor por ser devuelto dentro de la definición de una función, por ejemplo `return t`.

- LecturaAtributo(t): t es el resultado del acceso al atributo de un objeto, por ejemplo *connection.timeout*.
- EscrituraAtributo(t): t es el valor asignado al atributo de un objeto. Por ejemplo, t podría ser la variable `password` en `user.secret = password`.
- EscrituraAtributo(b, t): t es el valor asignado al atributo de un objeto, con b el lado izquierdo de la asignación.

Habiendo caracterizado cómo se identifican los puntos de contacto y otras relaciones necesarias, podemos definir CandidatoA* a continuación:

$$\begin{aligned}
&\text{CandidatoASource}(t) \leftarrow \text{PuntoDeEntrada}(t),\\
&\quad (\text{LlamadaAFunción}(t); \text{EsArgumento}(t); \text{LecturaAtributo}(t)).\\
&\text{CandidatoASanitizer}(t) \leftarrow \text{LlamadaAFunción}(t).\\
&\text{CandidatoASink}(t) \leftarrow\\
&\quad \text{PuntoDeSalida}(t),\\
&\quad (\text{EsReturn}(t); \text{EsArgumentoOBase}(t); \text{EscrituraAtributo}(t)).
\end{aligned} \tag{4.8}$$

Veamos un poco más en detalle cada una de las caracterizaciones. La definición de **sanitizer** es la menos específica, debido a que es un patrón común que estos se implementen de manera ad-hoc en cada programa, o se utilicen mediante un patrón de diseño que imposibilita encontrar un término de programa donde dicho sanitizer se vea en acción. Un patrón muy común entre estos es *middleware*, donde se agrega funcionalidad de manera ortogonal a las interfases con las cuales el programa interactúa. Si consideramos los primeros casos, sea o no proveniente de una biblioteca externa, las sanitizaciones suelen ocurrir en términos de llamadas a función. Las otras dos definiciones predican en particular sobre puntos de contacto. Los **sources** son términos donde un valor que se mueve por el programa se vuelve de nuestro interés, ya que interactúa o surge de una biblioteca externa, es decir adquiere *taint*. Es por ello que los términos que consideramos como sources son tanto puntos de entrada, así como estructuras donde potencialmente ocurre una asignación sobre una variable, la cual consideraremos como marcada, de ahora en más. Por último, siguiendo la misma idea que los sources, los **sinks** son puntos de salida donde el término indica que el valor de una variable puede tener como destino una pieza de código externa al programa, ya sea en el caso del valor devuelto por una función que es usada como callback, el de un argumento a una función externa, o la escritura de un atributo.

4.3.3. Caracterizando flujo de información

En la sección 2.3 mencionamos que CodeQL permite realizar consultas sobre un programa en diferentes niveles de abstracción, entre ellos data flow. Por este motivo, nuestra técnica aprovecha esta funcionalidad para construir el propagation graph.

Primero caracterizaremos, más en detalle, cómo CodeQL implementa sus funcionalidades a nivel de dataflow. Luego describiremos por medio de una serie de reglas en Datalog, cómo determinar si dos vértices del grafo (o términos) son alcanzables entre sí. En esta parte nos detendremos, ya que, además de utilizar las funcionalidades proveídas por CodeQL, Jeldon tiene en cuenta algunos casos adicionales al momento de contemplar si

hay flujo de información entre dos términos. Por último, haremos una comparación con el propagation graph construido por Seldon.

Cuando hablamos de data flow, generalmente nos referimos a un tipo particular de análisis estático de código que responde a la pregunta de cómo fluyen valores a lo largo del programa, donde los valores son una abstracción descripta en el análisis. En nuestro caso particular, nos interesa si un valor es sensible (*tainted*) o no, variante de dataflow conocida como **taint tracking** o **taint analysis**. Un análisis de dataflow puede ser caracterizado de diversas maneras, siendo una de ellas, si el mismo opera solamente dentro de una función (intra-procedural), o puede manejar llamadas a funciones y responder a cambios en las variables al entrar y salir de estas (inter-procedural).

CodeQL permite consultar si un término t_2 es alcanzable desde otro término t_1 tanto de manera intra como inter-procedural. Nuestra definición de alcanzabilidad entre dos vértices (o términos) del grafo hará fuertemente uso de estas capacidades. Haremos referencia a estos dos tipos de consultas en nuestra abstracción en Datalog por medio de los hechos PasoIntraProcedural(t_1, t_2) y PasoInterProcedural(t_1, t_2) respectivamente. Estos dos hechos responden si desde t_1 se puede alcanzar t_2 en **un solo paso** a nivel de dataflow, ya sea dentro de la misma función o cruzando esta barrera. CodeQL no especifica detalles acerca de las implementaciones de dataflow, más que si las mismas funcionan dentro de una función o través de ellas, y que, en el segundo caso, implementa además un points-to analysis para poder realizar el seguimiento de asignaciones a través de objetos, pudiendo así modelar el heap[6] del programa.

Las reglas en Datalog descriptas a continuación definen cómo consideramos si un término es alcanzable por otro en nuestro propagation graph, mediante la relación Alcanzable(t_1, t_2).

$$
\begin{aligned}
&\textsc{Alcanzable}(t, t).\\
&\textsc{Alcanzable}(desde, hacia) \leftarrow \textsc{Alcanzable}(desde, int),\\
&\quad (\textsc{PasoLocal}(int, hacia);\ \textsc{PasoInterProcedural}(int, hacia)).\\
&\textsc{PasoLocal}(desde, hacia) \leftarrow\\
&\quad \textsc{PasoIntraProcedural}(desde, hacia);\\
&\quad \textsc{EscrituraAtributo}(hacia, desde);\\
&\quad \textsc{EsArgumentoOBase}(hacia, desde);\\
&\quad \textsc{Guarda}(desde, hacia).
\end{aligned}
\tag{4.9}
$$

Como fue mencionado en la introducción de esta sección, además de los pasos de dataflow definidos por CodeQL, nuestra técnica tiene en cuenta algunos casos adicionales, dentro la variante intra-procedural de Alcanzable. Las relaciones que definen los dos primeros casos adicionales ya fueron descriptas en la definición de término relevante. En cambio, la relación de Guarda$(desde, hacia)$ contempla una ocurrencia en particular donde consideraremos que hay flujo de información entre términos. El fragmento de código 4.6 ilustra este caso mediante un ejemplo.

Asumamos que el término de programa representado por *desde* es la llamada a función `f(a,b,c)` dentro de la guarda del condicional, y la variable j el término *hacia*. La relación

[6] La memoria de *heap* de un programa es un espacio de memoria dinámico, con quien los programas suelen interactuar para solicitar espacio de memoria durante su ejecución.

```
if (f(a,b,c)) { // condicional con su guarda
    // ...
    let j = c;  // primer uso de la variable `c`
    func(j); // segundo uso indirecto de la variable `c`
    // ...
}
```

Fig. 4.6: Snippet de código JavaScript que ilustra la relación entre dos términos capturada por la regla GUARDA(*desde*, *hacia*).

GUARDA establece que hay un flujo entre ambos términos, ya que, en caso de ser *desde* considerado relevante, y al controlar el flujo de ejecución del programa, existe la posibilidad que la llamada a `f` produzca alguna mutación en alguna de las variables utilizadas como parámetro. Entonces, GUARDA(a, b) será verdadero si existe una estructura similar a la mostrada en el ejemplo, y alguno de los parámetros parte de la llamada a función en la guarda del condicional, es usado dentro del cuerpo del mismo. En el anexo, la figura 7.2 muestra y explica algunos detalles de la implementación en QL de esta regla, la cual ilustra bien cómo el lenguaje de consulta permite escribir un predicado que maneja diferentes niveles de abstracción dentro de un programa.

4.3.4. Conjunto inicial de términos anotados

Como en Seldon, nuestra técnica también requiere de un conjunto inicial de términos de programa que serán usados para definir, en el modelo, variables con un rol conocido. Pero, como fue mencionado en la introducción de este capítulo, JELDON utiliza como entrada del proceso de inferencia un conjunto de consultas $\mathcal{Q}_M$. La etapa GENERAR ENTIDADES es responsable de ejecutar estas consultas para así obtener los términos de programa necesarios para la construcción del modelo.

El apéndice 7 muestra un ejemplo de un conjunto inicial de consultas para vulnerabilidades del tipo SQLInjection. Para cada uno de los tipos de vulnerabilidades estudiadas en este trabajo, los conjuntos de consultas iniciales siguen el mismo patrón de definir tres clases `Source`, `Sink` y `Sanitizer` por medio de las cuales podemos consultar si un término tiene un rol conocido. Siguiendo el mismo enfoque de definir una abstracción en Datalog con la cual trabajaremos en esta tesis, diremos que un término t tiene un rol conocido por medio de $\mathcal{Q}_M$ si alguno de los siguientes hechos es cierto:

$$\begin{aligned} &\text{EsSource}_{\mathcal{Q}_M}(t). \\ &\text{EsSink}_{\mathcal{Q}_M}(t). \\ &\text{EsSource}_{\mathcal{Q}_M}(t). \end{aligned} \tag{4.10}$$

4.3.5. Traducción a representación independiente

Como en Seldon, y como fue explicado en 4.2, nuestra técnica también utiliza una representación programa independiente. Esta es necesaria al momento de construir el modelo para poder traducir un término t a una variable que pasará a representar, no el término del programa, sino su representación independiente $Repr(t)$. Esto permite que las inferencias producidas por el modelo puedan extrapolarse a diferentes programas.

Como tanto los vértices de nuestro grafo como los términos cuyos roles son conocidos, y obtenidos por medio de $\mathcal{Q}_M$, son términos de programa, necesitamos una función que nos permita traducir desde el universo de términos a sus representaciones independientes. Para ello, construiremos una función $\theta : \mathcal{T} \to \rho(\mathcal{R})$, con $\mathcal{T}$ el universo de términos de programa como dominio de la función, y el conjunto de partes de $\mathcal{R}$, que denota todas las posibles representaciones construidas por $Repr$.

El procedimiento que transforma un término t a su representación $r = Repr(t)$, implementado en CodeQL, parte del término t y recorre las reglas definidas en 4.2 recursivamente, agregando a r la expresión que corresponda al término. Para acotar las estructuras producidas por el procedimiento recursivo, introducimos una **profundidad máxima** MaxReprDepth, la cual indica tanto la cantidad máxima de pasos recursivos que descomponen t, como la longitud máxima de la expresión que conforma a r. Luego, podemos definir θ de la siguiente manera:

$$\theta(t) = \{r / r = Repr(t) \wedge |r| \leq \text{MaxReprDepth}\} \tag{4.11}$$

Hiperparámetro: MaxReprDepth

Controla la longitud máxima con la cual pueden ser formadas las representaciones independientes de un término t.
La longitud de una representación independiente r está definida por la cantidad de expresiones anidadas que la misma tiene, con $*$ y las sentencias no recursivas de la gramática siendo el caso base con valor 0. Por ejemplo, `(parameter 1 (member find *))` y `(member document (global))` tienen longitud 2.

4.3.6. Puesta en común

Tras haber descripto las relaciones necesarias para caracterizar tanto los vértices como los ejes de grafo de propagación, podemos definir una relación que será el elemento principal de nuestra consulta a CodeQL, para extraer el grafo. Si recordamos las intuiciones acerca de cómo se mueve el flujo de información en un programa, explicadas en 3.3.1, están siempre compuestas por relaciones entre tres vértices diferentes del grafo. Entre estos vértices, dos tienen un rol conocido, y otro desconocido, pero siempre un potencial source, sink y sanitizer. Por este motivo, y para hacer la traducción del grafo al modelo más simple, representaremos el grafo por medio de la relación Tripla(a, b, c), siendo a, b y c los potenciales source, sanitizer y sink.

Esta relación es el elemento principal de nuestra consulta, y se construye a partir de todos los predicados antes definidos. El mismo predica sobre el potencial rol que cumple cada variable en la relación, y si una es alcanzable desde la otra. A continuación presentamos la definición:

$$\begin{aligned}
&\text{Tripla}(source, san, sink) \leftarrow \\
&\quad \text{CandidatoASanitizer}(san),\ \text{AlcanzableDesdeSource}(src, san), \\
&\quad \text{Alcanzable}(san, sink),\ \text{CandidatoASink}(sink). \\
&\text{AlcanzableDesdeSource}(src, dest) \leftarrow \\
&\quad \text{CandidatoASource}(src),\ \text{Alcanzable}(src, dest).
\end{aligned} \tag{4.12}$$

Dado un programa ρ, podemos realizar las siguientes consultas por medio de CodeQL, para obtener los resultados parciales esperados en la etapa GENERAR ENTIDADES:

$$\begin{aligned}
&\textsc{Tripla}(source, sanitizer, sink).\\
&\textsc{EsSource}_{\mathcal{Q}_M}(sourceConocido).\\
&\textsc{EsSanitizer}_{\mathcal{Q}_M}(sanitizerConocido).\\
&\textsc{EsSink}_{\mathcal{Q}_M}(sinkConocido).\\
&\theta(t, r).
\end{aligned} \tag{4.13}$$

Con $\theta(t, r)$ una relación definida como $\theta(t, r) \leftarrow r = Repr(t)$. Luego, habremos obtenido el grafo de propagación G_ρ definido por la relación TRIPLA, el conjunto de términos conocidos $\mathcal{A}^\rho_M$, definidos por los resultados de las relaciones ESSOURCE, ESSANITIZER y ESSINK, y la función de traducción de términos a su representación independiente θ.

Como comentamos al presentar la arquitectura de nuestra técnica, JELDON construye un modelo individual para cada programa en $\mathcal{D}$ combinando luego los resultados parciales obtenidos por cada uno, a diferencia de Seldon, que construye un único modelo para todos los programas. Como las variables de cada modelo individual refieren a la representación programa independiente n de un término relevante, y no al término en sí mismo, se puede llevar a cabo el proceso de "combinar" modelos diferentes antes o después de su optimización. En el primer caso, la unificación de las variables que refieren a un mismo n ocurre al momento de construcción del modelo. En el segundo, se debe realizar una agregación sobre los valores que el modelo resuelto les asignó a las variables que representan n.

Durante la implementación de nuestra técnica notamos que el modelo resultante de un solo programa puede tener rápidamente no miles sino decenas de miles de vértices, y más aún, si se crea un modelo único para todos los programas en el conjunto de entrenamiento. Una serie de pruebas nos dieron resultados similares para combinar los puntajes de modelos más pequeños (como será explicado más adelante), comparado con crear un modelo único, por lo que se seleccionó el segundo enfoque. Más allá de la reducción del tamaño del modelo de optimización, esta decisión permite que las primeras etapas del proceso de inferencia tengan la posibilidad de ser ejecutadas en paralelo, mejorando así la performance de la técnica (esta idea es planteada como trabajo a futuro en las conclusiones).

4.4. Del grafo al modelo

Como vimos en la visión arquitectural de nuestra técnica, la formulación del modelo a partir del grafo, los términos anotados y la función de traducción, se llevan a cabo en la etapa llamada GENERAR MODELO, en el contexto de un programa ρ. Por motivo de síntesis en la notación algebraica, todas las variables en esta sección deben ser consideradas en el contexto del programa ρ ya que se omitirá agregar un sub o super índice indicando esto. Por ejemplo, cuando hablemos del modelo $\mathcal{M}$ o del conjunto de restricciones de flujo C^{flow}, considerando nuestro contexto, estaremos hablando de $\mathcal{M}_\rho$ y C^{flow}_ρ.

En las secciones 3.3 y 3.4 se muestra cómo en Seldon, partiendo del grafo y el conjunto de términos anotados, se puede construir el modelo de optimización lineal (tipo de problemas que también nos referiremos como programación lineal, o LP). Recapitulando,

tengamos presente que este está conformado por un conjunto de variables con la forma n_t^{rol}, con t un término y rol entre source, sink y sanitizer. Sobre las variables que son parte de este conjunto aplican tres tipos de restricciones:

1. Acotaciones sobre los valores que pueden tomar las variables, de forma de poder interpretarlas como probabilidades, y los valores que pueden tomar las variables de error ϵ_i. El conjunto de estas restricciones es llamado C^{var}.

2. A las variables que representan términos de programa anotados, se restringe el valor de la variable con dicho rol a un valor de 1, y el resto a 0. Estas conforman el conjunto C^{known}.

3. Por último, y con rol protagónico, el conjunto de restricciones relajado que surge de formalizar las intuiciones presentadas en la sección 3.3.1 para capturar el problema de inferencia como un problema de LP. Este es llamado C^{flow}.

Nuestra técnica utiliza el modelo descripto por Seldon sin mayores cambios. La formulación del problema de inferencia se lleva a cabo utilizando las mismas intuiciones (3.3) que predican sobre "triplas" de nodos en el propagation graph (de aquí el origen de nuestra relación de Tripla al producir el grafo). Los únicos cambios realizados sobre la formulación algebraica del modelo consisten en reordenar las restricciones en C^{flow} , de manera que la traducción al formato esperado por el solver utilizado sea más simple.

El solver utilizado, descripto en la sección 4.5, consume un formato de descripción de modelos de LP compuesto por tres partes: una función objetivo con su variante (*Maximize* o *Minimize*), una serie de restricciones sobre las variables, y un apartado que describe los límites de las variables. La función objetivo debe ser una función sobre las variables del modelo, y las restricciones deben tener la forma $L \star k$, con L una función sobre las variables, $\star$ un operador de comparación como $\leq, \geq, =$ y k una constante. Por último, los límites sobre las variables son descriptos como $a \leq var \leq b$.

La formulación de las restricciones de flujo en su versión relajada, presentadas en la sección 3.4, tienen la variable de error ϵ en su lado derecho. Por ello, debemos reescribirlas de forma que el lado derecho de la inecuación sea una constante, en este caso C. Tomemos como ejemplo para esta reescritura la restricción presentada en 3.5, en su versión relajada 3.8:

$$\begin{aligned} (n_{san})^{san} + (n_{snk})^{snk} - \sum_{j=1}^{k} n_j^{src} - C \leq \epsilon_i \\ (n_{source})^{src} + (n_{sink})^{snk} - \sum_{i=1}^{k} n_i^{san} - \epsilon_i \leq C \end{aligned} \tag{4.14}$$

Luego, si tomamos el conjunto de restricciones de flujo C^{flow}, relajadas, y reescritas como fue mostrado antes, podemos formular el problema de optimización $\mathcal{M}$ de la siguiente manera:

$$min \left(\sum_{i=1}^{M} \epsilon_i + \lambda \sum_{r \in \mathcal{R}} ((n_r)^{src} + (n_r)^{san} + (n_r)^{snk}) \right) \tag{4.15}$$

$$\text{sujeto a } 0 \leq (n_r)^{rol} \leq 1 \; \forall n_r \text{ y} \tag{4.16}$$

$$0 \leq \epsilon_i \; \forall i \in \{1, ..., M\} \text{ y} \tag{4.17}$$

$$\begin{cases} (n)^r = 1 \\ (n)^t = 0 \quad \forall t \in \{src, san, snk\} \setminus \{r\} \end{cases} \quad \forall \langle v, r \rangle \in \mathcal{A}_M \mid n = Rep(v) \tag{4.18}$$

$$\text{y cada una de las } M \text{ restricciones } c_i \text{ en } C^{flow} \tag{4.19}$$

Haciendo hincapié una vez más, JELDON utiliza la misma formulación del problema de optimización presentada en Seldon (3.9).

Hiperparámetro: C y λ

Hiperparámetros del modelo previamente definidos en la sección 3.4. C es una constante, y λ un parámetro de regularización para penalizar valores altos en las variables.

Nos queda explicar de qué manera, usando las resultados parciales que conforman la entrada de la etapa actual de la técnica, podemos construir el modelo antes descripto. El algoritmo 1 muestra cómo usando G_ρ, θ_ρ y $\mathcal{A}_M^\rho$, podemos construir los cuatro elementos que conforman el modelo. Por motivos de síntesis, en el algoritmo se hace uso del operador $A \xleftarrow{\cup} B$ para A un conjunto. Este es equivalente a la siguiente expresión $A \leftarrow A \cup \{B\}$. También en el algoritmo se usa un tipo de datos descripto por la forma $\langle \vec{x} \rangle$. Este representa una n-upla cuyos elementos son los del vector $\vec{x}$. Por último, el algoritmo hace uso de diccionarios en los casos en los que una variable es inicializada con el valor $D \leftarrow \{\}$. Estos permiten escrituras y lecturas, dada una clave k: $D[k]$ y $D[k] \leftarrow v$.

La primera parte del algoritmo construye una serie de estructuras de soporte para computar fácilmente las restricciones de flujo. Los resultados de evaluar sobre ρ la relación TRIPLA denotan el propagation graph G_ρ, usado para construir las restricciones de flujo. Tomemos por ejemplo la restricción que surge de la figura 3.1a. Esta dice que dados dos vértices san y snk con potencial rol sanitizer y sink, tales que $san \to snk \in E$, luego alguno de los vértices v anteriores a san tales que $v \to san \in E$ debería tener un rol de source. Relacionando esto con la relación TRIPLA(v, san, snk), y asumiendo que san y snk son dados, sería de utilidad hallar los v que hacen valer la relación. De esta forma, podríamos construir la restricción tomando las variables que representan cada uno de los términos involucrados.

Otra característica a tener en cuenta es que las variables del modelo son asignadas no sobre el conjunto de términos $\mathcal{T}$, sino sobre el conjunto de representaciones independientes de los términos $\mathcal{R}$. Para ello, en el algoritmo 1 usaremos la función $\mathcal{N} : \mathcal{T} \to \mathcal{V}$, con $\mathcal{V}$ el conjunto de posibles variables dentro del modelo. En el caso de que un término t tenga una única representación posible r, luego $\mathcal{N}(t) = n_r$, con n_r una variable única en el modelo para r. Por otro lado, es posible que un término t tenga más de una representación posible. El algoritmo no muestra en detalle cómo se manejan estos casos para simplificar su comprensión, pero asumamos que un término t tiene las siguientes representaciones independientes posibles $\{r_1, .., r_m\}$. Luego, $\mathcal{N}$ trataría cada una de estas representaciones como una variable diferente, de manera que cada una de las representaciones n_{r_i} sería parte de las siguientes restricciones:

- Si t es parte del conjunto de especificaciones iniciales, se agregaría la correspondiente restricción en C^{known}.

- Si t es parte de una relación de TRIPLA, esta se "expandiría" de forma que exista dicha relación para cada una de las representaciones de t. Por ejemplo, si t es parte de la relación $\text{TRIPLA}(t, a, b)$, luego la relación sería "expandida" en el modelo al equivalente de si las relaciones $\text{TRIPLA}(n_{r_i}, a, b)$ fueran parte del grafo.
- Se agregan las restricciones necesarias que limitan los valores de las nuevas variables n_{r_i}.

Como fue mencionado en la sección donde se introdujo la arquitectura de la técnica, esta etapa debe producir también la inversa de la función de asignación $\mathcal{N}$, de forma que una vez obtenidos los resultados de optimizar el modelo, se pueda convertir una asignación a una variable, a una asignación de un rol y un puntaje a un término. Esta inversa $\mathcal{N}^{-1}$ se construye simplemente mediante la biyección entre el identificador de cada variable asignada, y la representación independiente a la cual se asigna.

A continuación, describimos cada parte del algoritmo:

- **Parte 1**: Para cada uno de los tipos de restricciones de flujo se construyen las estructuras de soporte descriptas anteriormente de manera que, dados dos términos fijos, podamos encontrar un tercero con rol el faltante.
- **Parte 2**: Dados los términos de programa anotados $\mathcal{A}_M$, construye para cada uno las restricciones sobre el rol que deben que tener asignado.
- **Parte 3**: Construye, para las variables que representan a cada término usado en el modelo, restricciones indicando que sus valores pueden fluctuar entre $0 \leq v \leq 1$.
- **Parte 4.1**: Utilizando las estructuras construidas en la primera parte, agrega las restricciones de flujo para 3.1a. También preserva la nueva variable de error creada para ser usada en la función objetivo.
- **Parte 4.2**: Utilizando las estructuras construidas en la primera parte, agrega las restricciones de flujo para 3.1b. También preserva la nueva variable de error creada para ser usada en la función objetivo.
- **Parte 4.3**: Utilizando las estructuras construidas en la primera parte, agrega las restricciones de flujo para 3.1c. También preserva la nueva variable de error creada para ser usada en la función objetivo.

4.5. Optimización

Para optimizar el modelo de inferencia, Seldon utiliza un solver que implementa una técnica de optimización no pensada para problemas de optimización lineal. Eso requiere cierta traducción en la formulación del problema. Para eliminar la necesidad de esta interfaz, nuestra técnica utiliza un solver y un formato de descripción de modelos, diseñados para tratar con problemas de programación lineal algebraica (LP).

Como solver utilizamos CLP[7], un solver open-source desarrollado por COIN-OR[8] que implementa los algoritmos de Simplex primal y dual. Este permite ser usado como biblioteca, o como un programa independiente. Optamos por el segundo método, pudiendo

[7] Repositorio de GitHub de CLP.
[8] Computational Infrastructure for Operations Research.

comunicar el modelo a optimizar en un formato conocido como CPLEX LP[9], utilizado para describir problemas de LP en su formulación algebraica. El siguiente ejemplo muestra un modelo codificado en CPLEX LP:

```
Optimize
 COST:    XONE + 4*YTWO + 9*ZTHREE
Subject To
 LIM1:    XONE + YTWO          <= 5
 LIM2:    XONE        + ZTHREE >= 10
 MYEQN:         - YTWO + ZTHREE  = 7
Bounds
       XONE <= 4
 -1 <= YTWO <= 1
End
```

Una vez obtenidos los resultados del proceso de optimización, necesitamos convertir las asignaciones de variables de vuelta a nuestro universo de términos y representaciones. Para ello, usamos la inversa de la función $\mathcal{N}$ definida en el algoritmo 1, que nos permite, dada una variable del modelo, obtener la representación independiente del término al cual corresponde. Dada una asignación resultante de la forma $n^{rol} = score$ donde $score \in \mathbb{R}_{[0,1]}$, podemos construir el conjunto de términos anotados resultante, conformado por $\langle \mathcal{N}^{-1}(n), rol, score \rangle$.

4.6. Combinar inferencias

Una vez obtenidos los resultados de las primeras tres etapas de la técnica (GENERAR ENTIDADES, GENERAR MODELO y OPTIMIZAR) para cada programa en $\mathcal{D}$, habremos obtenido las especificaciones inferidas en cada $\mathcal{A}_I^\rho$. La última etapa de la técnica, COMBINAR PUNTAJES, es responsable de reducir los diferentes conjuntos de términos anotados a un único conjunto, y luego traducirlo al conjunto de consultas $\mathcal{Q}_I$, que contenga los términos, sus roles y sus valores de confianza (las puntuaciones).

La sección anterior explica que los resultados generados por el solver son convertidos del universo del modelo (donde cada variable corresponde a un término y su rol), a uno donde un rol y un puntaje son asignados a un término en su representación independiente. Una asignación en un conjunto resultado tiene una forma $\langle r, rol, score \rangle$, con r una representación en $\mathcal{R}$, rol un valor entre $\{\text{source}, \text{sanitizer}, \text{sink}\}$, y $score$ un valor en $\mathbb{R}$. El objetivo de esta etapa es combinar cada $\mathcal{A}_I^\rho$ de forma que si un término cuya representación es w, y cuyo rol inferido es r, tiene asignados las puntuaciones $s_1, ..., s_i$ en los programas $\rho_1, .., \rho_i$, luego $\langle w, r, \mathcal{Z}(s_1, .., s_n) \rangle$ sea vea reflejado en $\mathcal{Q}_I$. La función $\mathcal{Z} : \mathbb{R}^n \to \mathbb{R}$ se encarga de combinar los puntajes obtenidos por cada término candidato.

Además, esta etapa tiene la potestad de descartar las inferencias que crea y que no son relevantes para el conjunto de consultas final. Durante el desarrollo de la técnica, descubrimos que tanto sources y sanitizers son roles donde las especificaciones inferidas no ayudaban mucho a mejorar la performance resultante de evaluar el análisis de código final. Primero, y para los tipos de vulnerabilidades cuyos conjuntos de consultas intentamos mejorar con la técnica, los términos cuyo rol es de source, suelen tener una estructura

[9] https://web.mit.edu/lpsolve/doc/CPLEX-format.htm

muy similar, ya que los puntos de un programa donde un usuario puede introducir valores son comúnmente pedidos HTTP, o algún otro protocolo utilizado para desarrollo de aplicaciones web. Esto hace que las consultas desarrolladas para encontrarlos tengan un muy buen desempeño, ya que suelen modelar las bibliotecas que implementan estos protocolos. Luego, en el caso de los sanitizers, sin contar casos particulares donde una biblioteca implementa alguna lógica de sanitización, estos suelen ser procedimientos definidos de manera ad-hoc para cada programa. Más aún, teniendo en cuenta que la definición de qué términos podrían llegar a ser considerados como sanitizers, en 4.8, es muy generalista, esto hace la técnica muy propensa a falsos positivos. Por estas razones, **esta etapa tiene en cuenta solamente los resultados cuyo rol inferido es sink**. Como la técnica infiere los otros dos tipos de roles, pero no los utiliza, una posibilidad de trabajo a futuro es investigar formas de mejorar las cualidades de la técnica al inferir sources y sanitizers.

El algoritmo 2 explica el procedimiento que Jeldon lleva a cabo para combinar los conjuntos resultado de cada programa. Nuestra técnica utiliza como $\mathcal{Z}$ la media de las puntuaciones. El procedimiento GenerarConsulta se encarga de traducir el conjunto de términos anotados, en su forma $\langle rep, rol, score \rangle$, a una consulta válida en QL que puede ser usada en el contexto de taint analysis. Como los detalles de implementación de esta traducción carecen de valor para las ideas centrales de nuestra técnica, se hará omisión de los mismos.

Más allá del proceso de traducción de un conjunto de términos a la consulta, podemos ver de qué forma entran en juego las especificaciones inferidas al momento de denotar en taint analysis si un término tiene un rol asignado. En 4.10, definimos una serie de relaciones en Datalog, que representan el conjunto de especificaciones iniciales $\mathcal{Q}_M$. Manteniendo la misma aridad de las relaciones, podemos definir siguiendo detalles usados en su implementación real en QL, de qué forma las especificaciones inferidas cooperan con las iniciales. Primero, la relación EsSinkInferido(t) especifica si un término puede ser considerado como sink dados los resultados del proceso de inferencia. La relación Resultado se construye a partir del conjunto de términos anotados obtenido de la agregación antes descripta, mientras que Repr representa la rutina de traducción entre términos a representaciones independientes mencionada en 4.3.5. Las constantes MinScore$_{rol}$ nos permiten filtrar inferencias donde el puntaje asignado no es significativo, previniendo falsos positivos.

$$\begin{aligned}
&\textsc{Resultado}(t, rol, score) \leftarrow \textsc{Repr}(t, r),\ \langle r, rol, score \rangle \in \mathcal{A}_I.\\
&\textsc{EsSourceInferido}(t) \leftarrow \textsc{Resultado}(t, \textsc{source}, score),\ score \geq \textsc{MinScore}_{src}.\\
&\textsc{EsSanitizerInferido}(t) \leftarrow \textsc{Resultado}(t, \textsc{sanitizer}, score),\ score \geq \textsc{MinScore}_{san}.\\
&\textsc{EsSinkInferido}(t) \leftarrow \textsc{Resultado}(t, \textsc{sink}, score),\ score \geq \textsc{MinScore}_{snk}.
\end{aligned} \tag{4.20}$$

Una vez definida una relación que nos permite corroborar si un término tiene un rol asignado en las especificaciones inferidas, este se puede incorporar con las especificaciones iniciales llegando a la relación final EsSink.

$$\textsc{EsSink}(t) \leftarrow \textsc{EsSink}_{\mathcal{Q}_M}(t);\ \textsc{EsSinkInferido}(t). \tag{4.21}$$

Hiperparámetro: MinScoreRol

Parámetro que permite controlar a partir de qué valor de los puntajes inferidos consideramos a la asignación de un rol a un término, como válida. Este parámetro no afecta los resultados del proceso de inferencia, sino la evaluación del mismo, y sus resultados a la hora de utilizar el nuevo conjunto de consultas.
Como se mencionará en el capítulo 5, aumentar el valor de cada MinScoreRol, causará un incremento en la precisión y una disminución en el recall.

4.7. Cierre

En este capítulo presentamos el funcionamiento de Jeldon , en primer término, al mostrar su arquitectura. Se describieron cada una de las etapas que llevan a cabo el proceso de inferencia de especificaciones de taint analysis, explicando las diferencias con la implementación original de Seldon, y formalizando tanto el proceso de generación del grafo mediante Datalog, como el modelo, mediante su representación algebraica. Se mostró también una representación independiente, diferente a la usada en la publicación original, que permite generalizar la estructura de términos fácilmente, así como caracterizar interacciones complejas entre diferentes elementos de un programa como son las *callbacks*.

Algoritmo 1 Generar modelo

Requiere $\mathcal{T}$ conjunto de términos en el programa. Puede ser computado del dominio de θ

Requiere $\mathcal{A}_M$ conjunto de términos cuyo rol es conocido

Requiere $\mathcal{TR}$ conjunto resultado de la evaluación del predicado TRIPLA

Requiere Una función $\mathcal{N}$ que dado un término t y la función de traducción a representaciones independientes θ, asigna una variable dentro del modelo a $\theta(t)$

$C^{flow} \leftarrow \emptyset$
$C^{var} \leftarrow \emptyset$
$C^{known} \leftarrow \emptyset$
$\mathcal{E} \leftarrow \emptyset$
SOURCESANITIZER $\leftarrow \{\}$
SANITIZERSINK $\leftarrow \{\}$
SOURCESINK $\leftarrow \{\}$
for $\langle src, san, sink\rangle \in \mathcal{TR}$ **do** ▷ Parte 1
  SOURCESANITIZER$[\langle src, san\rangle] \leftarrow^{\cup} sink$
  SANITIZERSINK$[\langle san, sink\rangle] \leftarrow^{\cup} src$
  SOURCESINK$[\langle src, sink\rangle] \leftarrow^{\cup} san$
end for
for $\langle t, rol\rangle \in \mathcal{A}_M$ **do** ▷ Parte 2
  $C^{known} \leftarrow^{\cup} \{\mathcal{N}(t)^{rol} = 1\}$
  for $otro \in \{\text{source}, \text{sink}, \text{sanitizer}\} \setminus \{rol\}$ **do**
    $C^{known} \leftarrow^{\cup} \mathcal{N}(t)^{otro} = 0$
  end for
end for
for $t \in \mathcal{T} \wedge rol \in \{\text{source}, \text{sink}, \text{sanitizer}\}$ **do** ▷ Parte 3
  $C^{var} \leftarrow^{\cup} 0 \leq \mathcal{N}(t)^{rol} \leq 1$
end for
for $\langle\langle san, snk\rangle, \beta\rangle \in$ SANITIZERSINK $\wedge\, \epsilon =$ nueva variable **do** ▷ Parte 4.1
  $C^{flow} \leftarrow^{\cup} \mathcal{N}(san)^{san} + \mathcal{N}(snk)^{snk} - \sum_{s\in\beta} \mathcal{N}(s)^{src} - \epsilon \leq C$
  $\mathcal{E} \leftarrow^{\cup} \epsilon$
end for
for $\langle\langle src, san\rangle, \beta\rangle \in$ SOURCESANITIZER $\wedge\, \epsilon =$ nueva variable **do** ▷ Parte 4.2
  $C^{flow} \leftarrow^{\cup} \mathcal{N}(src)^{src} + \mathcal{N}(san)^{san} - \sum_{s\in\beta} \mathcal{N}(s)^{snk} - \epsilon \leq C$
  $\mathcal{E} \leftarrow^{\cup} \epsilon$
end for
for $\langle\langle src, snk\rangle, \beta\rangle \in$ SOURCESINK $\wedge\, \epsilon =$ nueva variable **do** ▷ Parte 4.3
  $C^{flow} \leftarrow^{\cup} \mathcal{N}(src)^{src} + \mathcal{N}(snk)^{snk} - \sum_{s\in\beta} \mathcal{N}(s)^{san} - \epsilon \leq C$
  $\mathcal{E} \leftarrow^{\cup} \epsilon$
end for
$\mathcal{O} \leftarrow \lambda \sum_{t\in\mathcal{T}} (\mathcal{V}(t)^{src} + \mathcal{V}(t)^{san} + \mathcal{V}(t)^{snk}) + \sum_{\epsilon\in\mathcal{E}} |\epsilon|$
return $\langle \mathcal{O}, C^{flow}, C^{var}, C^{known}\rangle$

Algoritmo 2 Combinar puntajes

Requiere $\mathcal{Q}_I^\rho$ conjunto de términos inferidos $\forall \rho \in \mathcal{D}$
  $\mathcal{P} \leftarrow$ diccionario de conjuntos $\{\}$
  $\mathcal{R} \leftarrow \emptyset$ ▷ Conjunto de claves de $\mathcal{P}$
  $\mathcal{A}_I \leftarrow \emptyset$
  for $\rho \in \mathcal{D}$ **do**
    for $\langle r, rol, score \rangle \in \mathcal{Q}_I^\rho$ **do**
      if $rol = \text{sink}$ **then** ▷ Solo tenemos en cuenta inferencias sobre sinks
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{r\}$
        $\mathcal{P}_r \leftarrow \mathcal{P}_r \cup \{score\}$ ▷ $\mathcal{P}_r$ es el conjunto asociado con la representación r
      end if
    end for
  end for
  for $r \in \mathcal{R}$ **do**
    $\mathcal{A}_I \leftarrow \mathcal{A}_I \cup \{\langle r, \text{sink}, \frac{1}{|\mathcal{P}_r|} \sum_{s \in \mathcal{P}_r} s \rangle\}$ ▷ Tomamos para cada representación, la media de todos sus puntajes
  end for
  $\mathcal{Q}_I \leftarrow$ GenerarConsulta($\mathcal{A}_I$)

5. EVALUANDO LA TÉCNICA

Habiendo presentado en el capítulo anterior una posible solución al problema de inferencia de especificaciones taint analysis, necesitamos una manera de confirmar si la información inferida es o no correcta. Para ello, es necesario considerar las especificaciones dentro del contexto del análisis estático estudiado. Las especificaciones inferidas mejoran la cobertura del análisis, pudiendo detectar nuevos casos donde existe flujo de información entre un source y un sink, situación que referiremos como **alertas**. Por ello es esperable que, si las especificaciones inferidas son de utilidad, el analizador encuentre mayor cantidad de alertas con ellas que sin ellas. Podríamos así decir que una forma de evaluar si las especificaciones inferidas son de utilidad es mediante la diferencia entre el conjunto de alertas encontrado usando las especificaciones iniciales $\mathcal{AL} = \textsc{Evaluar}(\mathcal{Q}_M, \rho)$, y el conjunto encontrado usando la combinación de las especificaciones originales con las inferidas $\mathcal{AL} = \textsc{Evaluar}(\mathcal{Q}_B, \rho)$, donde $\mathcal{Q}_B = \mathcal{Q}_M \cup \mathcal{Q}_I$. El resultado de esta diferencia sería el conjunto de alertas encontradas por medio de las nuevas especificaciones, que sin ellas no hubieran sido reportadas. Como las técnicas de análisis estático sobreaproximan el comportamiento de un programa, los resultados producidos pueden no ser verdaderos. Esto conlleva a que ambos conjuntos de alertas encontradas puedan tener resultados espurios, lo cual nos hace preguntarnos: ¿Cómo corroboramos si una nueva alerta descubierta por $\mathcal{Q}_B$ es cierta? Más aún, ¿Cómo corroboramos si cualquier alerta reportada es realmente cierta?

Seldon resuelve este problema mediante la inspección manual de las alertas detectadas. En [3] los autores reportan que, del total de las alertas encontradas tanto por las especificaciones iniciales como las inferidas, 662 y 21318 respectivamente, se estudió de manera manual una muestra aleatoria de 25 alertas para cada una, pudiendo así corroborar si eran verdaderas. Corroborar si cada una de estas alertas es realmente cierta conlleva una labor importante, ya que es necesario primero adentrarse y entender el programa donde es reportada, y realizar un seguimiento desde el source hasta el sink, sino una prueba de concepto demostrando que la alerta reportada es correcta.

Nuestro trabajo presenta una nueva metodología de evaluación que permite realizar la validación de si un alerta atribuida al conjunto de especificaciones inferido es cierta o no, de manera automática. Esto nos permite utilizar la totalidad de las alertas halladas para medir la performance del aprendizaje logrado por la técnica.

Mediante esta metodología, responderemos nuestras preguntas de investigación:

1. **RQ1**: ¿La técnica presentada en Seldon, puede ser usada para descubrir nuevas especificaciones de taint analysis en otros lenguajes, como JavaScript?

2. **RQ2**: ¿Son los resultados de nuestra técnica comparables con los de Seldon?

3. **RQ3**: ¿Es útil la técnica para descubrir nuevas alertas, y pueden las mismas constituir un aporte para la comunidad?

Este capítulo describe la metodología de evaluación usada. Por medio de ella, obtuvimos métricas indicativas sobre la performance de la técnica dentro del contexto de inferencia de especificaciones de taint analysis. Luego, se discuten los resultados obtenidos

en conjunto con un análisis cualitativo de cómo fueron seleccionados los programas usados como entrada para el proceso de inferencia. Por último, se analiza si es posible realizar una comparación de los resultados obtenidos, contra los de Seldon, y finalmente, se describen una serie de contribuciones a CodeQL, producto de este trabajo.

5.1. Metodología de evaluación

Como se mencionó en la introducción de este capítulo, pudiendo obtener para un programa el conjunto de alertas atribuidas a las especificaciones inferidas, surge el problema de cómo corroborar si estas alertas son realmente ciertas. Debido a la naturaleza del análisis estático de código, donde se sobreaproxima el comportamiento de un programa para poder predicar sobre o atacar todas las ejecuciones posibles, es muy común obtener resultados falsos positivos. En nuestro caso, esto ocurre cuando una alerta hallada mediante las especificaciones inferidas (EI), resulta no ser verdadera, a causa de suponer que existe flujo de información entre términos donde no lo hay, modelar un comportamiento del programa de manera errónea, o simplemente ser un caso donde realmente no es posible que ocurra.

Existen dos metodologías para poder conocer, dentro de un conjunto de alertas, cuáles realmente son verdaderas. La primera, usada por Seldon, consiste en la inspección manual de la validez de cada alerta. Más allá de lo tedioso del proceso, es una metodología muy usada para validar vulnerabilidades sobre programas, o para realizar una curaduría fina. La segunda consiste en tener o construir un oráculo, un analizador de referencia (ADR) cuyos resultados consideramos como verdad absoluta, contra el cual podemos comparar un conjunto de resultados obtenidos, determinando así si son o no verdaderos.

Con el segundo enfoque, al comparar las alertas obtenidas contra un analizador de referencia, que nos indica si las mismas son verdaderas o falsas, podemos construir una **matriz de confusión**. Estas matrices son usadas en el campo de aprendizaje automático para poder clasificar el tipo de error o acierto que una técnica de aprendizaje origina en sus predicciones. En nuestro caso, Jeldon infiere, o "aprende", nuevas especificaciones de taint analysis. Por lo tanto, queremos caracterizar los errores y los aciertos que las especificaciones inferidas producen cuando son usadas dentro de un analizador, descubriendo alertas. Esta categorización nos permite calcular dos métricas muy usadas al momento de evaluar cuantitativamente una técnica de aprendizaje, **precision** y **recall**.

Jeldon (EI)	Analizador de referencia (ADR)	
	V	F
V	Verdadero positivo (TP)	Falso positivo (FP)
F	Falso negativo (FN)	Verdadero negativo (TN)

Tab. 5.1: Matriz de confusión comparando las alertas obtenidas por medio de las especificaciones inferidas, y el analizador de referencia. V es verdadero, y F es falso.

La tabla 5.1 muestra, para una alerta descubierta mediante las especificaciones inferidas, cómo debe ser caracterizada, dependiendo de si es catalogada como verdadera o falsa por el ADR. Analicemos en detalle qué significa cada una de las posibles clases:

- **TP**: Una alerta fue encontrada mediante el uso de las EI, y también fue descubierta por el analizador de referencia. Como los resultados producidos por el ADR son considerados como verdad absoluta, la alerta es considerada como verdadera, y por consiguiente, la predicción generada usando las EI, también.

- **FP**: Una alerta encontrada mediante las EI no forma parte del conjunto resultado del ADR. Por consiguiente, es considerada como un caso donde las especificaciones inferidas llevaron a una alerta espuria.
- **FN**: Una alerta es descubierta por el ADR, considerada entonces como verdadera, pero no es parte del conjunto resultado de las EI. Por consiguiente, las especificaciones inferidas no pudieron "recuperar" esta alerta.
- **TN**: En el dominio de nuestro problema, el hecho de que una alerta sea falsa, o negativa, significa la ausencia de la misma en el conjunto resultado de un analizador. La categorización de un resultado como *verdadero negativo* ocurre cuando una alerta es catalogada como falsa tanto por el ADR, como también mediante las EI. Esto quiere decir que la misma no fue detectada por ninguna de las dos partes. Al no ser encontrada por ninguna, esta condición es imposible de corroborar. Por este motivo, no vamos a tener en cuenta este tipo de casos.

La ausencia del último tipo de resultado no afectará a nuestra metodología para calcular las métricas de aprendizaje, ya que las mismas no dependen de TN.

Tras haber definido cómo podemos comparar y catalogar los resultados obtenidos del analizador de referencia, y las especificaciones inferidas, podemos definir formalmente métricas mediante las cuales evaluaremos si nuestra técnica es capaz de descubrir nuevas alertas. Utilizando la cantidad de ocurrencias de cada clase de resultado en la matriz de confusión, podemos definir precision y recall de la siguiente manera:

$$\begin{aligned} Precision &= \frac{TP}{TP + FP} \\ Recall &= \frac{TP}{TP + FN} \end{aligned} \tag{5.1}$$

Los valores de ambas métricas oscilan entre 0 y 1, ya que $TP \leq TP + FP$ y $TP \leq TP + FN$. En primer término, **precision** corresponde a una proporción sobre las alertas descubiertas por las EI, que mide cuántas del total son realmente verdaderas. En caso de que el valor sea 1, esto querría decir que $FP = 0$, que significa que el total de las alertas descubiertas por las EI también son catalogadas así por el analizador de referencia, y por consiguiente, verdaderas. Si la precisión es baja, indica que una gran parte de las alertas descubiertas por las EI son falsos positivos.

Por otro lado, recall mide, del total de alertas producidas por el ADR que consideramos realmente verdaderas, cuántas fueron descubiertas, o "recuperadas" por las EI. Esta métrica resulta particularmente útil para cuantificar la capacidad de aprendizaje de la técnica de inferencia, ya que está directamente relacionada a la cantidad de resultados nuevos y correctos producidos, en relación a la mejora que la técnica realizó sobre un conjunto de consultas existente.

5.1.1. Analizador de referencia

En la sección anterior presentamos dos métricas que nos permiten cuantificar la capacidad de aprendizaje, y la correctitud de los resultados producidos por un conjunto de especificaciones inferido mediante Jeldon . Para obtener los resultados necesarios y calcular esas métricas, es necesario construir un analizador de referencia contra el cual comparar

las alertas descubiertas por nuestras especificaciones inferidas. Como mencionamos en la introducción del capítulo 4, nuestra técnica parte de un conjunto de consultas iniciales, que especifican elementos de taint analysis para un tipo de vulnerabilidad en particular. Estas consultas son parte de las bibliotecas de consultas desarrolladas para CodeQL, proyecto con cientos de colaboradores en GitHub. Como el código fuente de estas bibliotecas se encuentra disponible de manera pública[1], nos permite acceder al mismo y a su historial de cambios. El hecho de tener disponible la historia de los cambios que fueron y van a ser desarrollados para las bibliotecas, nos permite navegar libremente entre versiones nuevas y antiguas de las mismas, así como revertir ciertas modificaciones como, por ejemplo, un commit que agrega nuevos sinks a la biblioteca[2].

El primer paso de la metodología de evaluación presentada en este trabajo consiste en construir, por medio de la historia de cambios de las bibliotecas, pares de conjuntos de consultas para cada vulnerabilidad. Para ejemplificar, tengamos en cuenta que estamos trabajando con un conjunto de consultas que predica sobre NoSQL Injection, un sub-tipo de vulnerabilidades de inyección. El método consiste en tomar primero una versión reciente de las bibliotecas, a la que llamaremos $\mathcal{Q}_{\mathcal{O}}$, como especificaciones que, usadas con el motor de análisis de CodeQL, constituyen nuestro analizador de referencia. Luego, accediendo al historial de cambios podemos tomar una versión anterior a $\mathcal{Q}_O$, e inclusive revertir ciertos commits que consideramos relevantes para las vulnerabilidades estudiadas, construyendo así una versión "peor", o antigua de las especificaciones. Este conjunto de consultas, al que llamaremos $\mathcal{Q}_W$, es el que usaremos como conjunto de especificaciones iniciales en nuestra técnica. Podemos pensar así que el método busca mejorar las especificaciones de entrada, para que las mismas puedan alcanzar las capacidades de detección de su versión mejorada $\mathcal{Q}_O$.

Como fue mencionado, primero debemos construir dos conjuntos de consultas que actuarán como las especificaciones iniciales esperadas como entrada en la técnica, $\mathcal{Q}_W$, y otro que será usado en conjunto con el motor de análisis como análisis de referencia, $\mathcal{Q}_O$. Además de estos, son necesarios dos conjuntos de programas, $\mathcal{D}_{train}$, usado como entrada del proceso de inferencia, y $\mathcal{D}_{eval}$, utilizado para evaluar las especificaciones inferidas. Es importante que $\mathcal{D}_{train} \cap \mathcal{D}_{eval} = \emptyset$, para prevenir un fenómeno conocido en aprendizaje automático como overfitting. La metodología consistirá entonces en, primeramente, generar un conjunto de especificaciones inferidas por JELDON , técnica que queremos evaluar. Luego, mediante los tres conjuntos de consultas ($\mathcal{Q}_W, \mathcal{Q}_O$, y la combinación del primero con las EI), se evaluará cada programa parte de $\mathcal{D}_{eval}$ para así obtener los conjuntos resultado de las alertas correspondientes. Por último, mediante las definiciones vistas en 5.1 podremos calcular las métricas de precision y recall a partir de la totalidad de los resultados.

[1] `https://github.com/github/codeql/commits/main/javascript/ql/lib/semmle/javascript/security/dataflow`

[2] `https://github.com/github/codeql/commit/b1957623c1e19e0fa5399d3790b5baa67f152328`

Overfitting es un fenómeno que ocurre cuando mediante una técnica de aprendizaje, se entrena un modelo con datos que luego serán usados para evaluarla su potencial de aprendizaje. Al ocurrir esto, el modelo tiende a sobre-ajustarse a los datos de entrada, produciendo inferencias que a pesar de dar valores altos en las métricas de aprendizaje, generalizan mal a datos no conocidos durante el entrenamiento.

El algoritmo 3 muestra en detalle los pasos que el método de evaluación lleva a cabo. La primera etapa consiste en utilizar la técnica para inferir nuevos sinks, y con los diferentes conjuntos de consultas, evaluar taint analysis para así obtener las alertas mediante las cuales calcularemos las métricas de precision y recall.

- Línea 1: Se evalúa la técnica presentada en este trabajo.
- Líneas 2-9: Para cada programa en el conjunto de evaluación, calculamos, mediante el motor de análisis de CodeQL, taint analysis, utilizando las especificaciones definidas por el conjunto de consultas correspondiente al primer argumento de la llamada a EVALUAR. Se realizan evaluaciones para cada una de las especificaciones involucradas en la metodología: el conjunto inicial ($\mathcal{Q}_W$), la composición de las iniciales con los resultados del proceso de inferencia ($\mathcal{Q}_I$), y la versión más reciente que es usada como referencia y verdad absoluta ($\mathcal{Q}_O$). Es conveniente resaltar que, aunque las especificaciones definidas en $\mathcal{Q}_I$ incluyan las iniciales, estas deben ser evaluadas por separado, ya que una vez realizado el análisis, no es posible determinar si una alerta surgió utilizando un sink inferido o no.

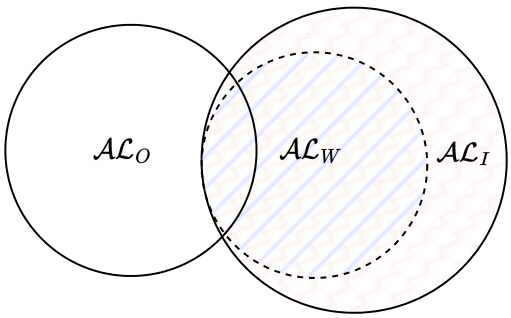

Fig. 5.1: Diagrama de Venn con los conjuntos resultados de alertas, de las tres evaluaciones en el algoritmo 3.

Luego, la segunda etapa consiste en, una vez obtenidos los conjuntos resultado de alertas para cada conjunto de especificaciones, calcular las métricas obtenidas. Para ello, se debe categorizar cada resultado dentro de las clases de error o acierto definidos en la tabla 5.1. Como solamente nos interesa clasificar las alertas que son atribuidas a los sinks descubiertos por nuestra técnica, debemos descartar las alertas detectadas por medio de las especificaciones iniciales. Por esto, las líneas 10-11 definen dos conjuntos que son los que tendremos en cuenta para clasificar en la matriz de confusión. En la figura 5.1 podemos ver como $\mathcal{AL}_W \subseteq \mathcal{AL}_I$, y cómo es la interacción entre todos los conjuntos. La clasificación de los resultados dentro de las categorías de error o acierto se lleva a cabo de la siguiente manera:

- Línea 12: Una alerta es considerada como TP, o verdadero positivo, si esta es descubierta tanto por el analizador de referencia como por las especificaciones bajo escrutinio. Como en nuestro caso, $\mathcal{Q}_O$ son las primeras y $\mathcal{Q}_I$ la segundas, una alerta a sería considerada como cierta si $a \in \mathcal{AL}_O \wedge a \in \mathcal{AL}_B$. De esta manera, podemos descubrir la cantidad de casos de esta categoría mediante la intersección de ambos conjuntos.

- Línea 13: Una alerta se categoriza como FP cuando la misma es descubierta por las EI, pero no por el ADR. Esto podemos definirlo en nuestro caso como las alertas que forman parte de $\mathcal{AL}_B$, pero no $\mathcal{AL}_O$. Por consiguiente, tomamos como falsos positivos la cantidad de elementos en la diferencia de ambos conjuntos.

- Línea 14: Caso análogo al anterior, pero donde una alerta es considerada como FN cuando es descubierta por el ADR, y no por las EI.

Por último, habiendo categorizado todas las alarmas en las clases de resultado, podemos calcular las métricas de precision y recall.

Algoritmo 3 Método de evaluación

Requiere $\mathcal{D}_{train}$, $\mathcal{D}_{eval}$ conjuntos de programas
Requiere $\mathcal{Q}_W$, $\mathcal{Q}_I$, $\mathcal{Q}_O$ conjuntos de consultas
1: $\mathcal{Q}_I \leftarrow$ JELDON$(\mathcal{D}_{train}, \mathcal{Q}_W)$
2: $\mathcal{AL}_W \leftarrow \emptyset$
3: $\mathcal{AL}_I \leftarrow \emptyset$
4: $\mathcal{AL}_O \leftarrow \emptyset$
5: **for** programa $\rho \in \mathcal{D}_{eval}$ **do**
6: $\quad \mathcal{AL}_W \leftarrow^{\cup}$ EVALUAR$(\mathcal{Q}_W, \rho)$
7: $\quad \mathcal{AL}_I \leftarrow^{\cup}$ EVALUAR$(\mathcal{Q}_I, \rho)$
8: $\quad \mathcal{AL}_O \leftarrow^{\cup}$ EVALUAR$(\mathcal{Q}_O, \rho)$
9: **end for**
10: $\mathcal{AL}_B \leftarrow \mathcal{AL}_I \setminus \mathcal{AL}_W$
11: $\mathcal{AL}_O \leftarrow \mathcal{AL}_\mathcal{O} \setminus \mathcal{AL}_W$
12: $TP \leftarrow |\mathcal{AL}_O \cap \mathcal{AL}_B|$
13: $FP \leftarrow |\mathcal{AL}_B \setminus \mathcal{AL}_O|$
14: $FN \leftarrow |\mathcal{AL}_O \setminus \mathcal{AL}_B|$
15: $precision \leftarrow \frac{TP}{TP+FP}$
16: $recall \leftarrow \frac{TP}{TP+FN}$
17: **return** $precision, recall$

5.2. Conjuntos de consultas y de programas

Para poder evaluar el desempeño de la técnica, necesitamos primero obtener un conjunto de especificaciones inferidas, partiendo de un conjunto de programas y otro de consultas iniciales. Como se mencionó en la introducción del capítulo 4, los conjuntos de consultas en CodeQL de los cuales partimos, y pretendemos mejorar mediante la técnica, modelan una vulnerabilidad en particular. Para ello, utilizaremos tres tipos de vulnerabilidades

del ranking de vulnerabilidades *MITRE CWE Top 25* [12]: NoSQL Injection[3], XSS[4] y TaintedPath[5]. NoSQL Injection es parte de una clase de vulnerabilidades conocida como ataques de inyección. En estas, un usuario logra introducir en un programa un dato construido de una manera específica, de manera de poder controlar la consulta a una base de datos NoSQL, y logrando que la misma realice un comportamiento no esperado por el programador. XSS, o *cross-site scripting*, representa un tipo de vulnerabilidades donde un fragmento de código JavaScript malicioso puede ser introducido en el DOM[6], y por consiguiente evaluados. Por último, TaintedPath es un tipo de vulnerabilidad donde un atacante puede controlar de manera no esperada la ruta de donde un archivo es escrito, o leído.

En este capítulo, en la experimentación llevada a cabo, el conjunto inicial de consultas fue elaborado partiendo de una versión más antigua de las bibliotecas de consultas de CodeQL, cuyo último commit fue en Septiembre de 2021. A esta, se le revirtieron una serie de cambios que agregaban sinks para los diferentes tipos de vulnerabilidades que estudiamos. Entre ellos se encuentran:

- XSS: Sinks agregados que modelan el comportamiento de ciertas APIs como el antes mencionado DOM, Angular y jQuery, además de otro cambios generales.
- TaintedPath: Cambios en los modelos de la API que interactúa con el sistema de archivos.
- NoSQL: Cambios generales.
- Cambios revertidos que afectaban diversas funcionalidades modeladas como promesas, modelos de la biblioteca de APIs REST Express, comportamientos nativos de Node.js, entre otros.

Esta versión manipulada de las bibliotecas se encuentra disponible en GitHub[7].

Además del conjunto de consultas iniciales, como fue mencionado en la sección anterior, utilizamos una versión más reciente de las bibliotecas de CodeQL junto con el motor de análisis como analizador de referencia. Para esto, utilizamos una versión[8] de las mismas cuyo último commit es de Septiembre 2022, aproximadamente un año más reciente que la "versión antigua". Para dar una idea de la diferencia entre ambas versiones de las bibliotecas, la versión usada como especificaciones iniciales, y la de referencia, existen 2108 commits de una a la otra. Es decir, más de dos mil cambios dedicados de una forma y otra a mejorar la "calidad" de alertas detectada por CodeQL.

Además de las especificaciones iniciales y las de referencia, necesitamos un conjunto de programas sobre el cual aplicar nuestra técnica de aprendizaje. Como cada conjunto de consultas predica sobre una vulnerabilidad en particular, es lógico que el conjunto de consultas elegido para cada una resulte de "interés" en cada contexto. Para obtener los resultados empíricos de InspectJS [4] se usó un enfoque similar, dividiendo los conjuntos

[3] `https://learn.snyk.io/lesson/nosql-injection-attack/`
[4] `https://cwe.mitre.org/data/definitions/79.html`
[5] `https://cwe.mitre.org/data/definitions/22.html`
[6] DOM, o *document object model*, es una API que los navegadores presentan para poder controlar el sitio web mostrado al usuario.
[7] Fork de CodeQL, donde está publicada la versión alterada usada como especificaciones iniciales.
[8] Commit usado como bibliotecas para el analizador de referencia.

de datos por tipo de vulnerabilidad. En esta publicación, se partió de un conjunto de alrededor de doscientos mil programas open source disponibles en GitHub [9], filtrándolos y categorizándolos, buscando programas que tuvieran al menos una alerta de cada vulnerabilidad antes mencionada. Del total, se seleccionó un conjunto de 562 programas para Tainted Path, 2834 para XSS, y 833 para NoSQL Injection.

Para reducir el tamaño de algunos de los conjuntos de programas, considerando que la experimentación debía ser realizada en un tiempo pautado, se realizó una curaduría sobre cada conjunto. La tabla 5.2 muestra el tamaño de cada conjunto de programas luego de este proceso. La curaduría realizada sobre los conjuntos originales se realizó teniendo en cuenta los siguientes factores:

- Disponibilidad del código fuente de cada programa. Desde la publicación de [4], es posible que algunos repositorios hayan sido eliminados, o cerrados al público.
- En caso de no tener una versión particular del código fuente del programa, se utilizó el commit más reciente del repositorio.
- Para disminuir el tamaño de cada conjunto, se tomó como métrica de filtrado la proporción de código JavaScript sobre el total de líneas en el código fuente. Realizando un histograma de las proporciones de código JavaScript sobre un conjunto en particular, se tomó un valor de corte para conservar en el conjunto seleccionado al menos la mitad de los programas originales.

Vulnerabilidad	$\lvert\mathcal{D}\rvert$
NoSQL Injection	257
Tainted Path	336
XSS	90

Tab. 5.2: Tamaño de los conjuntos de programas utilizados para la evaluación de la técnica.

5.3. Experimentación

Como fue mencionado en la introducción de este capítulo, la principal pregunta de investigación que queremos responder es "si podemos utilizar la técnica presentada en Seldon para descubrir nuevas especificaciones de taint analysis en otros lenguajes". En el capítulo 4 describimos cómo sería tal implementación para JavaScript. En la sección 5.1 de este capítulo exponemos una metodología que nos permite evaluar las especificaciones aprendidas mediante la técnica, sin necesidad de corroborar manualmente si cada especificación inferida es o no verdadera.

La experimentación para responder dicha pregunta sigue el esquema presentado en el algoritmo 3 para cada una de las vulnerabilidades mencionadas en la sección 5.2.

La metodología de evaluación recibe dos conjuntos diferentes de programas, uno usado para el proceso de inferencia de sinks, y otro para evaluar la utilidad de las especificaciones descubiertas por medio del análisis estático de programas. Partiendo de un único conjunto de datos, existen diversas técnicas para particionarlos en dos o más sub-conjuntos que

[9] Realmente se utilizó un servicio adquirido por GitHub, conocido como LGTM. El mismo permitía ejecutar consultas de CodeQL sobre cualquier programa disponible de manera open-source.

serán usados para "entrenar" una técnica de aprendizaje (inferencia en nuestro caso), y luego evaluarla. Como veremos más adelante en el análisis de cada uno de los resultados, las especificaciones obtenidas, y las alertas detectadas en cada uno de los programas pueden variar de manera significativa. Un programa puede tener muchos términos de una API que se encuentra modelada entre las especificaciones iniciales, pero otros, ninguno. Por este motivo, y para en un experimento en particular, repartir el impacto de esta diferencia entre sucesivas corridas del proceso de inferencia y evaluación, utilizamos una técnica conocida como K-Fold Cross Validation [8]. La misma, aplicada a nuestro dominio, consiste en partir un conjunto de programas $\mathcal{D}$ en n fragmentos disjuntos de igual tamaño. Luego, el experimento bajo estudio que consta en una primera fase de entrenamiento, y otra de evaluación, ambas requiriendo un conjunto de programas $\mathcal{D}_{train}$ y $\mathcal{D}_{eval}$ respectivamente, será repetida n veces. Cada una de estas repeticiones es conocida como un **fold**. Dentro de cada uno de estos folds o iteraciones (supongamos el i-ésimo), los conjuntos de entrenamiento y evaluación estarán conformados de la siguiente manera:

$$\begin{aligned} \mathcal{D}_{train} &= \cup\{\mathcal{D}_j | 0 \leq j < n \wedge j \neq i\} \\ \mathcal{D}_{eval} &= \mathcal{D}_i \end{aligned} \tag{5.2}$$

Si tomamos, por ejemplo n con un valor de 5, la figura 5.2 muestra los cinco fragmentos del conjunto original, y los dos primeros folds.

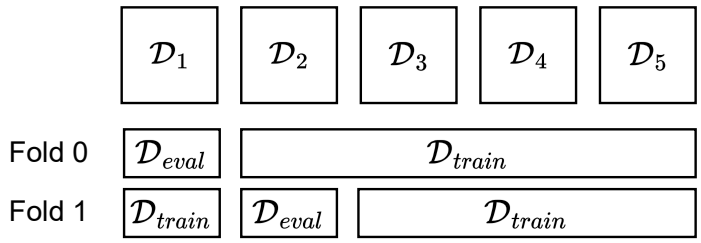

Fig. 5.2: Ejemplo de K-fold Cross Validation para $n = 5$, mostrando los dos primero folds.

Para la experimentación realizada en este trabajo, partiendo de cada uno de los conjuntos de programas presentados en la sección anterior, se realizó el esquema de experimentación presentado en el algoritmo 3 usando 5-fold Cross Validation, y tomando como resultado final la media de cada medición. Como se verá en los resultados, se logra así atenuar las diferencias que pudieran existir entre cada conjunto diferente de programas. La tabla 5.3 muestra la cantidad de elementos en los conjuntos de evaluación y entrenamiento, usados en cada fold del experimento.

Vulnerablidad	$\lvert\mathcal{D}_{train}\rvert$	$\lvert\mathcal{D}_{eval}\rvert$
NoSQLInjection	205	52
TaintedPath	268	68
XSS	72	18

Tab. 5.3: Tamaño de los conjuntos de entrenamiento y evaluación en cada *fold*, usando 5-fold Cross Validation.

La siguiente sección explicará de qué forma fueron elegidos los hiper-parámetros esperados por la técnica. Luego, se presentarán los resultados obtenidos para cada tipo de

vulnerabilidad estudiado. Se presentarán los resultados de las dos métricas principales, precision y recall, así como una serie de mediciones adicionales. Se realizará luego una comparación entre los resultados obtenidos en Seldon y este trabajo, respondiendo a la segunda pregunta de investigación. Por último, se comentará un caso donde algunos resultados de la técnica fueron aportados a las bibliotecas de CodeQL, contestando a la tercera y última pregunta.

5.3.1. Elección hiper-parámetros

A lo largo del capítulo 4 se presentaron una serie de hiper-parámetros que permiten alterar el comportamiento de la técnica en cierta manera. Parte de ellos proviene de la formulación central del problema de inferencia, C y λ. El primero, C, es una constante usada en la formulación de las restricciones de flujo del modelo, y el segundo, λ, es un parámetro de regularización. En Seldon [3] se presentan una serie de criterios mediante los cuales se elige $C = 0{,}75$ y $\lambda = 0{,}1$. Para la experimentación de este trabajo decidimos utilizar los mismos valores sugeridos en la publicación original. Esta decisión fue tomada luego de experimentar manualmente con pequeñas variaciones en ambos, y notar un comportamiento similar al descripto en Seldon.

Además, en Jeldon se definen dos nuevos parámetros: MaxReprDepth, que controla la longitud máxima permitida en las representaciones independientes generadas para un término; y MinScoreSink, parámetro que actúa como filtro para sinks inferidos con un bajo puntaje. Para el primero, se probaron diferentes valores entre 3 y 5. Como lo indica su definición, cuanto mayor sea su valor, mayor será la cantidad de diferentes representaciones independientes que podrá tener un término. Esto impacta directamente en la cantidad de variables que conforman los modelos de LP, y por consiguiente, el tiempo de entrenamiento. Por otro lado, disminuir este valor conlleva menor cantidad de representaciones, y menos específicas. Para lograr un balance entre estos dos extremos, optamos por MaxReprDepth = 4. Por último, MinScoreSink actúa como un valor de corte que nos permite elegir cuáles inferencias considerar y cuáles descartar durante la evaluación de la técnica. A través de múltiples ejecuciones sobre las diferentes clases de vulnerabilidad estudiadas, notamos que la mayor cantidad de inferencias son asignadas a un puntaje apenas por encima de 0.2, mientras que pocas quedan por debajo de 0.1. Como, teniendo en cuenta el enunciado de la RQ1, buscamos maximizar la cantidad de especificaciones inferidas, optamos por elegir un valor de corte lo más bajo posible, tomando en cuenta una mayor cantidad de sinks inferidos durante la evaluación, pero impactando fuertemente los valores de precisión, ya que se mantendrían especificaciones con una puntuación muy baja.

5.3.2. Resultados

Los resultados se encuentran divididos entre dos tablas. En primer lugar, la tabla 5.4 muestra los resultados de las métricas principales que buscamos evaluar: precision y recall. Luego, la tabla 5.5 muestra una serie de mediciones secundarias que nos permiten entender los resultados de manera menos abstracta, inspirados en la evaluación realizada en [4]. Estas mediciones son:

- AAR: Alertas a recuperar, cantidad de alertas marcadas por $\mathcal{Q}_O$, pero no por $\mathcal{Q}_W$. La idea de que las alertas puedan ser recuperadas o no, hace referencia a que la técnica

intenta "recuperar" o "aprender" alertas existentes en una versión más reciente de las bibliotecas, o que fueron removidas de manera manual de una versión inicial.

- Recuperadas: Cantidad de alertas que forman parte de las AAR, y fueron recuperadas por el proceso de inferencia, es decir, que fueron encontradas usando $\mathcal{Q}_B$ como especificaciones. Se corresponde con la cantidad de alertas catalogadas como verdaderos positivos.
- Espurias: Cantidad de alertas marcadas por $\mathcal{Q}_I$, pero no por $\mathcal{Q}_O$. Se corresponden con las alertas categorizadas como falsos positivos.
- Programas con AAR: Cantidad de programas con al menos un alerta a recuperar. Esta medición puede darnos una idea de cuántos programas fueron relevantes en la evaluación.
- $\overline{AAR}$: Cantidad media de alertas a recuperar, por programa.
- $|\mathcal{A}_I|$: Cantidad media de sinks inferidos.

Podemos así señalar que, en respuesta a la **RQ1** presentada en la introducción de este capítulo, podemos decir que JELDON puede inferir nuevos sinks para el lenguaje JavaScript. La principal métrica para evaluar si la técnica tiene la capacidad de inferir especificaciones es **recall**, ya que de acuerdo con su definición, mide la proporción de alertas descubiertas mediante las especificaciones inferidas, sobre el total que podrían haber sido. El mayor valor para esta métrica es obtenido usando las consultas de NoSQL Injection como especificaciones iniciales, obteniendo un valor de recall superior a 81 %. Más aún, este porcentaje corresponde a una media de 121 alertas descubiertas por el analizador de referencia, a través de 52 programas en los diferentes conjuntos de evaluación, lo cual indica que el valor es una proporción no menor sobre una cantidad total de alertas a recuperar. Como mencionamos en la sección 5.3.1, el principal objetivo buscado durante la evaluación fue maximizar la capacidad de la técnica para descubrir nuevas alertas, es decir, maximizar recall. Por este motivo, los resultados presentan una cantidad alta de falsos positivos, reflejada en los valores de precision obtenidos, que llegan solamente al 20 %. Para ejemplificar, este valor indica que solamente una de cada cinco alertas descubiertas por las especificaciones inferidas es realmente cierta, mientras que las cuatro restantes son falsos positivos. Siendo el valor de recall obtenido significativo, existen diversas formas de mejorar precision que serán mencionadas en la sección 6.2, pero una estrategia notable es la presentada en InspectJS [4] que consiste en filtrar especificaciones inferidas según su similitud sintáctica a otras existentes.

Diversas formas de mejorar precision, una vez realizada la inferencia (es decir, no cambios en el proceso de inferencia). Eso hace inspectJS

En cuanto a los otros casos de evaluación, Tainted Path y XSS, los valores de recall obtenidos son menores. En el caso de las consultas iniciales que modelan Tainted Path, estas alcanzaron un valor de recall que supera el 66 %, siendo el valor de precisión aún más afectado que los resulados de NoSQL, alcanzando solamente 5.3 %. Resulta interesante que, a pesar de que el conjunto de programas utilizado es de mayor tamaño, la cantidad de alertas a recuperar en NoSQL resulta ser 4 veces mayor comparado con Tainted Path. Múltiples factores pueden ser los causantes de este comportamiento: baja frecuencia de ocurrencia en flujos considerados como candidatos para esta clase de vulnerabilidad, lo cual sería un causante directo en la menor cantidad de alertas en general entre una y otra clase de consultas; similitud entre las especificaciones iniciales y de referencia, caso que

conllevaría a que $\mathcal{AL}_W$ y $\mathcal{AL}_O$ sean muy semejantes y, por consiguiente, menor cantidad de alertas a recuperar; entre otras. Todas estas causas podrían dar lugar a que la cantidad de alertas que son consideradas en las métricas de evaluación fuera reducida, y por ende, la evaluación más sensible a pequeñas variaciones.

Las especificaciones iniciales que modelan XSS dieron resultados no satisfactorios. Viendo los resultados de la tabla 5.5 se puede apreciar la baja relevancia en los conjuntos de evaluación, lo cual vuelve la prueba extremadamente sensible debido a la poca cantidad de alertas encontradas. Como el tamaño del conjunto de programas es relativamente chico comparado al resto (un tercio del segundo mayor en tamaño, con 90 programas), aplicar 5-Fold Cross Validation hace que el conjunto de evaluación sea aún de menor tamaño, resultando en un conjunto con solamente 18 programas. Suponiendo esto como causa de los resultados negativos, se evaluaron otras técnicas de partición de datos como Shuffle Split[10] usando una mitad para evaluación y otra para entrenamiento, y Validation Set[11]. Todas las alternativas dieron resultados similares.

Este último caso, XSS, resulta particularmente interesante para analizar más en detalle el significado de recall, en el contexto de esta metodología. Un resultado de recall con un valor de cero, o cercano a cero, no significa que la técnica no produjo nuevas especificaciones, ya que como podemos apreciar en la tabla 5.5 el promedio de sinks inferidos entre las diferentes pruebas de XSS es de 187, más que NoSQL Injection, caso donde se obtuvieron las mejores mediciones. Para darle sentido a este resultado, debemos tener en cuenta que no solo estamos comparando dos conjuntos de consultas diferentes, sino que el conjunto usado como referencia es una versión más reciente (y por consiguiente mejor) que las especificaciones iniciales. Esto sería diferente si para nuestra metodología usáramos como referencia un conjunto $\mathcal{Q}$, y una versión "empeorada" de este como especificaciones iniciales, como lo fue para el caso de $\mathcal{Q}_W$. Aquí, una alerta recuperada significaría que la técnica pudo inferir conocimiento que fue removido manualmente de las especificaciones iniciales. Al contrario, en nuestro caso, una alerta recuperada significa que la técnica pudo inferir especificaciones que no solo no eran parte de las especificaciones iniciales, sino que surgen de ser detectadas por un analizador más moderno y potente que ellas. Esto hace que los valores obtenidos de recall para los casos exitosos, NoSQL Injection y Tainted Path, sean aún más valiosos.

Una posible causa de que en XSS no se hayan observado alertas recuperadas es que la versión de las consultas usada como referencia contenga cambios más allá de las especificaciones, como por ejemplo, cambios en cómo se propaga el taint a lo largo del programa, o en algunos de los algoritmos que forman el analizador (points-to, aliasing, entre otros); o que, de tener cambios en las especificaciones, los conjuntos de entrenamiento no tengan un programa que los ejemplifique, en cuyo caso, la técnica no podría haberlos inferido.

Por último, en los resultados obtenidos para NoSQL Injection y Tainted Path, encontramos una característica muy interesante como lo es la estabilidad de los mismos. La figura 5.3 muestra cómo se relacionan los valores de precision y recall, para cada uno de los folds de cada experimento. Podemos ver que para los dos tipos de especificaciones con mejores resultados, ambas métricas presentan una baja dispersión a través de cada

[10] Shuffle Split consiste en partir un conjunto de N elementos en dos cortes, uno de evaluación con T elementos, y otro de entrenamiento con $N - T$, de manera aleatorio y sucesivas veces. Al ser aleatoria la selección, no hace falta partir el conjunto inicial en M partes para conformar cada fold, sino que puede haber solapamiento entre diversas iteraciones.

[11] Validation Set [8] consiste en partir el conjunto entero en dos partes, una de entrenamiento y otra de evaluación, con diferentes tamaños.

	Precision	**Recall**
Tainted Path	0.0527	0.6693
NoSQL Injection	0.1454	0.8192
XSS	0.2000	0.0167

Tab. 5.4: Resultados de precision y recall, promediados entre los resultados parciales de cada fold de Cross Validation.

	AAR	**Recuperadas**	**Espurias**	**Programas con AAR**	$\overline{AAR}$	$\lvert\mathcal{A}_I\rvert$
Tainted Path	33	22	488	13	2.4254	444
NoSQL Injection	121	94	547	27	4.2470	122
XSS	11	0	0	3	2.8300	187

Tab. 5.5: Resultados secundarios, promediados entre los resultados parciales de cada fold de Cross Validation, para cada conjunto de especificaciones iniciales estudiado.

fold, con una desviación standard de $\sigma_{precision} = 0{,}0336$ y $\sigma_{recall} = 0{,}1650$ para NoSQL Injection, caso que se ve afectado por el valor bajo de recall obtenido para uno de los folds; y $\sigma_{precision} = 0{,}0{,}0254$ y $\sigma_{recall} = 0{,}0438$ para Tainted Path. Esto nos indica que más allá de los programas utilizados tanto para evaluación como entrenamiento, la técnica puede inferir nuevos sinks, manteniendo una proporción relativamente constante de falsos positivos. Para estos casos donde no notamos una fuerte dependencia de los resultados obtenidos con los datos de entrada, se podrían explorar mejoras como una mejor selección de hiper-parámetros, o cambios en la técnica misma. Algunos de estos serán mencionados en la sección 6.2.

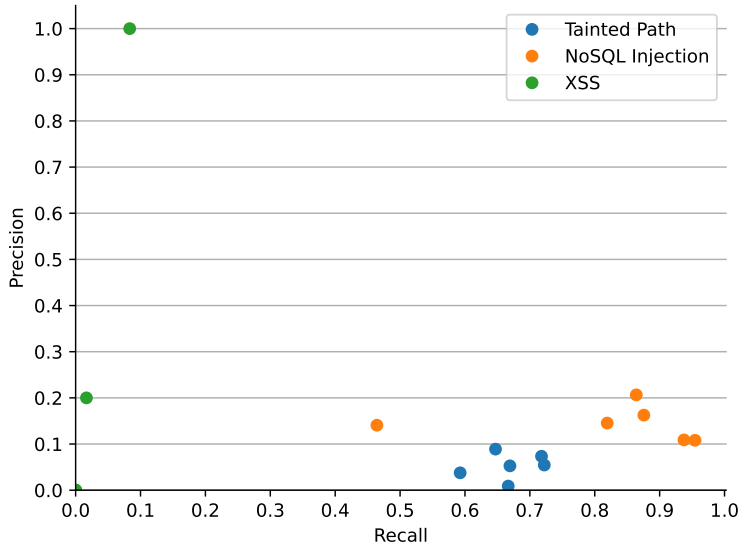

Fig. 5.3: Gráfico de precision-recall de los resultados parciales de cada fold, para cada conjunto de especificaciones iniciales.

5.3.3. Comparación contra Seldon

En [3] se plantean dos preguntas de investigación cuyos resultados son comparables con los presentados en este trabajo, pero para ello, hace falta tener en cuenta que las metodologías de evaluación difieren fuertemente.

En primer lugar, la segunda pregunta de investigación del trabajo de Seldon pretende determinar cuán precisas son las especificaciones inferidas por la técnica. En su caso, la evaluación consistió en tomar una muestra aleatoria de 50 sources, sanitizers y sinks cuyo rol fue inferido (es decir, 150 especificaciones inferidas), y manualmente corroborar la validez de las mismas. Mediante la cantidad de resultados falsos y verdaderos obtenidos de esta revisión manual de especificaciones, y extrapolando al total de especificaciones inferidas, los autores publican que obtuvieron, de manera estimada, una precision de 56 % para sinks, y 66 % en promedio para todos los roles. Es importante notar que, por ejemplo, para el caso de sinks, la muestra aleatoria de 50 casos representa solamente un 5.77 % del total de inferencias, mientras que para los tres roles un 2,17 %.

En segundo lugar, la cuarta pregunta de investigación plantea la evaluación de la relevancia de las especificaciones inferidas para descubrir nuevas alertas, directamente relacionada con la **RQ1** en este trabajo. A pesar de esto, los resultados no son comparables, ya que su metodología no evaluaba si las alertas detectadas fueron realmente ciertas, sino solo qué cantidad de nuevas alertas eran reportadas por las especificaciones inferidas, en comparación con las iniciales. Más allá de no poder comparar ambos resultados, los autores reportan que, analizando el mismo conjunto de programas, sus especificaciones iniciales encontraron solamente 662 alertas, mientras que las inferidas 21318, 32 veces o un orden de magnitud más. Esta medición no considerada al momento de realizar la experimentación en esta tesis, resultaría de sumo interés para poder comparar cuantitativamente ambas técnicas a pesar de que analizan diferentes lenguajes.

5.4. Especificaciones contribuidas a CodeQL

Aparte de los resultados cuantitativos presentados en la sección 5.3, que responden nuestra pregunta que valida que Jeldon puede ser usado para descubrir nuevos sinks, una segunda validación de esta declaración son los siguientes aportes a las bibliotecas de CodeQL, fruto de la técnica presentada en este trabajo en conjunto con [4]. Esto a su vez nos permite responder afirmativamente a la tercera pregunta de investigación, **RQ3**.

- Nuevos sinks para una biblioteca de NoSQL[12]
- Modelo de llamadas a función concatenadas para SQLite [13]
- Mejora en las bibliotecas que modelan la API de DOM [14]

5.5. Cierre

En este capítulo se presentó un marco teórico y una metodología para evaluar cuantitativamente la capacidad de Jeldon para inferir nuevos sinks. La metodología presentada construye un oráculo que permite llevar a cabo la evaluación sin la necesidad de un experto

[12] `https://github.com/github/codeql/pull/4753`
[13] `https://github.com/github/codeql/pull/5860`
[14] `https://github.com/github/codeql/pull/5262`

en los tipos de vulnerabilidades estudiadas, pudiendo así realizarla de manera no supervisada. Se presentó un análisis de los resultados obtenidos, que permitió confirmar que la técnica puede inferir nuevos sinks. Luego, se expusieron las diferencias con la metodología de evaluación usada en Seldon, trabajo del cual surge la técnica desarrollada en esta tesis. Por último, se hizo una mención a una serie de aportes realizados a las especificaciones que forman parte de las bibliotecas de CodeQL.

6. CONCLUSIÓN

En esta tesis se introdujo una técnica de análisis estático de código, taint analysis, que es de particular interés para poder detectar de manera temprana vulnerabilidades que son muy comunes en aplicaciones web, como son los ataques de inyección. En particular, se presentó este marco para el lenguaje de programación JavaScript porque, como fue mencionado, es uno de los lenguajes de programación más usados en la industria, y presenta propiedades que hacen el análisis estático más desafiante.

Se presentó una técnica de inferencia de especificaciones de taint analysis, elemento de suma importancia para este tipo de análisis, que define qué comportamiento busca detectar el analizador. La técnica, inspirada en el trabajo realizado en Seldon [3], parte de la idea central de plantear el problema de inferencia como uno de optimización lineal. En esta tesis, se introduce el trabajo original, formalizando las ideas centrales para luego presentar nuestra implementación, que utiliza como componente central el motor de análisis estático CodeQL. Nuestra técnica presenta ciertas innovaciones en comparación con la de Seldon, como una representación de términos de programa que permite generalizar un fragmento de código, para poder detectar casos similares en otros programas, y una nueva definición del grafo subyacente, utilizado para construir el modelo de programación lineal, que puede capturar patrones complejos de programación como son las callbacks. Nuestra implementación permite, dado un conjunto de programas, llevar a cabo de manera automática todos las etapas de la técnica, obteniendo los resultados en un formato que fácilmente puede ser usado para mejorar el funcionamiento de taint analysis en CodeQL.

Por último, se presentó una nueva metodología de evaluación, que permite de manera no-supervisada evaluar cuantitativamente la capacidad de la técnica para inferir nuevas especificaciones. La idea central de la metodología consiste en utilizar versiones antiguas de las especificaciones, mejorarlas con nuestra técnica, y comparar los resultados inferidos con versiones más recientes, usando como criterio de comparación la cantidad de alertas descubiertas por medio del análisis estudiado. Por medio de esta metodología, pudimos responder de manera afirmativa a la pregunta de si nuestra técnica es capaz inferir nuevos sinks, y corroborar su utilidad en la práctica.

6.1. Trabajos relacionados

Inferencia de especificaciones Existen múltiples trabajos que proponen soluciones para el problema de inferencia de especificaciones para taint analysis. Los trabajos más relevantes para esta tesis son Seldon [3], que sirvió de inspiración para la Jeldon , y es estudiado en detalle en el capítulo 3; e InspectJS [4], publicación donde la técnica desarrollada en esta tesis fue extendida con una metodología de curaduría de las especificaciones inferidas. Seldon toma algunas ideas y utiliza como punto de comparación a Merlin [11], donde se estudia el problema de inferencia para el lenguaje C#. Merlin y otros trabajos estudian el problema de inferencia para lenguajes de programación con un sistema de tipos estático, lo cual facilita su estudio mediante análisis estático. Otros estudian el problema mediante análisis dinámico de código, como TASER [17], que utiliza información capturada en tiempo de ejecución para construir nuevas especificaciones.

Taint analysis Muchos trabajos han estudiado la problemática de análisis estático de código JavaScript, como TAJS [9] donde se presenta una herramienta para inferir información de tipos mediante abstract interpretation. Esta herramienta fue mejorada por una serie de publicaciones realizadas en parte por Aarhus University[1], que estudian diversas fectas del análisis de código en JavaScript.

Otros trabajos, debido a las complejidades que presenta JavaScript para la versión estática, se enfocan en estudiar las aplicaciones de análisis dinámico de código, como NodeProf [18] y JSFlow [7], donde se implementa una versión del runtime de JavaScript que tienen la instrumentación necesaria para llevar a cabo los análisis. El primero, NodeProf, es utilizado en las bases de TASER [17], trabajo donde se presenta una versión anterior de la representación programa independiente usada en Jeldon .

6.2. Futuras líneas de investigación

La técnica desarrollada en esta tesis vislumbra futuros caminos de investigación, tanto para mejorar su capacidad de aprendizaje, ampliar la clase de resultados que produce y aplicaciones en la práctica. El siguiente listado presenta diversas ramas de trabajos a futuro:

Inferencia de sources y sanitizers
Extender la técnica y evaluar su performance para inferir sources y sanitizers.

Exploración de hiper-parámetros
Realizar una exploración del espacio generado por los hiper-parámetros de nuestra técnica. Alguna metodología como grid-search podría ser utilizada para acotar y discretizar el espacio de búsqueda.

Boostear últimas versiones de las especificaciones
Aplicar la técnica usando como especificaciones iniciales versiones recientes de las bibliotecas de CodeQL, y analizar manualmente los sinks descubiertos. Esta caracterización nos permitiría evaluar a la técnica infiriendo nuevo conocimiento, y de ser este útil, contribuir al campo específico en la comunidad.

Mejorar los tiempos de entrenamiento
Evaluar mejoras de performance de la etapa de entrenamiento, como re-usar resultados parciales, o paralelismo.

Experimentar con diferentes agregaciones de las inferencias individuales
Nuestra técnica calcula la media de los puntajes asignados a una misma representación independiente, a través de diferentes programas. Existen diferentes agregaciones como medias ponderadas, máximos, y otras, que podrían afectar los valores de recall y precision obtenidos.

[1] `https://www.brics.dk/TAJS/`

7. APÉNDICE: EJEMPLOS DE CONSULTAS QL

Para ejemplificar, la figura 7.1 muestra un extracto de la biblioteca de QL que nos permite identificar términos con un rol de source, sink o sanitizer para vulnerabilidades del tipo *SQLInjection*. Este ejemplo define 3 clases principales: `Source`, `Sink` y `Sanitizer`. En QL una clase debe ser interpretada como un conjunto de instancias y no como el "prototipo de una", por lo cual consultar si un término t es un source, no es otra cosa que saber si t forma parte del conjunto denotado por la clase `Source`. Más aún, la herencia entre clases se interpreta como inclusión entre conjuntos. Es por ello que en una biblioteca de consultas especificada en QL es común definir una clase de datos abstracta, `Source` por ejemplo, la cual es extendida definiendo sub-clases como `RemoteFlowSourceAsSource`. Esto permite a quien utiliza esta biblioteca preguntar si un término es un source solamente mediante la clase abstracta, `t instanceof Source`.

Por otro lado, el fragmento 7.2 muestra la implementación en QL de uno de los predicados usados para caracterizar cuándo consideramos en nuestra técnica que hay flujo entre dos términos. Resulta interesante que el predicado mezcla conceptos a diferentes niveles de abstracción como dominancia entre bloques dentro del Control Flow Graph (CFG), o relaciones entre variables a nivel de Static Single-Assignment[1].

[1] Static single-assingment, o SSA, es una representación intermedia dónde se require que cada variable sea asignada a lo sumo una vez.

```
import javascript

module SqlInjection {
  // Definición abstracta de un source
  abstract class Source extends DataFlow::Node { }

  // Definición abstracta de un sink
  abstract class Sink extends DataFlow::Node { }

  // Definición abstracta de un sanitizer
  abstract class Sanitizer extends DataFlow::Node { }

  // A source of remote user input, considered as a flow source
  // for string based query injection.
  class RemoteFlowSourceAsSource extends Source {
    RemoteFlowSourceAsSource() { this instanceof RemoteFlowSource }
  }

  // An SQL expression passed to an API call that executes SQL.
  class SqlInjectionExprSink extends Sink, DataFlow::ValueNode {
    override SQL::SqlString astNode;
  }

  // An expression that sanitizes a value for the purposes of string
  // based query injection.
  class SanitizerExpr extends Sanitizer, DataFlow::ValueNode {
    SanitizerExpr() { astNode = any(SQL::SqlSanitizer ss).getOutput() }
  }

  // An GraphQL expression passed to an API call that executes GraphQL.
  class GraphqlInjectionSink extends Sink {
    GraphqlInjectionSink() { this instanceof GraphQL::GraphQLString }
  }
}
```

Fig. 7.1: Ejemplo de una biblioteca que caracteriza sources, sinks y sanitizers en vulnerabilidades del tipo SQLInjection. Disponibles en Github.

```
private predicate guard(DataFlow::CallNode pred, DataFlow::Node succ) {
  exists(ConditionGuardNode g, SsaVariable v |
    // pred is the guard
    g.getTest() = pred.asExpr() and
    // in SSA, succ is an argument of the guard, which is a function call
    pred.getAnArgument().asExpr() = v.getAUse() and
    // succ is later used inside the conditional guarded by g
    succ.asExpr() = v.getAUse() and
    g.dominates(succ.getBasicBlock())
  )
}
```

Fig. 7.2: Fragmento de QL mostrando la implementación de la regla GUARDA descripta en 4.3.3.

BIBLIOGRAFÍA

[1] Open Worldwide Application Security Project (OWASP). *OWASP Top 10*. https://owasp.org/Top10/.

[2] Lars Ole Andersen. "Program analysis and specialization for the C programming language". En: (1994).

[3] Victor Chibotaru et al. "Scalable Taint Specification Inference with Big Code". En: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, págs. 760-774. ISBN: 9781450367127. DOI: 10.1145/3314221.3314648. URL: https://doi.org/10.1145/3314221.3314648.

[4] Saikat Dutta et al. "InspectJS: Leveraging Code Similarity and User-Feedback for Effective Taint Specification Inference for JavaScript". En: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, págs. 165-174. ISBN: 9781450392266. DOI: 10.1145/3510457.3513048. URL: https://doi.org/10.1145/3510457.3513048.

[5] Github. *CodeQL*. https://codeql.github.com/docs/codeql-overview/about-codeql/.

[6] Github. *Octoverse 2022 Report*. https://octoverse.github.com/2022/top-programming-languages. 2022.

[7] Daniel Hedin et al. "JSFlow: tracking information flow in JavaScript and its APIs". En: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. por Yookun Cho et al. ACM, 2014, págs. 1663-1671. DOI: 10.1145/2554850.2554909. URL: https://doi.org/10.1145/2554850.2554909.

[8] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. Springer, 2013. URL: https://faculty.marshall.usc.edu/gareth-james/ISL/.

[9] Simon Holm Jensen, Anders Møller y Peter Thiemann. "Type Analysis for JavaScript". En: *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*. Ed. por Jens Palsberg y Zhendong Su. Vol. 5673. Lecture Notes in Computer Science. Springer, 2009, págs. 238-255. DOI: 10.1007/978-3-642-03237-0_17. URL: https://doi.org/10.1007/978-3-642-03237-0%5C_17.

[10] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[11] V. Benjamin Livshits et al. "Merlin: specification inference for explicit information flow problems". En: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. por Michael Hind y Amer Diwan. ACM, 2009, págs. 75-86. DOI: 10.1145/1542476.1542485. URL: https://doi.org/10.1145/1542476.1542485.

[12] MITRE. *MITRE CWE (Common Weakness Enumeration) Top 25*. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html. 2023.

[13] Oege de Moor et al. “Keynote Address: .QL for Source Code Analysis”. En: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 2007, págs. 3-16. DOI: 10.1109/SCAM.2007.31.

[14] Benjamin Barslev Nielsen. “Static Analysis for Node.js”. En: *PHD thesis*. Aarhus University, dic. de 2020.

[15] Stack Overflow. *Stack Overflow Developepr Survey 2023*. https://survey.stackoverflow.co/2023/. 2023.

[16] Yannis Smaragdakis y George Balatsouras. “Pointer Analysis”. En: *Found. Trends Program. Lang.* 2.1 (2015), págs. 1-69. DOI: 10.1561/2500000014. URL: https://doi.org/10.1561/2500000014.

[17] Cristian-Alexandru Staicu et al. “Extracting Taint Specifications for JavaScript Libraries”. En: *Proc. 42nd International Conference on Software Engineering (ICSE)*. Mayo de 2020.

[18] Haiyang Sun et al. “Efficient dynamic analysis for Node.js”. En: *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*. Ed. por Christophe Dubach y Jingling Xue. ACM, 2018, págs. 196-206. DOI: 10.1145/3178372.3179527. URL: https://doi.org/10.1145/3178372.3179527.