Mejorando Taint Analysis para JavaScript con grandes cantidades de código

Pablo L. Balbi

Defensa de Tesis de Licenciatura

Departamento de Computación
Universidad de Buenos Aires

19 de julio de 2024

¿Qué veremos?

¿Qué veremos?

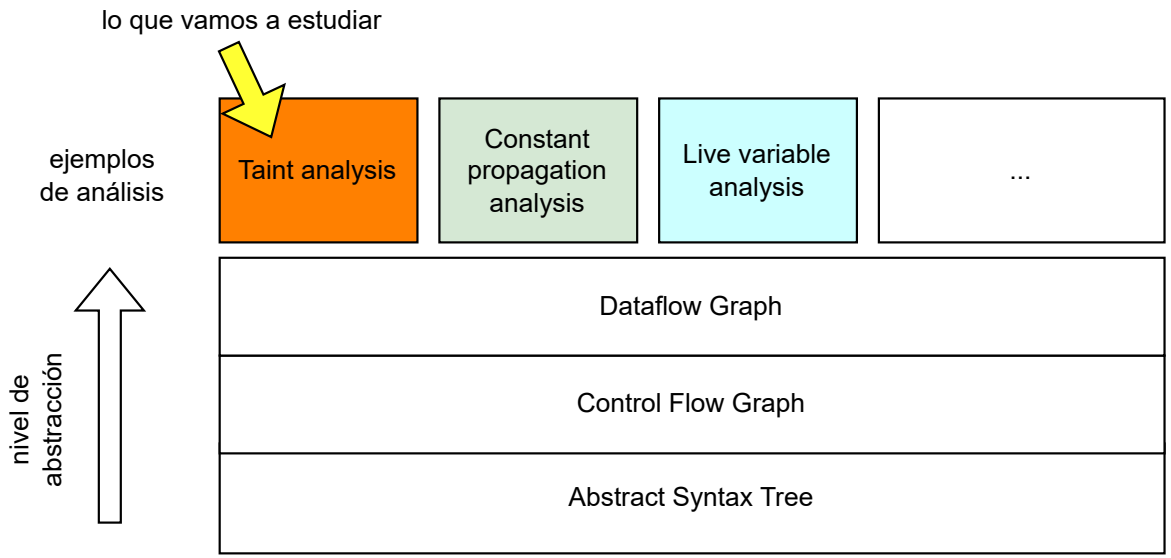
Introducción

- Tesis realizada dentro del **LaFHIS**, laboratorio que, entre otros temas, estudia las técnicas automáticas que permiten analizar diversos productos de la ingeniería de software, como requerimientos, diseños y programas
- Dirigida por el Dr. Diego Garbervetsky
- Parte de un proyecto de investigación, con participación de MSR (Microsoft Research) y Github

¿Qué veremos?

Program Analysis

- Estudio de programas que toman como entrada otro programa, y permiten aprender propiedades sobre los mismos (invariantes, modelos, uso de recursos, fallas, etc.).
- Nos interesa la variante estática.



JavaScript

- Lenguaje más usado en la industria desde 2014, según GitHub Octoverse report. Básicamente, cualquier aplicación web interactiva lo utiliza.
- Desde 2015, tiene más CVEs (vulnearbilidades) reportadas por año que lenguajes otros lenguajes vasto uso (Javascript 4548, Java 2633 y Python 776).
- Características hacen que sea dificil de analizar estáticamente: modelo de ejecución single-threaded y basado en eventos, multiparadigma, prototipado, operadores raros como `eval`, muy modular, etc.

Taint analysis - Intuición

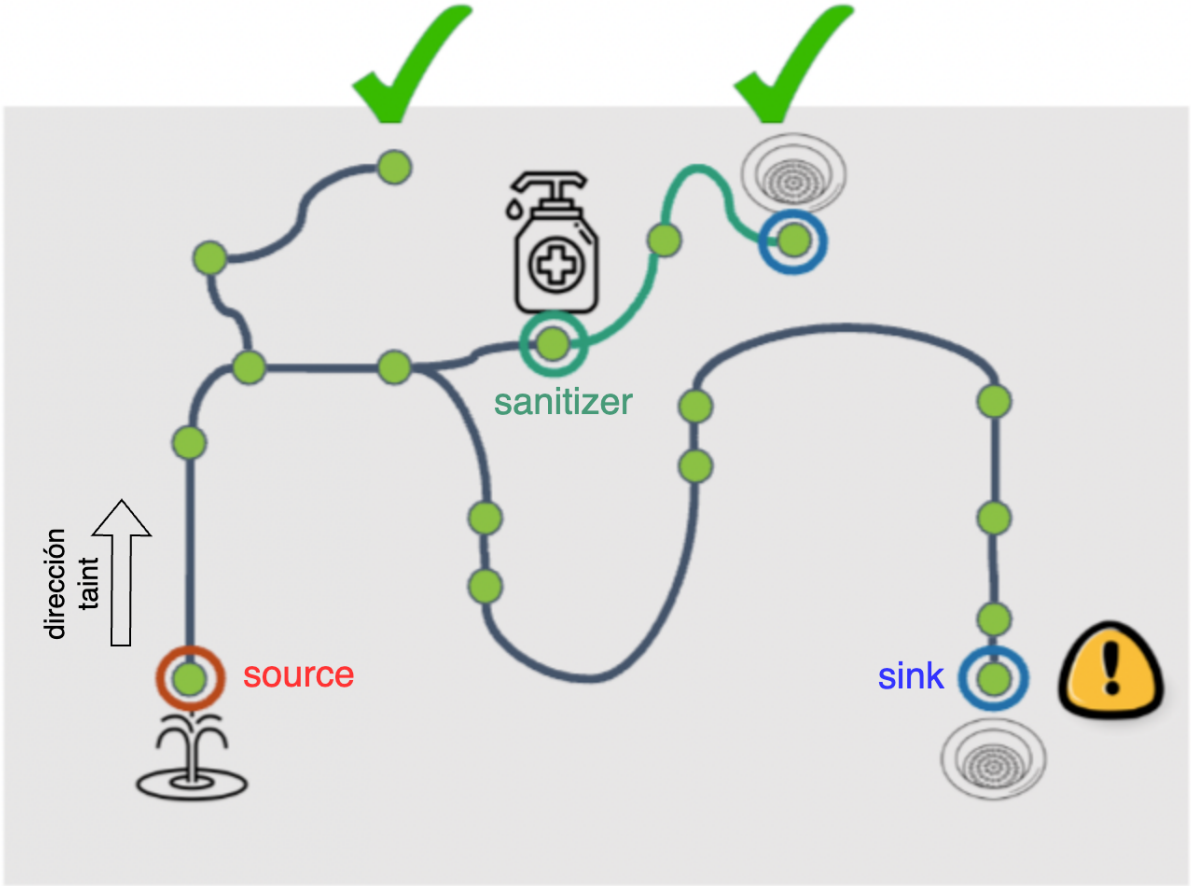


Figura 1: La idea detrás de taint analysis (o taint tracking) es encontrar si existe un camino (flujo de datos) de un source a un sink, sin pasar por un sanitizer. La descripción de qué es un source, sanitizer y sink es lo que llamaremos **especificaciones**.

Taint analysis - Ejemplo NoSQL Injection

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 });
```

Tengamos en cuenta las siguientes "especificaciones de taint analysis":

- **Sources**: Requests HTTP, `async (req, res) => {...}`
- **Sanitizers**: Ninguno
- **Sinks**: Queries a una base de datos, `db.collection(c).findOne(query)`

Taint analysis - Ejemplo NoSQL Injection

Listing 2 Ejemplo de código mostrando una vulnerabilidad de NoSQL injection

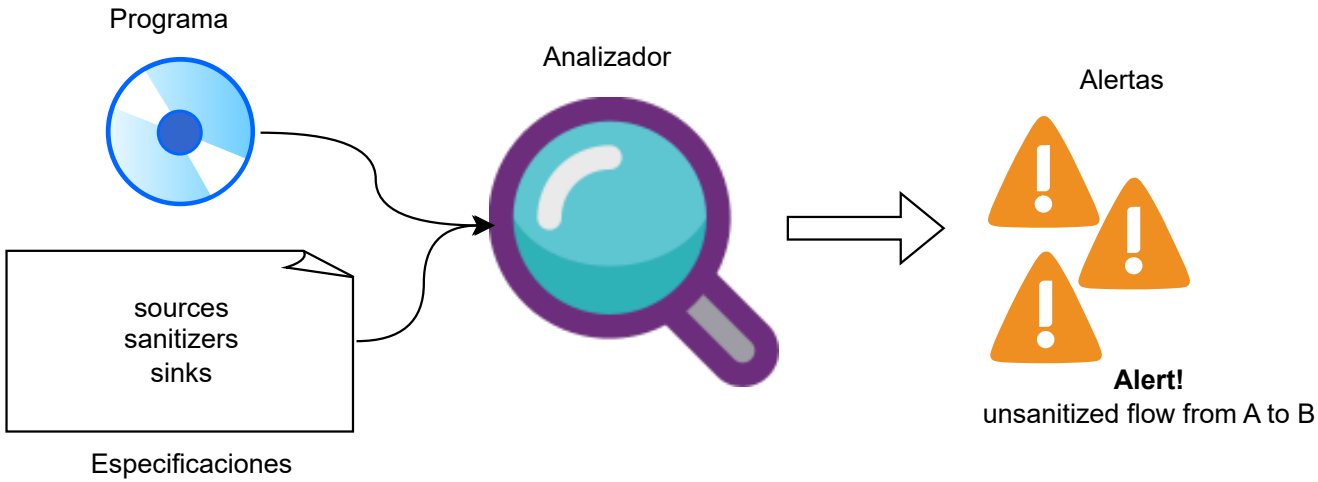
```
1 app.get('/search', async (req, res) => {
2   let title = req.query.title;
3   let data = await db.collection('movies').findOne({
4     $where: `this.title == '${title}'`,
5   })
6   res.json({
7     data: data,
8   });
9 });
```

Ejemplo de dónde sale **title**
http://url/search?**title=algo**

Ejemplos de valor de **title**
- " || "" == "
- Volver al futuro

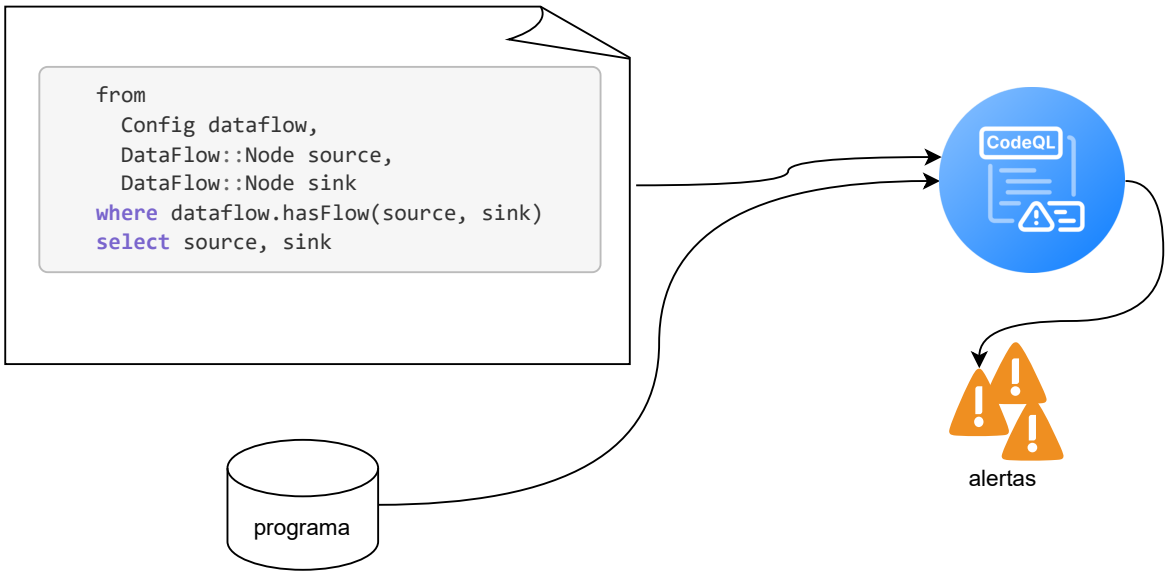
Taint analysis

- Tipo de particular de data-flow analysis, que estudia cómo se propagan valores "marcados" a lo largo de un programa.
- Busca encontrar caminos "prohibidos" entre un source y un sink, que no pasen por un sanitizer
- En esta tesis trabajamos con la versión estática.



CodeQL

- Motor de análisis estático de código, multilenguaje, desarrollado por Semmle[1]
- Lenguaje de consultas, llamado QL, inspirado en Datalog[2]
- Un programa es interpretado como una base de datos, sobre la cual se realizan consultas, combinando predicados a diferentes niveles de abstracción (syntax, control, data).



[1] Adquirida por GitHub.

[2] Subconjunto de Prolog, usado para bases de datos deductivas

CodeQL - Ejemplos de consultas en QL

Ejemplo - ¿Existe algún if cuya rama verdadera esté vacía?:

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

Ejemplo 2 - ¿Existen bloques de código no alcanzables?:

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

¿Qué veremos?

Problema de investigación

- Taint analysis es un tipo de análisis muy estudiado
- Donde las especificaciones son comúnmente redactadas **manualmente**
- Muchos trabajos previos abordan la problemática del aprendizaje automático de especificaciones
- Seldon [1] ataca este problema para Python, con una idea central: partiendo de un conjunto pequeño de especificaciones, y uno grande de programas, **formular el problema de inferencia como uno de programación lineal**

¿Qué preguntas guiaron la 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**: ¿Sirven las especificaciones inferidas por nuestra técnica para descubrir nuevas alertas? ¿Cómo se compara nuestra técnica con Seldon, en cuanto a esto?
3. **RQ3**: ¿Las especificaciones inferidas resultan relevantes para la comunidad?

Terminología

- **Término de programa**: También conocido en la literatura como *program element*, hace referencia a un nodo dentro del CFG, por ejemplo una asignación, una llamada a función; la guarda de un condicional, entre otros.
- **Especificaciones**: Especificaciones de taint analysis. Refiere a la descripción, en algún formato interpretable, acerca de cómo son los sources, sinks y sanitizers que nos interesa que tenga en cuenta el analizador.
- **EI**: Especificaciones inciales

¿Qué veremos?

¿Qué hace Seldon?

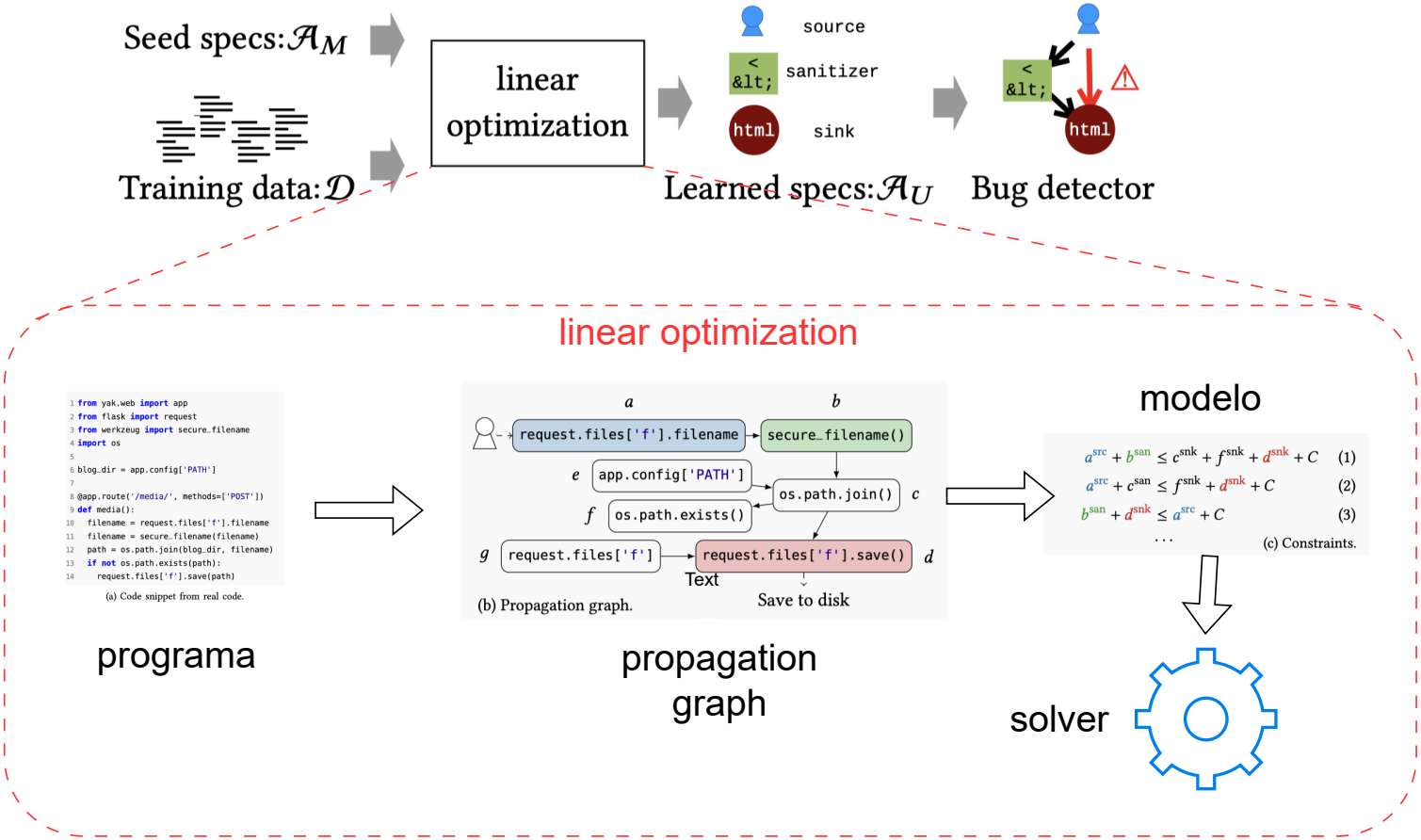
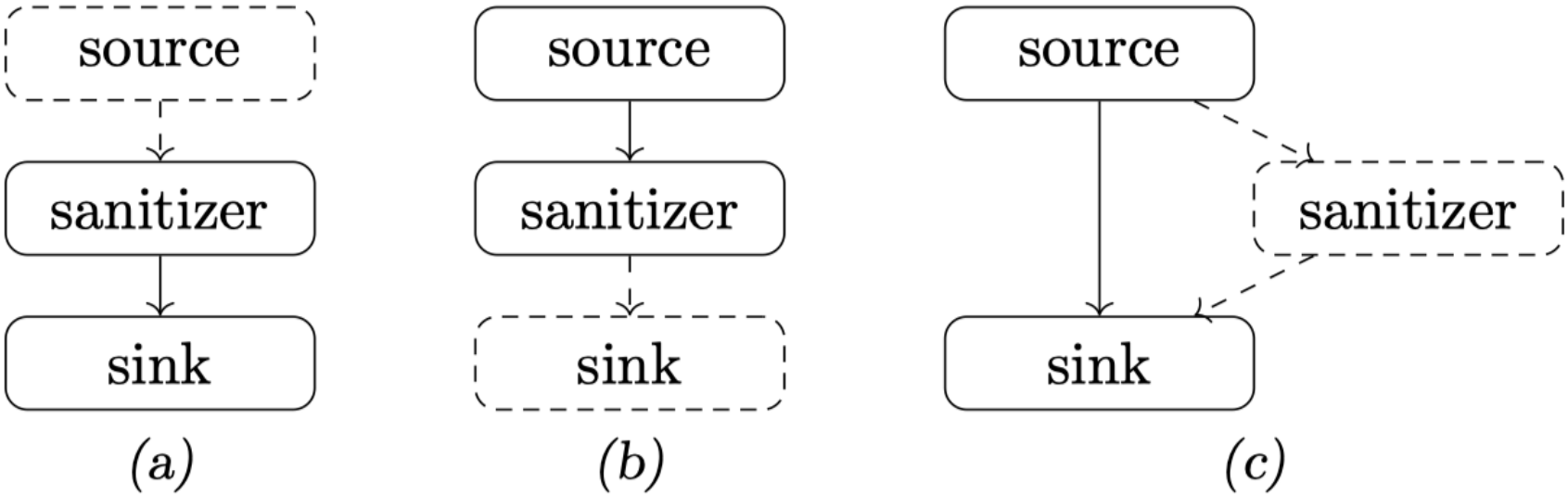


Figura 2: Diagrama de arquitectura de Seldon

Idea detrás del modelo

Seldon construye el modelo a partir de una serie de “intuiciones” o “patrones” que describen cómo se relacionan términos con diferentes roles de taint tracking, dentro de un programa.

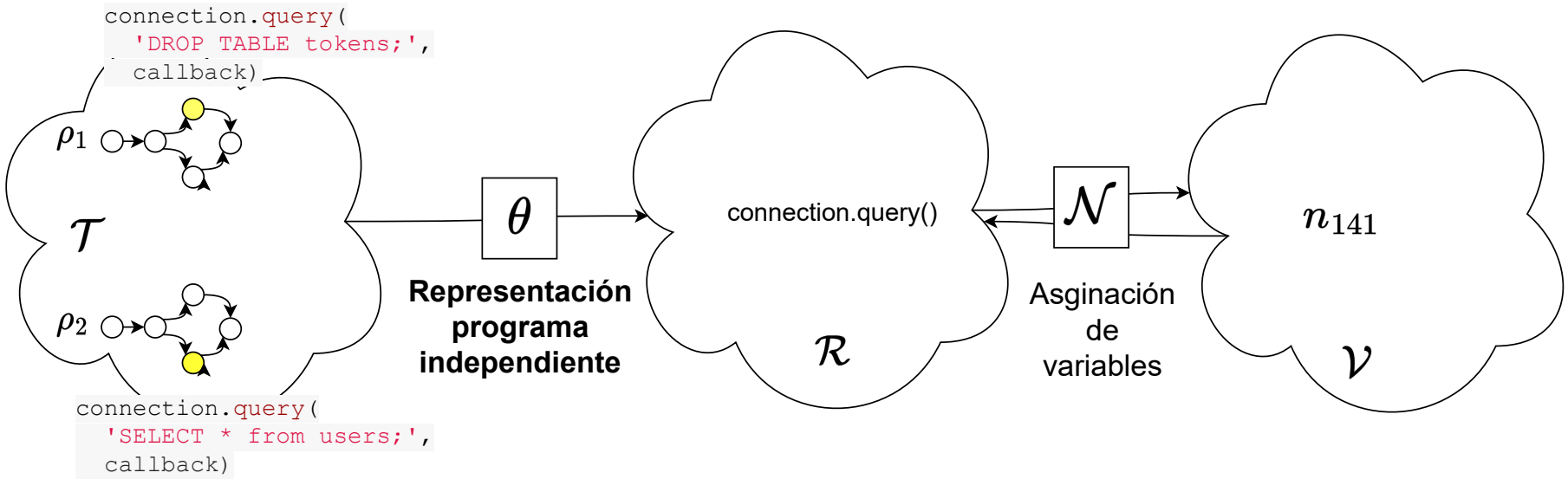


Propagation graph

- Es un grafo dirigido $G = \langle V, E \rangle$ donde los elementos en V son términos que consideramos **relevantes**, y los ejes en E nos indican que hay **flujo de información** entre dos vértices. También nos referiremos como **propgraph**.
- Ejemplo de términos relevantes en Seldon: Llamadas a función.
- Ejemplo de flujo de información: Flujo entre relaciones de Points-To ("si dos variables apuntan al mismo objeto").

¿Qué son las variables?

Es importante entender cómo un término de un programa pasa de ser un vértice del grafo, a una variable dentro del modelo. La relación θ define lo que llamaremos **Representación Programa Independiente**, o REPR.



El que las variables correspondan a REPR le permite a Seldon que grafos disjuntos (porque surgieron de programas diferentes), estén unificados dentro del modelo.

Restricciones sobre las variables del sistema

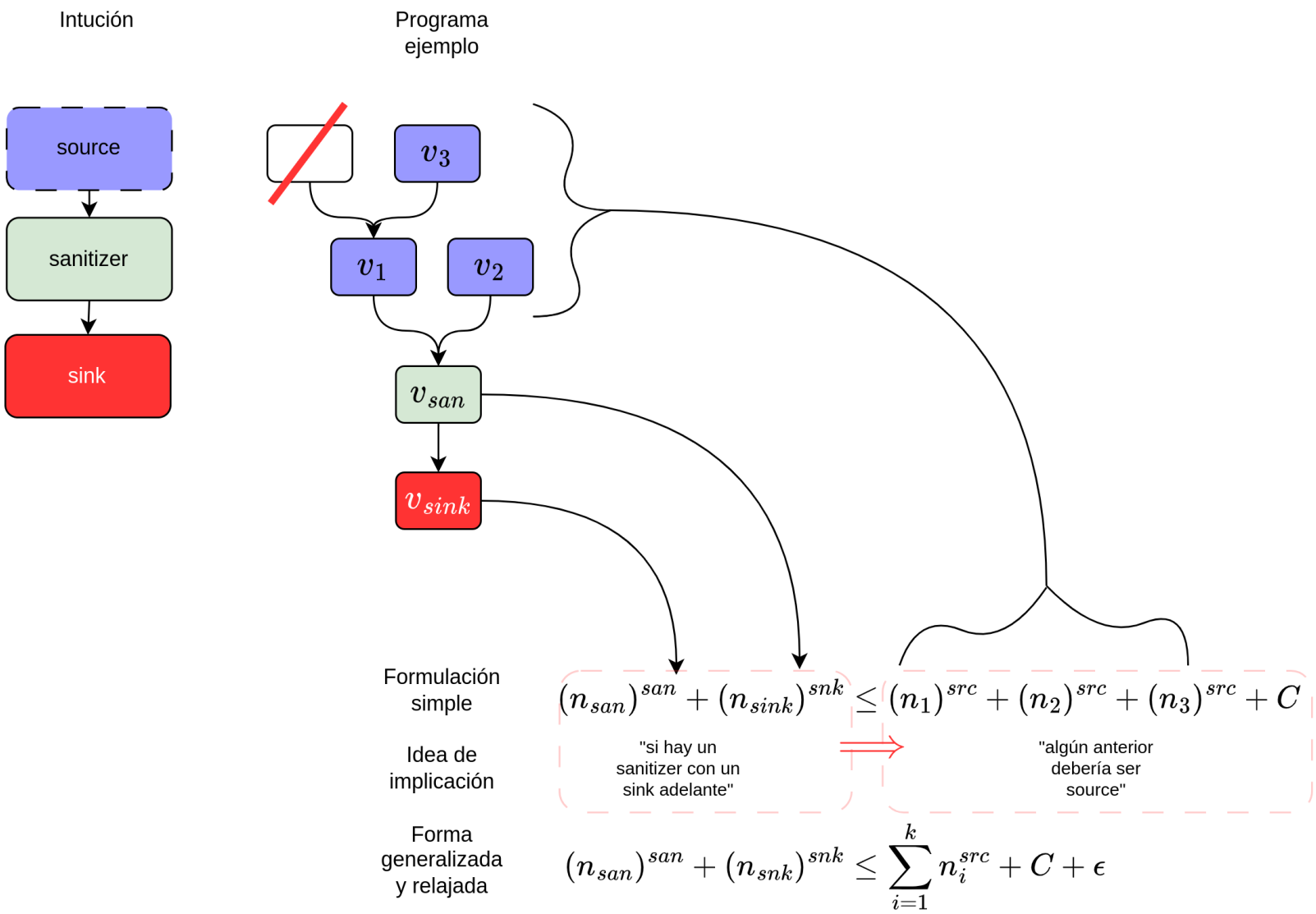
Restricciones sobre las variables, de forma que las mismas sean *interpretables* como una probabilidad:

$$0 \leq (n_r)^{rol} \leq 1 \,|\, r = Rep(t), rol \in \{src, san, snk\} \tag{1}$$

Restricciones construidas a partir de las especificaciones iniciales:

$$\begin{cases} n^{rol} = 1 \\ n^{oR} = 0 \quad | \, oR \in \{src, san, snk\} \setminus \{rol\} \end{cases} \forall \langle t, rol \rangle \in \mathcal{A}_M \mid n = Rep(t) \tag{2}$$

De intuición a restricciones



Formulando el problema de optimización

En Seldon se construye un **único modelo de optimización** con las restricciones que surgen de cada programa en $\mathcal{D}$. El modelo de optimización lineal está compuesto por las siguientes partes:

$$min \left(\sum_{i=1}^{M} \epsilon_i + \lambda \sum_{r \in \mathcal{R}} ((n_r)^{src} + (n_r)^{san} + (n_r)^{snk}) \right) \tag{3}$$

$$\text{sujeto a } 0 \leq (n_r)^{rol} \leq 1 \; \forall n_r \text{ y} \tag{4}$$

$$0 \leq \epsilon_i \; \forall i \in \{1, ..., M\} \text{ y} \tag{5}$$

$$\begin{cases} (n)^r = 1 \\ (n)^t = 0 \quad \forall t \in \{src, san, snk\} \setminus \{r\} \end{cases} \quad \forall \langle v, r \rangle \in \mathcal{A}_M \mid n = Rep(v) \tag{6}$$

$$\text{y cada una de las } M \text{ restricciones } c_i \text{ en } C^{flow} \tag{7}$$

Optimización y aplicación

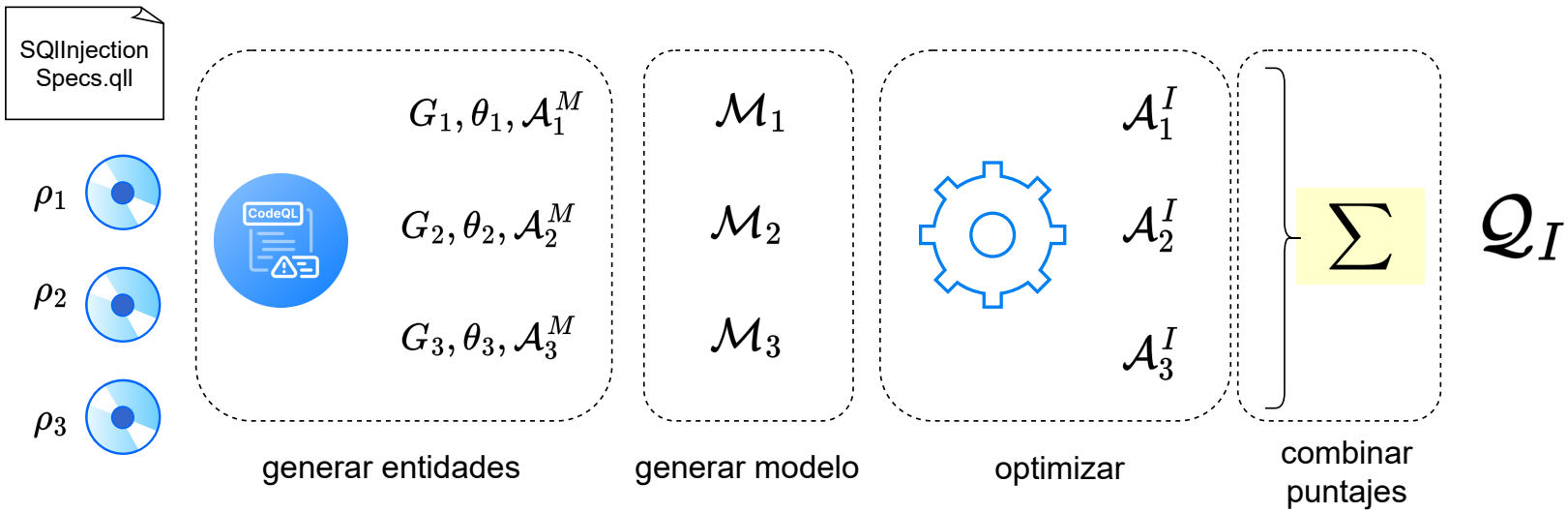
- La optimización es llevada a cabo mediante un solver proveniente de una biblioteca de ML[3].
- Una vez obtenidas las especificaciones inferidas, son usadas para una implementación de taint analysis construida sobre el mismo propagation graph.

[3] Tensorflow Adam Optimizer

¿Qué veremos?

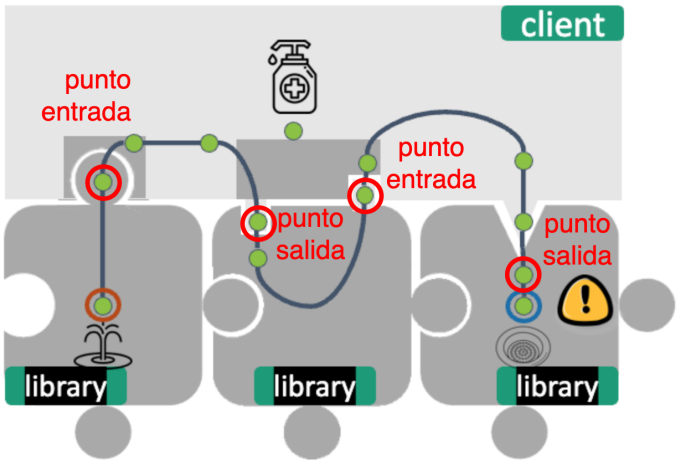
Introducción Jeldon

- Misma formulación del problema de optimización, pero con JavaScript como lenguaje objetivo. Un modelo por programa de entrada, y luego se "combinan" las inferencias.
- Usamos CodeQL como motor de análisis estático, tanto para la inferencia, como la aplicación de las especificaciones aprendidas.
- Las EI son queries de CodeQL, y que **predican sobre una vulnerabilidad en particular**. En nuestro caso, nos interesamos en NoSQL Injection, Tainted Path y XSS.

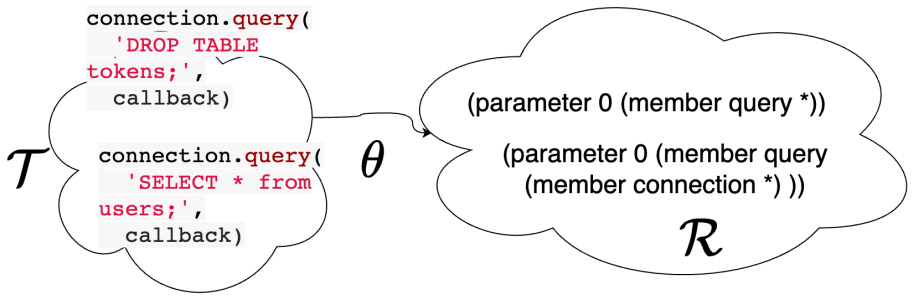


Representación programa independiente

- Implementar una aplicación HTTP súper simple, requiere aproximadamente 50 dependencias, para 20 LOCs.
- Definimos **puntos de contacto** como la frontera entre un programa y sus dependencias (pensar las interfaces que estas exponen, o su API).
- Estos "puntos" son un objetivo interesante para modelar bien.
- Nos basamos en *access paths*, representación introducida en TASER [3], donde se usa para capturar puntos de contacto, agregando la expresión $*$.

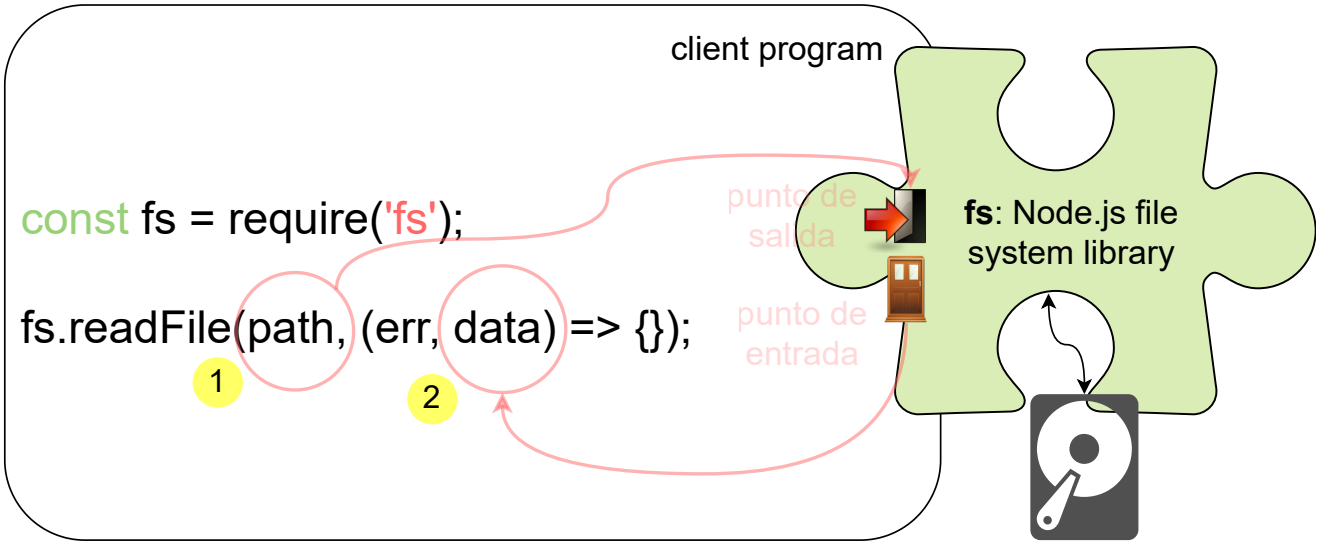


$$\theta : \mathcal{T} \rightarrow \rho(\mathcal{R})$$



REPR y puntos de contacto

En el siguiente ejemplo de una biblioteca de Node.js, `path` es un punto de salida, y `data` es un punto de entrada.



- **1 - opt 1**: (parameter 0 (member readFile *))
- **1 - opt 2**: (parameter 0 (member readFile (root fs)))
- **2**: (parameter 1 (parameter 1 (member readFile (root fs))))

El ejemplo nos muestra cómo la representación 1 y la 1.1 corresponden al mismo término, pero con diferente especificidad.

Generar entidades: Propagation graph y otros

- Este etapa se encarga de construir el propagation graph, hallar los términos que surgen de las EI[4], y la función de traducción θ, todo implementado sobre QL:
  - **EI**: Como son consultas en QL, podemos computar directamente $\mathcal{A}_M$
  - **Repr**: Computamos el $\theta(t)$ para t cada término dentro del programa
  - **Propgraph**: Veremos como se extrae a continuación.

[4] Especificaciones iniciales

[5] Datalog es un lenguaje de programación lógico declarativo, y sintácticamente, un subset de Prolog.

Generar entidades: Propagation graph y otros

- Este etapa se encarga de construir el propagation graph, hallar los términos que surgen de las EI[4], y la función de traducción θ, todo implementado sobre QL:
  - **EI**: Como son consultas en QL, podemos computar directamente $\mathcal{A}_M$
  - **Repr**: Computamos el $\theta(t)$ para t cada término dentro del programa
  - **Propgraph**: Veremos como se extrae a continuación.

En las siguientes slides se ilustran las consultas implementadas en CodeQL en esta tesis, usando Datalog[5]. Se usarán algunas relaciones de Datalog sin presentar su explicación, por motivo de síntesis.

[4] Especificaciones iniciales

[5] Datalog es un lenguaje de programación lógico declarativo, y sintácticamente, un subset de Prolog.

PropGraph: Vértices, o términos relevantes

Las siguientes reglas de datalog definen el conjunto de vértices del grafo. Las relaciones PuntoDeEntrada y PuntoDeSalida predican sobre si un término es un punto de contacto.

$$\begin{aligned}
&\textsc{Relevante}(t) \leftarrow \quad \textsc{CandidatoASource}(t);\ \textsc{CandidatoASanitizer}(t);\\
&\textsc{CandidatoASink}(t).\\
\\
&\textsc{CandidatoASource}(t) \leftarrow \textsc{PuntoDeEntrada}(t),\\
&\quad (\textsc{LlamadaAFunción}(t);\ \textsc{EsArgumento}(t);\ \textsc{LecturaAtributo}(t)).\\
&\textsc{CandidatoASanitizer}(t) \leftarrow \textsc{LlamadaAFunción}(t).\\
&\textsc{CandidatoASink}(t) \leftarrow\\
&\quad \textsc{PuntoDeSalida}(t),\\
&\quad (\textsc{EsReturn}(t);\ \textsc{EsArgumentoOBase}(t);\ \textsc{EscrituraAtributo}(t)).
\end{aligned} \tag{8}$$

Entonces, los "posibles sources y sinks" son puntos de contacto, y los sanitizers cualquier llamada a función.

PropGraph: Ejes, o flujo de información

Para computar los ejes del propagation graph utilizamos el motor de taint tracking de CodeQL (**usado internamente por taint analysis**), en sus versiones inter e intra-procedural.
En la consulta debajo, PasoIntraProcedular y PasoInterProcedular son parte del core de CodeQL.

$$\begin{aligned}
&\textsc{Alcanzable}(\mathit{desde}, \mathit{desde}).\\
&\textsc{Alcanzable}(\mathit{desde}, \mathit{hacia}) \leftarrow \textsc{Alcanzable}(\mathit{desde}, \mathit{int}), \\
&\quad (\textsc{PasoIntraProcedular}(\mathit{int}, \mathit{hacia}); \textsc{PasoInterProcedural}(\mathit{int}, \mathit{hacia})).
\end{aligned} \tag{9}$$

PropGraph: Uso para construcción del modelo

Para facilitar la construcción de las restricciones de flujo (C^{flow}), optamos por: a partir del grafo, construir la relación de TRIPLA. La misma nos permite fácilmente fabricar una restricción, a partir de sus resultados.

$$\text{TRIPLA}(source, san, sink). \tag{10}$$

Veamos como construyo una restricción a partir de TRIPLA:

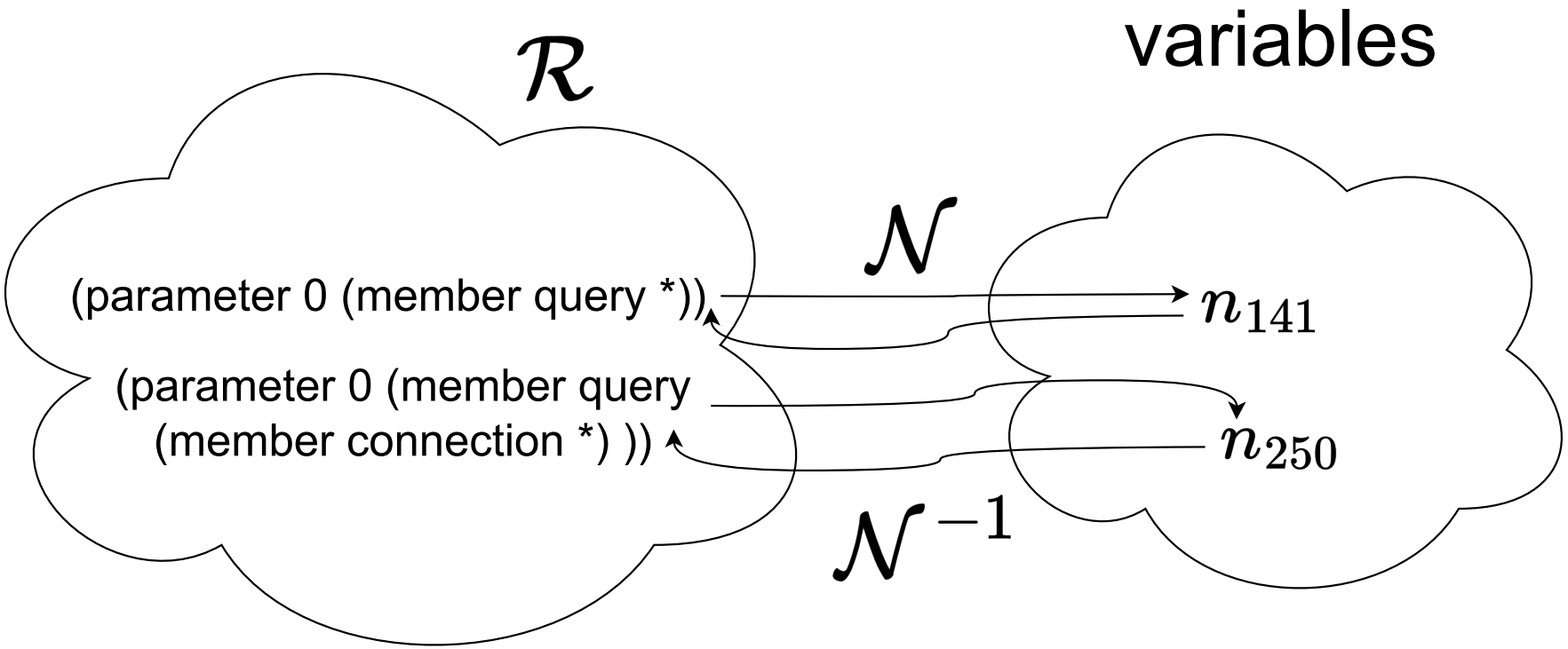
buscamos las tuplas de la relación Tripla cuyas primeras dos componentes sean a,b

$$Tripla(a,b,\star) \begin{cases} Tripla(a,b,c_1) \\ Tripla(a,b,c_2) \\ Tripla(a,b,c_3) \\ Tripla(a,b,c_4) \end{cases} \Longrightarrow \quad n_a^{src} + n_b^{san} - \sum_{i=1}^{4} n_i^{src} - C \leq \epsilon$$

construyo restricción

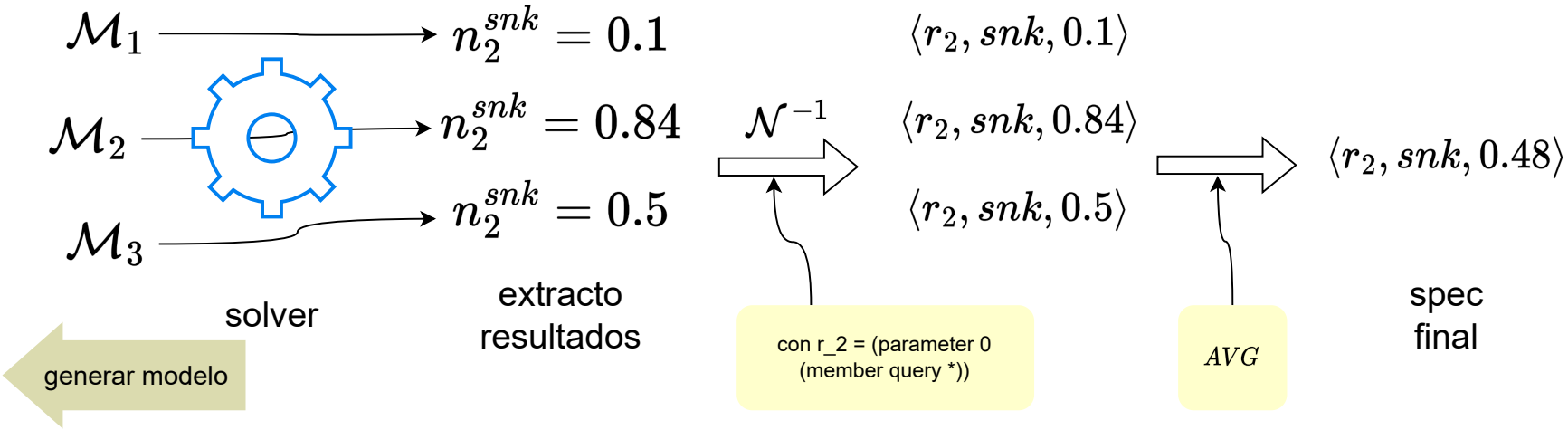
Modelo y asignación de variables

- Hasta este punto ya obtuvimos las partes del modelo: $\mathcal{A}_M$, θ y C^{flow}.
- Nos hace falta traducir del universo de $\textsc{Reprs}$ $(\mathcal{R})$ a variables, lo hacemos mediante una relación $\mathcal{N}$ inversible.



Solver, resolución del modelo, y combinación de inferencias

Utilizamos un solver llamado CLP[6]. El siguiente diagrama explica cómo pasamos de los modelos armados, a las specs finales:



Al momento de usarlas, filtramos las especificaciones inferidas con un umbral $\textsc{MinScore}_{rol}$.

[6]COIN-OR Linear Programming Solver. Github. Originalmente usamos Gurobi, pero es closed-source y pago.

¿Qué inferimos en Jeldon?

- Durante el desarrollo de la técnica, **en el contexto de las especificaciones iniciales que usamos**, notamos que inferir sources y sanitizers no aportaba valor a la performance de la técnica. ¿Por qué?
  - **Sources**: Los sources suelen ser parte bibliotecas HTTP, que son muy parecidas entre sí. Entonces, los modelos existentes tienen buena cobertura.
  - **Sanitizers**: Suelen desarrollarse de manera ad-hoc, y como al computar el propgraph tenemos una definición muy amplia de "posible sanitizer", esto da muchos falsos positivos.
- Por ello, la técnica tiene en cuenta solamente **sinks**

¿Qué veremos?

¿Cómo medimos unas especificaciones?

- Necesitamos una manera de medir qué aportan las especificaciones a taint analysis.
- Agregar elementos a una especificación, hace que el análisis tenga mayor "coverage", y por consiguiente, debería descubrir mayor cantidad de flujos prohibidos (alertas).
- Es esperable que si las especificaciones inferidas son de utilidad, descubran mayor cantidad de alertas.
- Pero... ¿Son correctas las alertas?
- ¿Cómo corroboramos si las alertas son correctas?

Intro. a metodología de evaluación

Para corroborar si una alerta es correcta o no, hace falta construir un "oráculo" (que nosotros llamaremos analizador de referencia, o ADR). Con él, podemos clasificar los errores que producen nuestras especificaciones al clasificar alertas:

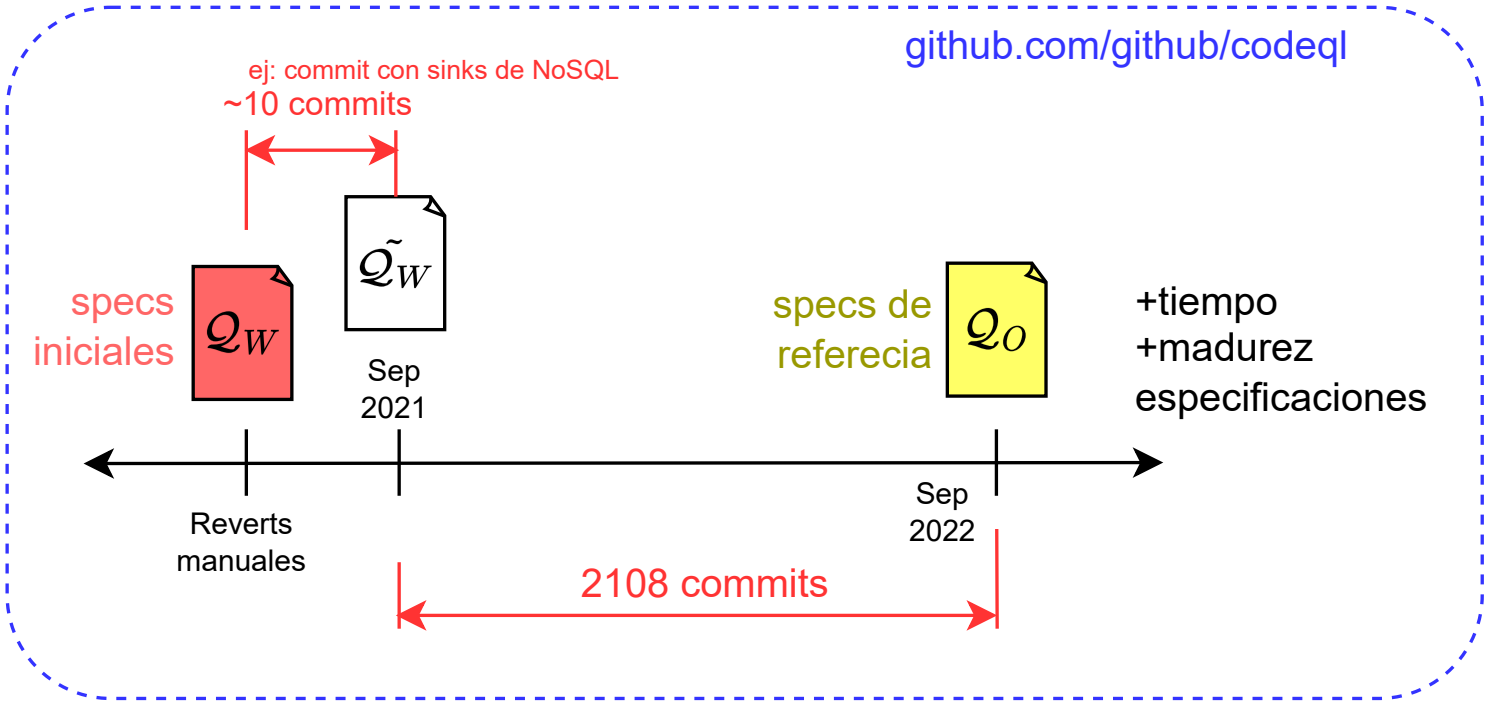
Jeldon	Analizador de referencia (ADR)	
	V	F
V	Verdadero positivo (TP)	Falso positivo (FP)
F	Falso negativo (FN)	~~Verdadero negativo (TN)~~

Y así calcular:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN} \tag{11}$$

Construyendo el ADR

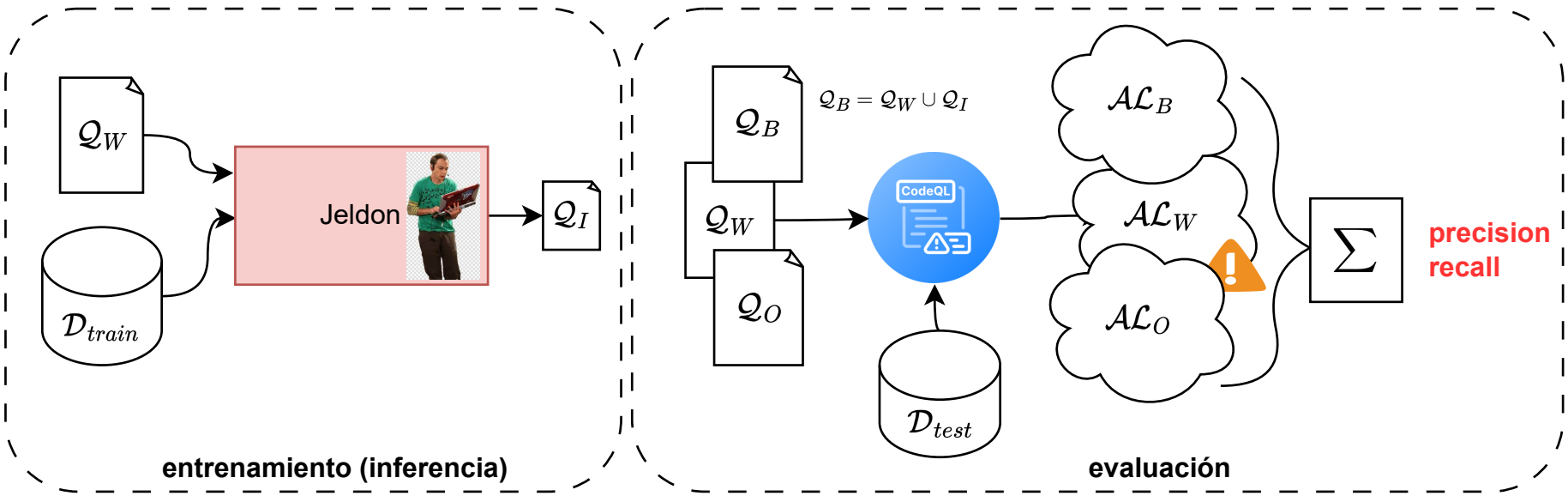
- Recordando, las especificaciones son parte de la biblioteca de consultas de CodeQL[7], cuya historia de cambios es pública.
- Construiremos pares de especificaciones $\mathcal{Q}_O$ y $\mathcal{Q}_W$, para cada clase de especificaciones (NoSQL, TaintedPath, y XSS).



[7] https://github.com/github/codeql/

Experimentación

- Se evaluarán 3 clases de especificaciones: NoSQL Injection, Tainted Path y XSS. Las EI y el ADR fueron construidas como mencionamos antes
- Se utilizará 3 conjuntos de programas, uno para cada clase de especificación, con tamaños 257, 336 y 90
- Para obtener los conjuntos de entrenamiento y evaluación, se usa 5-Fold Cross Validation



Experimentación - Hiper-paramétros

El principal motivador para la elección de los hiper-parámetros, fue que se buscó priorizar *recall* sobre *precision*.

Parámetro	Valor	¿Por qué?
C	$0{,}75$	Heredado de Seldon
λ	$0{,}1$	Heredado de Seldon
MaxReprDepth	4	Punto medio entre especificidad y cantidad de representaciones
MinScoreSink	$0{,}1$	Valor más bajo posible, buscando maximizar recall, pero impactando precision

Resultados

- Haber obtenido valores de $66\,\%$ y $81\,\%$ para dos de las clases evaluadas, **nos indica que las especificaciones inferidas son útiles**.
- Cantidad alta de falsos positivos (precision baja) en las tres clases de especificaciones. Surge de que buscamos maximizar el recall.
- Los resultados obtenidos para XSS no fueron satisfactorios, y los atribuimos a la dificultad de modelar este tipo de vulns con taint analysis, y la baja cantidad de datos.
- **No podemos compararnos contra los resultados presentados en Seldon**, ya que ellos verifican manualmente las especificaciones, y para alertas, solo miden la cantidad obtenida (sin corroborarlas)

	Precision	**Recall**	$\overline{t_{train}}$	$\sigma_{t_{train}}$
Tainted Path	0.0527	0.6693	2.531	0.252
NoSQL Injection	0.1454	0.8192	1.293	0.148
XSS	0.2000	0.0167	1.124	0.118

Performance

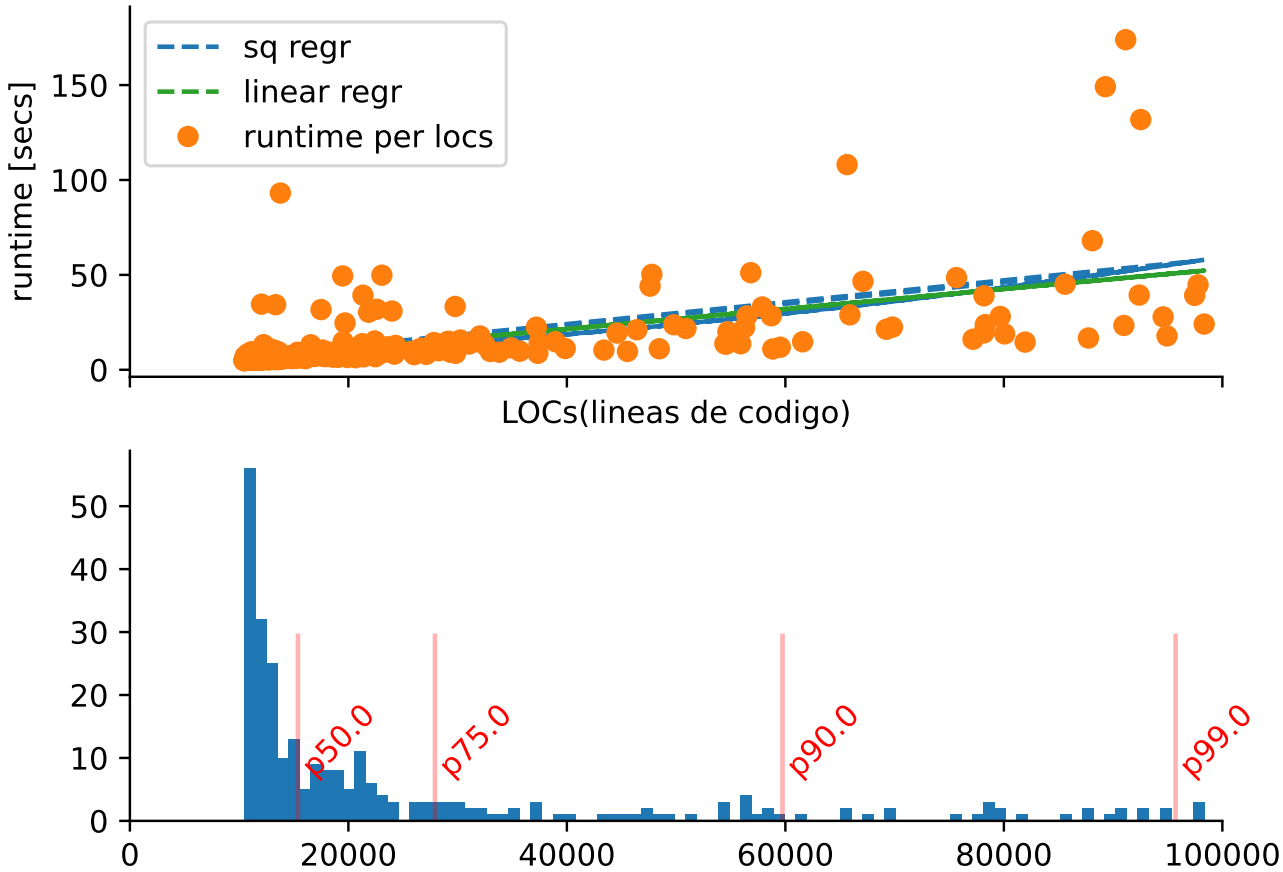


Figura 3: Estudio de 270 muestras de la duración de la etapa de evaluación, a través de las diferentes clases de especificaciones. La figura muestra los puntos de datos en crudo, como una regresión lineal y otra cuadrática, y en la figura inferior un histograma de la cantidad de programas con n líneas de código.

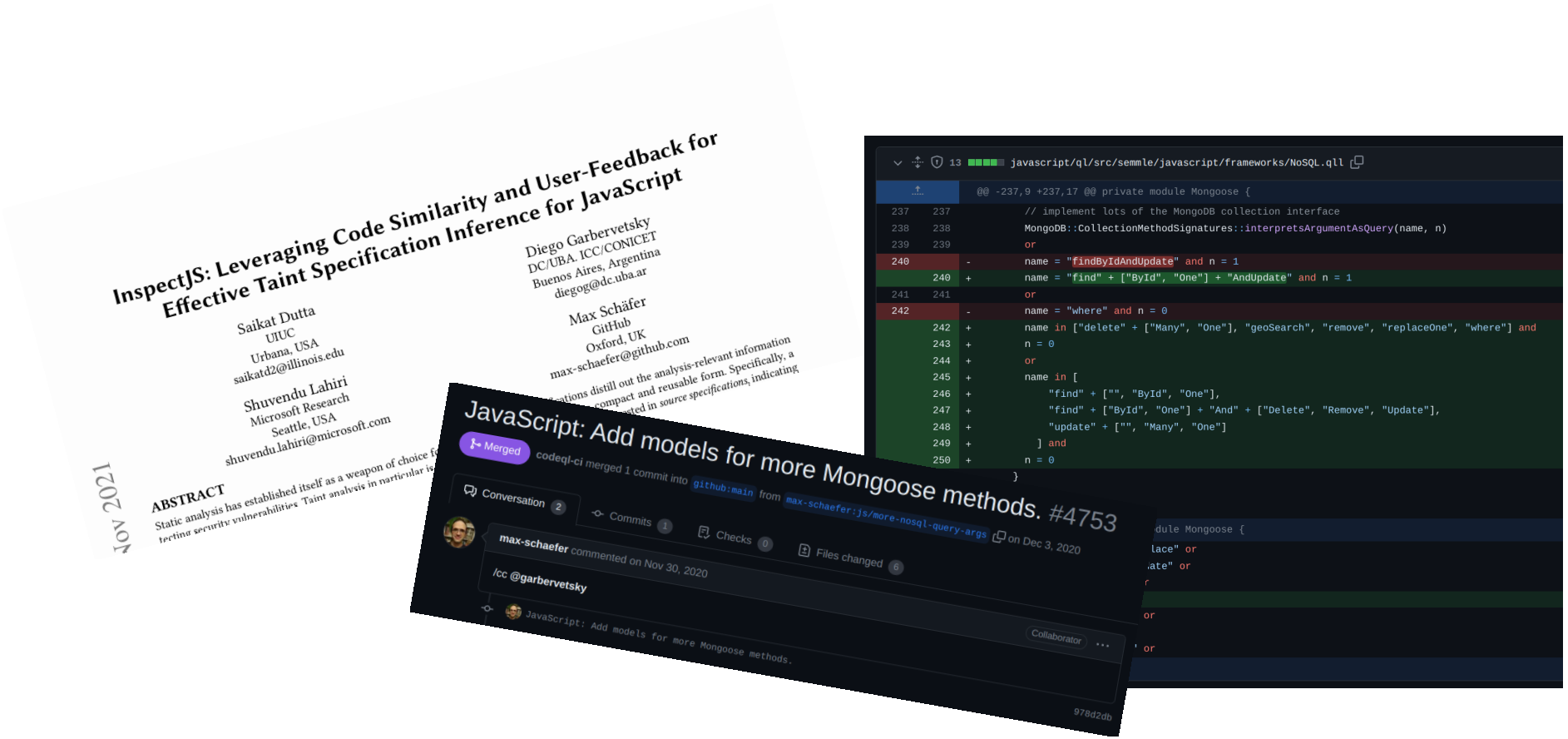
¿Qué veremos?

Conclusiones

- Se presentó una técnica de inferencia de especificaciones de taint analysis para JavaScript, inspirada en Seldon, y usando CodeQL.
- Se presentó una metodología de evaluación que permite medir la técnica de inferencia de manera automática, y por consiguiente, con gran cantidad de datos de prueba.
- Atacamos un problema software, usando técnicas core de Ciencias de la Computación, y construyendo una herramienta que puede ser aplicada en la industria.

Aportes

De este trabajo surgieron diferentes contribuciones a la industria y a la academia: InspectJS [2], donde se usa la técnica, en conjunto con una metodología de filtrado para mejorar la precision; y diversas contribuciones a las bibliotecas de CodeQL.



Trabajos futuros

- Extender la técnica para que se puedan tener en cuenta los sources y sanitizers inferidos
- Exploración de hiper-parámetros
- Explorar mejoras a la formulación del modelo de optimización
- Boostear versiones recientes de las especificaciones de CodeQL
- Evaluar alternativas para mejorar la performance de la técnica

Fin

¿Preguntas?

Agradecimientos

➡️ Gracias a mi hermana y a mis viejos, sin quienes esta tesis se habría hecho esperar mucho más. Doble agradecimiento a mi vieja, quien se dedicó a hacer decenas de revisiones.
➡️ Gracias a mis amigos por acompañarme, bancarme y hacer amenos estos ya casi ocho años. Mención especial a Nico, Ro, Juan y Teo que son fans del taint analysis a esta altura.
➡️ 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 Rodri y Javi por aceptar ser jurados de esta tesis.
➡️ Gracias a toda la gente del Lafhis, por darme un lugarcito durante estos últimos meses donde trabajar, compartir con ellos y aprender sobre investigación.

Referencias

[1] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 760–774, New York, NY, USA, 2019. Association for Computing Machinery.

[2] S. Dutta, D. Garbervetsky, S. K. Lahiri, and M. Schäfer. Inspectjs: Leveraging code similarity and user-feedback for effective taint specification inference for javascript. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 165–174, New York, NY, USA, 2022. Association for Computing Machinery.

[3] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting taint specifications for JavaScript libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*, 5 2020.

Performance en experimentos

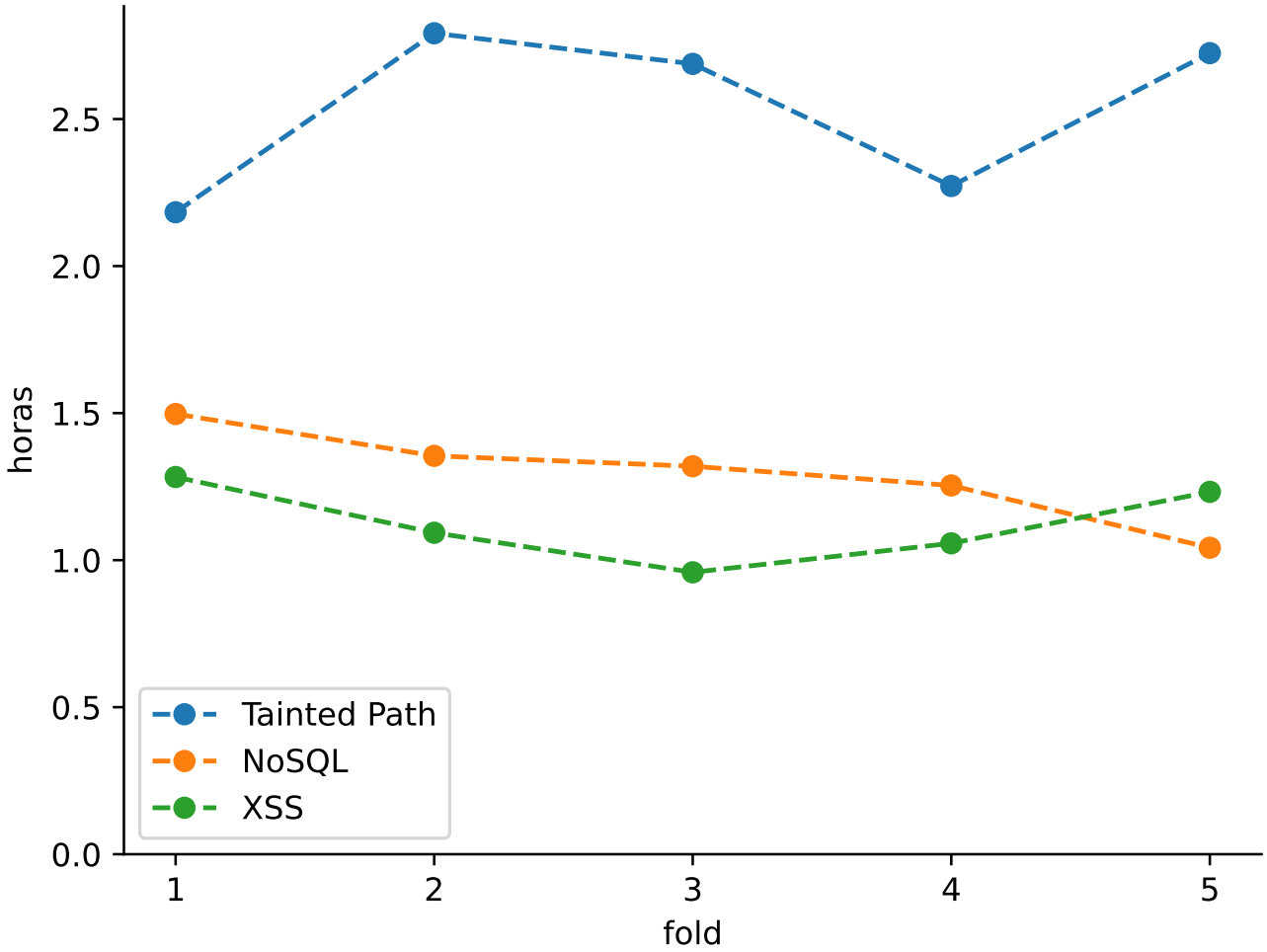


Figura 4: Análisis de la duración de la fase de entrenamiento de cada fold, en el contexto el experimento principal.