Informe final del trabajo de grado:

Análisis de vulnerabilidades en aplicaciones web basado en flujo de información

Alumno: Gabriel Videla
Director: Dr. Pablo E. Martínez López
Dr. Eduardo A. Bonelli

Índice general

Capítulo 1

Introducción

Desde mediados de los 90, el desarrollo web ha sido una de las industrias de mayor crecimiento. El crecimiento de esta industria, así como de pequeñas empresas de diseño y desarrollo web, está siendo promovido por grandes negocios que desean vender productos y servicios a sus consumidores por este medio [1].

El problema de la seguridad informática se ha agravado en los últimos años debido a que la explosión tecnológica que acompañó al mencionado desarrollo de las aplicaciones web durante los 90 no fue acompañada por medidas de seguridad acordes. Como resultado, una gran variedad de ataques cibernéticos continúan afectando la seguridad de las soluciones web con efectos devastadores. Numerosos ataques contra las aplicaciones web son conducidos sobrepasando sus mecanismos de seguridad, sin ser detectados por las soluciones típicas de seguridad (IPS – Intrusion Prevention Systems, IDS – Intrusion Detection Systems y firewalls). El impacto de los ataques a las aplicaciones web puede ser crítico: el atacante puede modificar bases de datos como desee (a través de ataques de inyección de SQL), desviar mecanismos de autenticación (a través de salto de directorio), comprometer el sistema atacado de muchas maneras (por ejemplo, a través de cross-site scripting) [2]. Otra amenaza es el robo de información confidencial de terceras personas (por ejemplo, datos de tarjetas de crédito de clientes de la organización, o información de salud). Cualquiera de ellas puede provocar pérdida de credibilidad y daños en la imagen corporativa, así como pérdidas monetarias.

Las aplicaciones pueden ser protegidas en dos momentos: durante el proceso de desarrollo y durante el ciclo de su vida útil. Independientemente del enfoque usado en el desarrollo para el diseño e implementación de la aplicación web, es recomendable tener algún tipo de test de seguridad o auditoría antes de que la aplicación entre en producción [3].

Conceptualmente conviene recordar que cuando se habla de la seguridad de un programa se hace referencia a las garantías que se pueden ofrecer respecto de que no haya terceras personas, sin acceso autorizado al mismo, que puedan alterar el funcionamiento esperado del programa, los datos con los que el mismo opera o utilicen esos datos para usos no previstos. Estas alteraciones se pueden producir siempre que las terceras personas ganen acceso al programa a través de errores en su codificación o configuración. Conocidos como puntos de vulnerabilidad o vulnerabilidades, están íntimamente relacionados con errores de diseño o errores en el código fuente del programa, ya sea por accidentes de los programadores, o por el uso de estilos de programación de baja confiabilidad que cambian seguridad por eficiencia o bajos costos.

Para ejemplificar, en el siguiente código se observa que si un atacante controla el valor de la variable `$usuario` puede cambiar el significado del comando SQL y hacer operaciones no previstas para el sistema:

```
mysql_query("
                UPDATE usuarios
                SET contrasena = '$contrasena'
                WHERE usuario = '$usuario'
            ")
```

En particular, si el valor de `$usuario` es

```
' OR '0' = '0
```

la consulta SQL pasa a ser

```
UPDATE usuarios
SET contrasena = '...'
WHERE usuario = '' OR '0' = '0'
```

lo cual tiene el efecto de actualizar las contraseñas de todos los usuarios de la base de datos.

Durante el proceso de desarrollo, dada la fuerte relación que existe entre las vulnerabilidades y la ocurrencia de errores en el código, una contramedida de importancia para garantizar la ausencia (o al menos minimizar la presencia) de tales vulnerabilidades es la revisión de código fuente por auditores expertos en seguridad. La tarea del auditor consiste en recorrer manualmente el código, buscando errores conocidos o porciones posiblemente erróneas donde las vulnerabilidades puedan manifestarse.

La revisión manual que se espera de un auditor es una tarea extremadamente cara, pues actualmente los programas están conformados por cientos de miles de líneas de código. No se han desarrollado herramientas robustas, automáticas o semi-automáticas de asistencia a la auditoría que puedan, por ejemplo, señalar puntos potencialmente peligrosos para luego ser revisados manualmente, evitando la necesidad de recorrer la totalidad del código. Por otra parte, si bien existen algunos productos orientados al ciclo de desarrollo, en la práctica son poco utilizados porque detectan patrones demasiado simples o requieren demasiado trabajo extra de los desarrolladores para poder usarlos.

En las siguientes secciones de este capítulo se describen vulnerabilidades en aplicaciones web, la técnica de análisis de programas basada en flujo de información, nuestra contribución y la estructura del informe.

1.1. Vulnerabilidades en aplicaciones web

Una *vulnerabilidad* es una debilidad en una aplicación, que puede ser un problema de diseño o un error en la implementación, que permite a un atacante causar daño a personas interesadas en una aplicación. Estas personas incluyen a los dueños de la misma, los usuarios y otras entidades que confían en la aplicación.

Una de las principales causas de las vulnerabilidades en aplicaciones web es la no validación de las entradas [2]. Para hacer uso de una entrada no validada, un atacante necesita cumplir con dos objetivos: insertar un dato no confiable y manipular la aplicación usando ese dato. Algunos métodos comunes para insertar datos no confiables son:

- *Manipulación de parámetros*: Pasar valores especiales en campos de un formulario HTML.
- *Manipulación de URL*: Usar parámetros especiales que son enviados a la aplicación web como parte de la URL.
- *Manipulación de campos ocultos*: Insertar valores no confiables en campos ocultos de formularios HTML en páginas web.
- *Manipulación de cabeceras HTTP*: Manipular partes de peticiones HTTP enviadas a la aplicación.

Algunos métodos comunes para manipular aplicaciones con datos no confiables son:

- *Inyección de SQL*: Pasar datos que contienen comandos SQL a un servidor de base de datos para su ejecución.
- *Cross-site scripting*: Hacer uso de aplicaciones que muestran entradas no validadas para convencer a los usuarios de ejecutar código no confiable.
- *Salto de directorio*: Hacer uso de entradas no validadas para controlar qué archivos son accedidos en el servidor.
- *Inyección de comandos*: Usar la entrada de usuario para ejecutar comandos.

El objetivo del presente trabajo es detectar este tipo de vulnerabilidades haciendo uso de un análisis basado en flujo de información.

1.2. Flujo de información

Una política de seguridad de *flujo de información* restringe el uso de la información de un sistema de computación, aún a medida que la información es transformada durante la computación.

Si un usuario desea mantener cierta información confidencial, puede indicar una política en la cual ningún dato visible por otros usuarios es afectado por información confidencial. Esta política permite a los programas manipular y modificar información privada, siempre y cuando esa información no sea revelada por medio de las salidas visibles de esos programas. Una política

de este tipo se conoce como una *política de no interferencia*, porque afirma que los datos confidenciales no pueden interferir con los datos públicos.

Se asume que un atacante o usuario no autorizado puede ver información que no es confidencial, es decir, pública. El método que se suele usar para mostrar que se cumple la no interferencia es demostrar que el atacante no puede observar ninguna diferencia entre dos ejecuciones de un programa que difieren sólo en la información confidencial utilizada.

Los mecanismos para el envío de información a través de un sistema de computación se conocen como *canales*. Reciben el nombre de *canales ocultos* aquellos que explotan un mecanismo cuyo propósito principal no es la transferencia de información. Estos últimos plantean el mayor desafío en la prevención de filtración de información sensible [4]. Existen varias categorias de canales ocultos:

- *Flujos implícitos*: Envían información a través de la estructura de control de un programa.
- *Canales de terminación*: Envían información a través de la terminación o no terminación de una computación.
- *Canales de tiempo*: Envían información a través del tiempo en que una acción ocurre y no a través de los datos relacionados con la acción. La acción podría ser la terminación de un programa, es decir, información sensible se puede obtener del tiempo total de ejecución de un programa.
- *Canales probabilísticos*: Envían información por medio del cambio de la distribución de probabilidad de datos observables. Estos canales son peligrosos cuando el atacante puede ejecutar repetidas veces una computación y observar sus propiedades estocásticas.
- *Canales de agotamiento de recursos*: Envían información por medio del posible agotamiento de un recurso finito compartido como ser la memoria o el espacio en disco.
- *Canales de energía*: Envían información a través de la energía consumida por la computadora, asumiendo que el atacante puede medir este consumo.

Los canales ocultos que requieren atención dependen de lo que los atacantes pueden observar del sistema de computación. Por ejemplo, los canales de energía son importantes para las tarjetas inteligentes, porque deben obtener su energía de la terminal no confiable en la que se insertan. Un programa que es seguro en una computadora abstracta sin requerimientos de energía puede ser parte de un sistema inseguro más grande cuando es ejecutado en una computadora real. Así, se puede decir que un sistema de computación protege información confidencial sólo respecto a un modelo de lo que atacantes y usuarios pueden observar de su ejecución.

1.3. Contribución

El presente trabajo de grado se desarrolló en el marco del proyecto *Plataforma híbrida de seguridad para protección de aplicaciones web* con código NA 080/04 para la Agencia Nacional de Promoción Científica y Tecnológica, entre la empresa Core SA y el laboratorio de investigación LIFIA. Core SA es una empresa especializada en desarrollos y consultoría en tecnologías de seguridad informática. Cuenta con un historial de éxitos en el desarrollo de productos especializados patentados internacionalmente, con un amplio mercado en el exterior, así como una amplia experiencia en consultoría de seguridad y auditoría de código fuente.

El resultado de este trabajo es la aplicación de una técnica de análisis estático de flujo de información a Java [5] con la finalidad de detectar vulnerabilidades en aplicaciones web. Se buscó un enfoque práctico para analizar código legacy Java sin la necesidad de anotaciones por parte de los usuarios (a diferencia de otros análisis [6, 7]).

La técnica de análisis de flujo de información implica realizar un análisis de un programa con el objetivo de probar que no hay filtración de información sensible. En la actualidad existen numerosos estudios teóricos pero poco uso práctico sobre flujo de información. Estos estudios, en general, han sido realizados sobre lenguajes reducidos. El objetivo del presente trabajo es analizar programas escritos en el lenguaje Java, que es uno de los lenguajes más usados para el desarrollo de aplicaciones web.

Se obtuvo además un prototipo que implementa el análisis. La interacción con el grupo encargado de la implementación del prototipo permitió mejorar el análisis y eliminar posibles errores.

1.4. Estructura del informe

En el capítulo 2 se explica en detalle el análisis basado en flujo de información diseñado para detectar vulnerabilidades en aplicaciones web escritas en el lenguaje Java. Se introducen los conceptos básicos del análisis, el lenguaje objeto y se explica cómo se analiza cada construcción del lenguaje. La descripción del análisis de las expresiones y comandos está acompañada de ejemplos para una mejor comprensión.

El capítulo 3 presenta el prototipo que implementa el análisis descripto en el capítulo 2. En la primera sección se describen detalles relacionados con la implementación del prototipo y en la segunda sección se mencionan aplicaciones que fueron probadas con el mismo.

El capítulo 4 concluye el presente informe de trabajo de grado mencionando los resultados obtenidos en el mismo, trabajos futuros y trabajos relacionados.

Capítulo 2

Análisis

En este capítulo se explica en detalle el análisis basado en flujo de información diseñado para detectar vulnerabilidades en aplicaciones web escritas en el lenguaje Java. Se introducen los conceptos básicos del análisis, el lenguaje objeto y se explica cómo se analiza cada construcción del lenguaje. La descripción del análisis de las expresiones y comandos está acompañada de ejemplos para una mejor comprensión.

2.1. Estados de ejecución y de análisis

El análisis de seguridad diseñado se basa en *ejecución simbólica*, un paradigma de análisis estático en el cual se simula la ejecución del programa analizado usando valores simbólicos en vez de valores concretos en las variables del programa [8].

En nuestro análisis, el efecto de analizar una expresión o comando del lenguaje, considerando el flujo de información, se modela a través de cambios en una representación simbólica apropiada del estado de ejecución. Esta representación incluye el ambiente de variables locales (parámetros formales y variables locales), el ambiente de variables estáticas y una heap simbólica. A cada nombre en los ambientes de variables locales y estáticas se le asigna un valor de seguridad o *etiqueta de seguridad*.

Una etiqueta de seguridad es un *nivel de seguridad*[1] o una *referencia simbólica*.

Etiqueta de seguridad	σ	::=	l	*Nivel de seguridad*
		|	(A, C, l)	*Referencia simbólica*
Nivel de seguridad	l	::=	U	*Confiable*
		|	T	*No confiable*

Las referencias simbólicas modelan la creación dinámica de objetos. Incluyen un conjunto de nombres de referencias simbólicas (o para acortar, referencias simbólicas) A, un nombre de clase C y un nivel de seguridad l. C modela la clase del objeto que reside en la heap y es referenciado por A. Por abuso de notación, cuando mencionamos el nivel de seguridad de un objeto con el valor de seguridad (A, C, l) nos referimos al nivel de seguridad l.

A continuación se detalla el estado de ejecución simbólico o *estado de memoria* mencionado anteriormente. Un estado de memoria, escrito $\mathcal{M}$, es una tupla $(\Gamma; H; S)$ donde

- Γ es un *ambiente de variables locales* que mapea variables locales a etiquetas de seguridad:

$$\Gamma : x \mapsto \sigma$$

[1] En la literatura se usa la palabra Untainted para referirse a datos confiables y Tainted para datos no confiables.

- H es una *heap simbólica* que se representa como un mapeo de nombres de referencias simbólicas a *estados de objeto*. Un estado de objeto es una lista de listas de campos, una por cada clase y superclase del objeto. Cada campo tiene asignado una etiqueta de seguridad:

$$H : a \mapsto [(\{C_j.f_i : \sigma_i\}^{i \in 1..n_j})_{j \in 1..k}]$$

  donde k es la cantidad de clases C_j entre la clase del objeto y sus superclases, y n_j la cantidad de campos de la clase C_j.

- S es un *ambiente de variables estáticas* que mapea variables estáticas a etiquetas de seguridad:

$$S : x \mapsto \sigma$$

El comportamiento de seguridad de los métodos requiere que se disponga de información adicional *antes* y *después* de la ejecución simbólica de modo de especificar adecuadamente cómo la ejecución de una construcción modifica el estado de ejecución. La información requerida antes de la ejecución simbólica recibe el nombre *pre-estado de análisis*, y la información requerida después *pos-estado de análisis*. Ambos, el pre-estado de análisis y el pos-estado de análisis, incluyen un estado de memoria. Los atributos adicionales, aparte del estado de memoria, requeridos para el pre-estado de análisis son:

- El nombre E de la clase host del método que está siendo analizado.

- Una pila para los comandos break puros (i.e. comandos break sin etiqueta), cuya necesidad se explica más adelante, dada por la siguiente gramática BNF:

$$B ::= \emptyset \mid B \cdot (\Gamma; H; S; l)$$

- Un estado de memoria y nivel de seguridad por cada etiqueta de un break etiquetado:

$$BL : \mathit{etiqueta} \mapsto (\Gamma; H; S; l)$$

- Una pila para los manejadores de excepciones:

$$\mathcal{C} ::= \emptyset \mid \mathcal{C} \cdot [(E_i, c_i)]$$

- El nivel de seguridad l del contador de programa. Representa el nivel de seguridad del contexto y es un elemento clave para controlar flujos de información implícitos (ver explicación del análisis de un condicional en la sección 2.5).

$\mathcal{M}$	Estado de memoria $(\Gamma; H; S)$
Γ	Ambiente de variables locales
H	Heap simbólica
S	Ambiente de variables estáticas
E	Nombre de la clase host
B	Estructura para breaks puros
BL	Estructura para breaks etiquetados
$\mathcal{C}$	Estructura para manejadores de excepciones
l	Nivel de seguridad del contador de programa
$\mathcal{R}$	Estructura para los valores de retorno
$\mathcal{E}$	Estructura para excepciones

Figura 2.1: Elementos del análisis

Los atributos adicionales, aparte del estado de memoria, requeridos para el pos-estado de análisis son:

- El estado de memoria[2] de un método al punto de retorno de la llamada junto con la etiqueta de seguridad del valor de retorno más sensible:

$$\mathcal{R} ::= \emptyset \mid (H; S) \mid (H; S; \sigma)$$

  (σ no está presente en métodos que no devuelven un valor.)

- Para métodos que pueden terminar con una excepción, el estado de memoria y etiqueta de seguridad de cada excepción E que declara el método:

$$\mathcal{E} : E \mapsto (H; S; \sigma)$$

La figura 2.1 resume los elementos del análisis descriptos en esta sección.

2.2. Esquemas de análisis y tablas de seguridad

Por cada expresión y comando en el lenguaje objeto definimos un esquema de análisis que indica cómo su ejecución afecta los estados de ejecución y de análisis (ver sección 2.4). Por lo tanto, hay dos juicios principales, uno para expresiones y uno para comandos. El *juicio de análisis para expresiones* toma la siguiente forma:

$$\underbrace{E; \mathcal{M}_1; l}_{Pre-estado} \vdash \underbrace{e}_{Expresión} \Rightarrow_e \underbrace{\mathcal{M}_2; \sigma}_{Pos-estado}$$

[2]De hecho, sólo las variables estáticas y la heap.

Esto se puede leer de la siguiente forma: dado el pre-estado de análisis que consiste de la clase host E, el estado de memoria $\mathcal{M}_1$ y el nivel de contador de programa l, luego de la ejecución de la expresión e se obtiene el pos-estado de análisis. El pos-estado consiste de un estado de memoria $\mathcal{M}_2$ junto con la etiqueta de seguridad σ del valor de la expresión.

A continuación se da un ejemplo (los esquemas de análisis se describen en detalle en la sección 2.5.2).

(E-AsignacionCampo)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ E; \mathcal{M}_2; l_1 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{M}_3; \sigma \\ \textsc{Buscar}(\textsc{Tipo}(e_1), f) = D \end{array}}{E; \mathcal{M}_1; l_1 \vdash e_1.f = e_2 \quad \Rightarrow_e \quad \Gamma_3; H_3[A.D.f := \sigma]; S_3; \sigma}$$

Este esquema describe el comportamiento de seguridad de una expresión que es una asignación a un campo. Primero, dado el pre-estado de análisis, la expresión e_1 se analiza obteniendo un pos-estado de análisis que consiste del estado de memoria $\mathcal{M}_2$ y la etiqueta de seguridad (A, C, l_2) del valor de e_1. Luego, se analiza e_2 con el estado de memoria $\mathcal{M}_2$ para obtener el pos-estado $\mathcal{M}_3$ y la etiqueta de seguridad σ. Se busca con la función Buscar la clase que define el campo f a partir de la clase estática de e_1 ($\textsc{Tipo}(e_1)$). Finalmente, se actualiza la heap simbólica, por cada nombre de referencia en A, con la nueva etiqueta de seguridad para el campo f, que en este caso es σ. Para esto último se utiliza la notación $H_3[A.D.f := \sigma]$.

El *juicio de análisis para comandos* toma la siguiente forma:

$$\underbrace{E; \mathcal{M}_1; B_1; BL_1; \mathcal{C}; l}_{Pre-estado} \vdash \underbrace{c}_{Comando} \quad \Rightarrow_c \quad \underbrace{\mathcal{M}_2; B_2; BL_2 \mid \mathcal{R} \mid \mathcal{E}}_{Pos-estado}$$

El pre-estado de análisis de comandos es similar al de expresiones. Una de las diferencias es que además requiere la información para comandos de break. Para break puros se tiene B que toma la forma de pila de tuplas, cada una de las cuales incluye un estado de memoria y un nivel de seguridad. La información para breaks etiquetados se mantiene en BL. Otra diferencia es que el pre-estado tiene a $\mathcal{C}$ para referirse a los manejadores de excepciones.

El pos-estado de análisis incluye, además de un estado de memoria,

- La pila de comandos break y ambiente de breaks etiquetados modificados.
- La etiqueta de seguridad del valor de retorno más sensible y de forma similar una etiqueta de seguridad por cada excepción declarada (y no manejada) en el método siendo analizado.

Principalmente para tratar con métodos recursivos definidos por el usuario pero también para tratar con métodos de bibliotecas se requiere un conjunto de *tablas de seguridad*. Estas tablas especifican cómo se comporta cada

método desde el punto de vista de la seguridad. Dada la información de entrada de un método, las tablas indican cómo el código del método afecta el estado de análisis. Nuestro análisis se puede ver como un sistema cuyo objetivo es computar dichas tablas.

Vamos a especificar cómo estas tablas y los esquemas de análisis se relacionan entre sí. Esencialmente, los esquemas de análisis computan las entradas de estas tablas. Si escribimos $\textsc{Metodos}(C)$ para el conjunto de métodos declarados en la clase C y $\mathbf{C}$ para el conjunto de todas las clases en el sistema, entonces la especificación mencionada se puede indicar de la siguiente forma:

$$\mu\mathcal{T}.\forall C \in \mathbf{C}.\forall m(\overline{t\,x})\ \texttt{throws}\ \overline{E}\ \{\,c\,\} \in \textsc{Metodos}(C).$$
$$E; \mathcal{M}_1; \emptyset; \emptyset; \mathcal{C}; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2; \emptyset; \emptyset \mid \mathcal{R} \mid \mathcal{E}$$
$$\text{y}$$
$$\mathcal{T}(C.m[\overline{t}])(\mathcal{M}_1)(l) = (H_{\mathcal{R}}, S_{\mathcal{R}}, \sigma_{\mathcal{R}}, \mathcal{E})$$

donde $\mathcal{R} = (H_{\mathcal{R}}; S_{\mathcal{R}}; \sigma_{\mathcal{R}})$.

Esto dice que estamos interesados en el conjunto *mínimo* de tablas de seguridad $\mathcal{T}$ tal que para cada método en el sistema, dado cualquier preestado de análisis, el resultado de analizar su código produce un pos-estado de análisis que coincide con la correspondiente entrada en la tabla para este método. Desde un punto de vista operacional, nuestro análisis computa $\mathcal{T}$ a través de progresivas iteraciones hasta alcanzar un punto fijo, excepto por las entradas relacionadas con métodos de bibliotecas cuya información se carga desde una base de datos antes de que el análisis comience. La base de datos con información de seguridad relacionada con métodos de bibliotecas es necesaria dado que al no tener el código fuente de los métodos de las bibliotecas no es posible analizarlos.

Algunos esquemas tienen información de análisis adicional, que se muestra en un recuadro, para la creación de entradas en la tabla de seguridad del método que está siendo analizado. Esta información sobre la generación de tablas de seguridad se explica en la sección 2.6.

2.3. Análisis basado en supremos y basado en ramas de ejecución

Consideremos el siguiente código:

```
int x = 0;
if (e)
  x = valor_secreto;
else
  x = valor_publico;
```

En un análisis estándar (basado en tipos u otro) de flujo de información la etiqueta de seguridad asignada a `x` en el punto en que las ramas then y else convergen es T(no confiable). Este acercamiento conservativo se basa en el hecho de que no se tiene conocimiento en tiempo de compilación de cuál rama va a ser ejecutada. Con las referencias simbólicas se presenta una situación similar en el siguiente código:

```
1   C x = null;
2   if (e)
3     x = new D();
4   else
5     x = new E();
```

Nuestra ejecución simbólica asignará a `x` en la línea 3 la etiqueta de seguridad $(\{a\}, D, \mathsf{U})$ y a `x` en la línea 5 la etiqueta de seguridad $(\{b\}, E, \mathsf{U})$ siendo a y b nombres de referencias nuevos. La pregunta que aparece es qué etiqueta se asigna a `x` luego del comando if. Para esto, diseñamos someramente dos posibles enfoques:

- El enfoque *basado en supremos*, en el que se toma el supremo de los valores de seguridad en ambas ramas de un comando if, y se mantiene un solo estado de ejecución. En este enfoque se asume la siguiente hipótesis: el comportamiento de seguridad de métodos sobreescritos por una subclase es idéntico al de la superclase. Esta hipótesis es similar a la condición estándar impuesta en la mayoría de los lenguajes orientados a objetos que establece que el tipo de un método sobreescrito no puede cambiar el tipo del método en la superclase (porque si no, el chequeo estático de polimorfismo de subtipos no es posible[3]). Bajo esta asunción, en el último ejemplo a `x` se le asignaría la etiqueta $(\{a, b\}, C, \mathsf{U})$. Notar que ahora esta referencia es del tipo declarado para `x`, es decir, C. El nivel de seguridad U resulta de tomar el supremo de los niveles de seguridad de cada referencia (de manera similar al primer ejemplo).

- El enfoque *basado en ramas de ejecución*, en el que se consideran todos los posibles estados que se pueden dar en la ejecución. Es más preciso y permite conocer el camino que tiene que seguir un atacante para aprovechar una vulnerabilidad. Además, no requiere la previa hipótesis invariante, permitiendo un análisis más flexible. La desventaja que tiene es que por cada rama de ejecución posible requiere un análisis por separado.

Decidimos hacer el análisis basado en supremos pensando en la practicidad de un prototipo que implementa el análisis, dado que la implementación de este enfoque requiere menos tiempo y espacio para ejecutarse.

[3] Java 1.5 permite cambios en los tipos de métodos sobreescritos siempre que los tipos de los argumentos sean más generales y el tipo de retorno más específico.

2.4. Sintaxis del lenguaje objeto

Esta sección presenta la gramática BNF del lenguaje a analizar. Las construcciones elegidas luego de estudiar diversos factores de usabilidad y las posibilidades del estado del arte en cuanto a su análisis, son las expresadas por la categoría sintáctica DC de la siguiente gramática (de la que se presentan sólo las partes principales por simplicidad en el informe – esto es, no se dan definiciones para nombres de clases, C, interfaces, I, excepciones, E, y otras construcciones):

Definición de clase

DC ::= `class` C [`extends` C] [`implements` $\overline{I}$] $\{\overline{f}\ \ \overline{M}\}$

Método

M ::= $t\ m(\overline{t\ x})$ [`throws` $\overline{E}$] $\{c\}$

Comando

c	::=	$x = e$	|	$x.f = e$
	|	`this`$.f = e$	|	`super`$.f = e$
	|	$e.m(\overline{e})$	|	`super`$.m(\overline{e})$
	|	`if` (e) c [`else` c]	|	$[et:]$`while`$(e)c$
	|	`do` c `while` (e)	|	`decl` $x = e$
	|	$c; c$	|	`break` $[et]$
	|	`continue`	|	`for` $(\overline{e};\ e;\ \overline{e})$ c
	|	`switch`$(e)\{\overline{\texttt{case}\ e : c}$ [`default` $: c]\}$	|	$\{c\}$
	|	`return` e	|	`throw` e
	|	`try` c $\overline{\texttt{catch}\ (E\ e)\ c}$ [`finally` c]	|	`assert` $e[: e]$

Expresión

e	::=	$x = e$	|	$x.f = e$
	|	`this`$.f = e$	|	`super`$.f = e$
	|	$\underline{n}$	|	$\underline{s}$
	|	`this`	|	x
	|	$C.x$	|	$e.f$
	|	`this`$.f$	|	`super`$.f$
	|	$e.m(\overline{e})$	|	`super`$.m(\overline{e})$
	|	`decl` $x = e$	|	$e\ op\ e$
	|	$e\ op$	|	$op\ e$
	|	$(C)\ e$	|	`null`
	|	e `instanceOf` C	|	`new` $C(\overline{e})$
	|	$e\ ?\ e : e$		

Algunas de las características del lenguaje que decidimos no abordar, por ser lo suficientemente complejas como para requerir un estudio por separado, son *concurrencia* y *reflection*.

2.5. Análisis basado en supremos

Esta sección describe un subconjunto de los esquemas de análisis para el análisis basado en supremos. El listado completo de los esquemas de análisis se encuentra en el apéndice A. Antes de proceder, necesitamos definir algunas nociones preliminares.

2.5.1. Nociones preliminares

El supremo de dos etiquetas de seguridad ρ y σ que está parametrizado por un nombre de clase C (escrito $\rho \sqcup_C \sigma$) se define como:

$$\begin{aligned} l_1 \sqcup_C l_2 &\stackrel{def}{=} l_1 \sqcup l_2 \\ l_1 \sqcup_C (A, C_2, l_2) &\stackrel{def}{=} (A, C_2, l_1 \sqcup l_2) \\ (A, C_1, l_1) \sqcup_C l_2 &\stackrel{def}{=} (A, C_1, l_1 \sqcup l_2) \\ (A_1, C_1, l_1) \sqcup_C (A_2, C_2, l_2) &\stackrel{def}{=} (A_1 \cup A_2, C, l_1 \sqcup l_2) \end{aligned}$$

El supremo de los ambientes de variables locales Γ_1 y Γ_2 con el mismo dominio (escrito $\Gamma_1 \sqcup \Gamma_2$) se define como:

$\forall x \in \textsc{Dominio}(\Gamma_1)$,

$$(\Gamma_1 \sqcup \Gamma_2)(x) \stackrel{def}{=} \Gamma_1(x) \sqcup_{\textsc{Tipo}(x)} \Gamma_2(x) \quad \text{si } x \text{ es un objeto } \textsc{ProyC}(\Gamma_1(x)) \neq \textsc{ProyC}(\Gamma_2(x))$$

$$(\Gamma_1 \sqcup \Gamma_2)(x) \stackrel{def}{=} \Gamma_1(x) \sqcup_\star \Gamma_2(x) \quad \text{caso contrario}$$

Aquí, $\textsc{Tipo}(x)$ denota el tipo declarado para x y $\textsc{ProyC}(\Gamma(x))$ es la clase del objeto asignado a x.

Una operación de supremo sobre etiquetas de seguridad a la que a veces recurrimos se denota $\rho \sqcup_\star \sigma$ y se define como:

$$\begin{aligned} l_1 \sqcup_\star l_2 &\stackrel{def}{=} l_1 \sqcup l_2 \\ l_1 \sqcup_\star (A, C_2, l_2) &\stackrel{def}{=} (A, C_2, l_1 \sqcup l_2) \\ (A, C_1, l_1) \sqcup_\star l_2 &\stackrel{def}{=} (A, C_1, l_1 \sqcup l_2) \\ (A_1, C, l_1) \sqcup_\star (A_2, C, l_2) &\stackrel{def}{=} (A_1 \cup A_2, C, l_1 \sqcup l_2) \end{aligned}$$

Notar que es aplicable solamente en el caso que ρ o σ sea un nivel de seguridad o bien ambos sean referencias simbólicas y sus nombres de clase sean idénticos.

El supremo de dos ambientes de variables estáticas es análogo al supremo de ambientes de variables locales. Respecto al supremo de heaps simbólicas se procede de una forma similar como se indica a continuación. Sean H_1 y H_2 heaps simbólicas, definimos $H_1 \sqcup H_2$ como:

1. $\forall a \in \text{DOMINIO}(H_1) \cap \text{DOMINIO}(H_2)$, computamos el supremo por cada campo $C_i.f_j$ del objeto referenciado por a:

$$(H_1 \sqcup H_2)(a) \stackrel{def}{=} [C_i.f_j : H_1(a.C_i.f_j) \sqcup_{\text{TIPO}(H_1(a.C_i.f_j))} H_2(a.C_i.f_j)]$$

si $H_1(a.C_i.f_j)$ es un objeto
y $\text{PROYC}(H_1(a.C_i.f_j)) \neq \text{PROYC}(H_2(a.C_i.f_j))$

$$(H_1 \sqcup H_2)(a) \stackrel{def}{=} [C_i.f_j : H_1(a.C_i.f_j) \sqcup_\star H_2(a.C_i.f_j)]$$

caso contrario

2. Para los restantes $a \in \text{DOMINIO}(H_1) \cup \text{DOMINIO}(H_2)$, definimos

$$(H_1 \sqcup H_2)(a) \stackrel{def}{=} \begin{cases} H_1(a) & \text{si } a \in \text{DOMINIO}(H_1) \\ H_2(a) & \text{si } a \in \text{DOMINIO}(H_2) \end{cases}$$

El supremo de dos pilas de break y el supremo de dos ambientes de breaks etiquetados se computa por cada valor usando las definiciones dadas arriba. Respecto a la estructura $\mathcal{R}$, definimos su supremo como sigue:

$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_1 \sqcup \mathcal{R}_2$
$\emptyset$	$\mathcal{R}_2$	$\mathcal{R}_2$
$\mathcal{R}_1$	$\emptyset$	$\mathcal{R}_1$
$(H_1; S_1; \sigma_1)$	$(H_2; S_2; \sigma_2)$	$(H_1 \sqcup H_2; S_1 \sqcup S_2; \sigma_1 \sqcup_\star \sigma_2)$
$(H_1; S_1; \sigma_1)$	$(H_2; S_2; \sigma_2)$	$(H_1 \sqcup H_2; S_1 \sqcup S_2; \sigma_1 \sqcup_{C_{retorno}} \sigma_2)$

La cuarta línea considera el caso en que se devuelve un objeto y las clases de σ_1 y σ_2 no coinciden. $C_{retorno}$ es el tipo de retorno del método que está siendo analizado.

La definición de $\mathcal{E}_1 \sqcup \mathcal{E}_2$ es similar a la de $\mathcal{R}_1 \sqcup \mathcal{R}_2$, con la diferencia de que se hace el supremo por cada excepción declarada por el método.

2.5.2. Esquemas de análisis para expresiones

Recordemos que el juicio de análisis para expresiones toma la siguiente forma:

$$\boxed{E; \mathcal{M}_1; l \vdash e \qquad \Rightarrow_e \qquad \mathcal{M}_2; \sigma}$$

El siguiente juicio de análisis auxiliar se usa para secuencias de expresiones.

$$\boxed{E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \qquad \Rightarrow_{\overline{e}} \qquad \mathcal{M}_2; (\sigma_1, \ldots, \sigma_n)}$$

Y se define por los siguientes dos esquemas. El primero pertenece a una secuencia no vacía de expresiones mientras que el segundo se aplica a secuencias vacías.

$$\frac{E; \mathcal{M}_1; l \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma_1 \qquad E; \mathcal{M}_2; l \vdash (e_2, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_3; (\sigma_2, \ldots, \sigma_n)}{E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_3; (\sigma_1, \sigma_2, \ldots, \sigma_n)}$$

$$\frac{}{E; \mathcal{M}; l \vdash () \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}; ()}$$

- **Asignación.** Hay dos esquemas para una asignación $x = e$ dependiendo si x se refiere a una variable local o estática. Estos esquemas simplemente actualizan el ambiente correspondiente con el valor de seguridad que resulta del análisis de e. Quizás vale aclarar que este valor tiene al menos el nivel l del contador de programa.

  (E-Asignacion)

$$\frac{x \text{ es una variable local o parámetro} \qquad E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \Gamma_2[x := \sigma]; H_2; S_2; \sigma}$$

  (E-Asignacion-Estatica)

$$\frac{x \text{ es estática} \qquad E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \Gamma_2; H_2; S_2[x := \sigma]; \sigma}$$

  Las variables locales a un método sólo existen durante la ejecución del mismo, por esto, no se consideran en las tablas de seguridad. El caso de las variables estáticas sí se considera porque existen independientemente del método.

  Para la asignación de una variable estática x se agrega la variable en la estructura S_{out} para indicar su modificación. Si es la primera vez que se lee esta variable ($x \notin S_{in}$) se guarda su valor en S_{in}.

$$\frac{x \text{ es estática} \qquad \mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2}{\mathcal{T}_1 \vdash x = e \quad \Rightarrow_e \quad \mathcal{T}_2 \;\; [S_{out} \cup = \; \{x\}] \;\; [S_{in}(x) = S(x)] \quad (x \notin S_{in})}$$

- **Asignación de campo.** A diferencia de la asignación de una variable local o estática, la asignación de un campo $e_1.f = e_2$ implica actualizar la heap simbólica dado que los objetos residen en la heap.

  Primero, se analiza la expresión e_1 que representa un objeto con campo f, obteniendo la referencia A para actualizar la heap. Luego, el análisis de e_2 nos da el nuevo valor de seguridad para el campo f.

  En la cadena formada por la clase de e_1 y sus superclases podría haber varias definiciones de f. Para saber cuál f se actualiza, Java realiza una búsqueda estática, es decir, a partir de la clase del tipo asignado a e_1 en tiempo de compilación. Para obtener el tipo que se le asigna a e_1 en tiempo de compilación usamos la función Tipo. La búsqueda de la clase que define el campo la realizamos con la función Buscar.

  La actualización $H[a.D.f := \sigma]$ para cada referencia $a \in A$ la escribimos con la notación más corta $H[A.D.f := \sigma]$.

  (E-AsignacionCampo)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ E; \mathcal{M}_2; l_1 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{M}_3; \sigma \\ \textsc{Buscar}(\textsc{Tipo}(e_1), f) = D \end{array}}{E; \mathcal{M}_1; l_1 \vdash e_1.f = e_2 \quad \Rightarrow_e \quad \Gamma_3; H_3[A.D.f := \sigma]; S_3; \sigma}$$

  Cuando se modifica un campo, el cambio se ve fuera del método sólo si la referencia al objeto que tiene el campo sigue existiendo fuera del método ($A \in \Gamma_{in} \cup H_{in} \cup S_{in}$). En este caso, se guarda en H_{out} la posición $A.D.f$ que se actualizó. Además, si es la primera vez que se analiza este campo en el método ($A.D.f \notin H_{in}$) se guarda su valor de seguridad en H_{in}.

  El siguiente es un ejemplo en el que el campo f no es visible fuera del método:

```
void m() {
  C x = new C();
  x.f = 0;
}
```

$$\frac{\begin{array}{c}\mathcal{T}_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{T}_2; (A, C, l_2) \\ \mathcal{T}_2 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{T}_3 \\ \textsc{Buscar}(\textsc{Tipo}(e_1), f) = D\end{array}}{\begin{array}{l}\mathcal{T}_1 \vdash e_1.f = e_2 \quad \Rightarrow_e \quad \mathcal{T}_3 \ [H_{out} \cup = \ \{A.D.f\}] \\ \qquad\qquad\qquad\qquad\quad (A \in \Gamma_{in} \cup H_{in} \cup S_{in}) \\ \\ \qquad\qquad\qquad\qquad\quad [H_{in}(A.D.f) = H(A.D.f)] \\ \qquad\qquad\qquad\qquad\quad (A \in \Gamma_{in} \cup H_{in} \cup S_{in} \\ \qquad\qquad\qquad\qquad\quad \wedge \\ \qquad\qquad\qquad\qquad\quad A.D.f \notin H_{in})\end{array}}$$

- **Literal.** En el caso de un literal de un tipo primitivo, el estado de memoria permanece igual y el valor de seguridad de la expresión es el del contador de programa.

$$\frac{}{E; \mathcal{M}; l \vdash \underline{p} \quad \Rightarrow_e \quad \mathcal{M}; l} \text{ E-Primitivo}$$

Cuando tenemos un literal que representa un objeto como es el caso de un string, creamos un objeto nuevo de la clase correspondiente. El esquema para un string es el siguiente:

$$\frac{E; \mathcal{M}_1; l \vdash \texttt{new } String() \quad \Rightarrow_e \quad \mathcal{M}_2; (\{a\}, String, l)}{E; \mathcal{M}_1; l \vdash \underline{s} \quad \Rightarrow_e \quad \mathcal{M}_2; (\{a\}, String, l)} \text{ E-String}$$

- **Variable.** El valor de seguridad de una variable es el almacenado en el ambiente apropiado y actualizado para incluir el nivel del contador de programa. Hay dos esquemas dependiendo si la variable es estática o no.

Para la tabla de seguridad, se verifica si es la primera vez que se lee x, en cuyo caso se agrega a Γ_{in} o S_{in} según corresponda.

$$\frac{x \text{ es una variable local o parámetro}}{E; \mathcal{M}; l \vdash x \quad \Rightarrow_e \quad \mathcal{M}; \Gamma(x) \sqcup l} \text{ E-Variable}$$

$$\frac{x \text{ es un parámetro}}{\mathcal{T} \vdash x \quad \Rightarrow_e \quad \mathcal{T}[\Gamma_{in}(x) = \Gamma(x)] \quad (x \notin \Gamma_{in})}$$

$$\frac{x \text{ es estática}}{E; \mathcal{M}; l \vdash x \quad \Rightarrow_e \quad \mathcal{M}; S(x) \sqcup l} \text{ E-Variable-Estatica}$$

$$\frac{x \text{ es estática}}{\mathcal{T} \vdash x \quad \Rightarrow_e \quad \mathcal{T}[S_{in}(x) = S(x)] \quad (x \notin S_{in})}$$

- **Acceso a campo.** Hay dos esquemas dependiendo del receptor del acceso a campo, que puede ser *super* o una expresión e diferente de *super*.

  Si tenemos la expresión $e.f$, primero analizamos e para obtener la referencia A. Como mencionamos antes, la búsqueda del campo f se hace de forma estática. Luego, se devuelve el supremo de los valores de cada referencia ($\bigsqcup_{a \in A} H(a.D.f)$) que lo notamos de forma corta $H(A.D.f)$, actualizado con el nivel de seguridad de la referencia al objeto e.

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ \textsc{Buscar}(\textsc{Tipo}(e), f) = D \end{array}}{E; \mathcal{M}_1; l_1 \vdash e.f \quad \Rightarrow_e \quad \mathcal{M}_2; H_2(A.D.f) \sqcup l_2} \text{ E-}\textsc{Campo}$$

  Para la tabla de seguridad se verifica si es la primera vez que se lee el campo.

$$\frac{\begin{array}{c} \mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2; (A, C, l_2) \\ \textsc{Buscar}(\textsc{Tipo}(e), f) = D \end{array}}{\mathcal{T}_1 \vdash e.f \quad \Rightarrow_e \quad \mathcal{T}_2 \ [H_{in}(A.D.f) = H(A.D.f)] \quad (A.D.f \notin H_{in})}$$

- **Invocación a método.** Existen tres esquemas para analizar un método dependiendo del receptor, que puede ser una expresión e que representa un objeto, *super* o una clase C. Cuando el receptor es una clase tenemos un método estático.

  Cuando tenemos una expresión e que representa un objeto al que se le envía un mensaje m, se analiza primero esta expresión para obtener su valor de seguridad que es el que corresponde al parámetro *this*. Luego, se obtienen los valores de seguridad de los parámetros reales. El método m se busca dinámicamente a partir de la clase del receptor subiendo por sus superclases. Esto es porque Java tiene binding dinámico para métodos. Observar que para el acceso a campos esto no se cumple. Finalmente, se obtiene a partir de los datos anteriores y la tabla de seguridad del método m el estado de memoria resultante en el caso de terminación normal $(H_4; S_4; \sigma)$ y los estado de memoria que resultan de una terminación excepcional $(H^{E_i}, S^{E_i}, \sigma^{E_i})$. Las E_i

son las excepciones que declara el método m. El valor de seguridad de la invocación está dado por el valor de seguridad del valor de retorno (σ).

Las invocaciones a métodos que no devuelven un valor no pueden ser expresiones. Para estos métodos el σ no está presente.

(E-Invocacion)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ E; \mathcal{M}_2; l_1 \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_3; (\sigma_1, \ldots, \sigma_n) \\ \textsc{Buscar}(C, m) = D \\ \left(\begin{array}{l} (H_4, S_4, \sigma), \\ \\ \{(H^{E_i}, S^{E_i}, \sigma^{E_i})\} \end{array} \right) = \mathcal{T} \begin{array}{l} (D.m[\textsc{Tipo}(e_1), \ldots, \textsc{Tipo}(e_n)]) \\ (\{\ x_i : \sigma_i\ \} \quad \cup \quad \{\ this : (A, C, l_2)\ \} \\ \quad ; H_3 \\ \quad ; S_3) \\ (l_1) \end{array} \end{array}}{E; \mathcal{M}_1; l_1 \vdash e.m(e_1, \ldots, e_n) \quad \Rightarrow_e \quad \Gamma_3; H_4; S_4; \sigma}$$

Cuando estamos analizando un método m_1 y encontramos una invocación a un método m_2, obtenemos la entrada que coincide en la tabla de m_2 para modificar el estado actual de la tabla de m_1. Por ejemplo, si tenemos el código:

```
int m1(A x) {
  ...
  this.m2(x);
  ...
}

void m2(A z) {
  ...
}
```

y la entrada de la tabla de m_2 que coincide con los elementos del análisis al momento de evaluar la invocación:

Γ_{in}	H_{in}	S_{in}	l	H_{out}	S_{out}
...				...	
$z \mapsto (\{\alpha\}, A, \mathsf{U})$		$w \mapsto \mathsf{U}$	U		$w \mapsto \mathsf{T}$
...				...	

haremos las siguientes modificaciones al estado actual de la tabla de m_1:

$$\mathcal{T}[\Gamma_{in}(x) = \Gamma(x)] \quad (x \notin \Gamma_{in})$$

$$\mathcal{T}[S_{out} \cup = \ \{w\}]$$

El primer cambio es para indicar que se lee el parámetro x y el segundo indica que la variable estática w fue modificada.

- **Casting.** A continuación presentamos dos situaciones a contemplar en el análisis de expresiones que involucran un cast. Los casts para tipos primitivos no influyen en el análisis.

  - *Cast válido.* El siguiente esquema se usa cuando un objeto se castea a su propia clase o a una de sus superclases. Habrá un error en tiempo de ejecución si casteamos un objeto a una de sus subclases o a una clase que no tiene relación con el objeto.

$$\frac{\begin{array}{c}\text{C es D o una de sus superclases} \\ E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2)\end{array}}{E; \mathcal{M}_1; l_1 \vdash (C)e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2)} \text{ E-CAST-ARRIBA}$$

  - *Métodos del sistema.* Como los métodos del sistema no son analizados, creamos como valor de retorno un objeto de la clase de retorno. Pero el verdadero resultado del método podría ser un objeto de una subclase de la clase de retorno. Por esto, en la ejecución simbólica del programa podría haber un cast a una subclase del objeto. Por ejemplo:

```
1   m(Vector<String> files) {
2     Iterator iter = files.iterator();
3     while(iter.hasNext()) {
4       String file = (String) iter.next();
5       this.m2(file);
6     }
7   }
8
9   m2(String file) {
10    file = file.trim();
11    ...
12  }
```

    El valor que retorna el análisis para `iter.next()`, un método del sistema, en la línea 4 es un objeto nuevo de la clase `Object`, el tipo de retorno del método. El programa luego castea este objeto a `String` para enviarlo como parámetro al método $m2$.

    Para resolver este problema, creamos un objeto de la clase que se requiere, `String` en este caso, como lo indica el siguiente esquema:

(E-Cast-Abajo)

$$\frac{\begin{array}{c}\text{C es una subclase de D}\\ E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2) \\ E; \mathcal{M}_2; l_1 \vdash \texttt{new } C() \quad \Rightarrow_e \quad \mathcal{M}_3; (\{b\}, C, l_3)\end{array}}{E; \mathcal{M}_1; l_1 \vdash (C)e \quad \Rightarrow_e \quad \mathcal{M}_3; (\{b\}, C, l_3)}$$

2.5.3. Esquemas de análisis para comandos

En esta sección se detallan los esquemas que definen el juicio de análisis para comandos:

$$\boxed{E; \mathcal{M}_1; B_1; BL_1; \mathcal{C}; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2; B_2; BL_2 \mid \mathcal{R} \mid \mathcal{E}}$$

Varios de estos esquemas se basan en los de expresiones. En estos casos omitimos más explicaciones y en cambio remitimos al lector a la explicación dada más arriba para las expresiones correspondientes.

Para simplificar la notación mencionamos B, BL, $\mathcal{C}$, $\mathcal{R}$ y $\mathcal{E}$ sólo en los esquemas que los modifican directamente. Cuando $\mathcal{R}$ y $\mathcal{E}$ no están presentes, sus valores se asumen $\emptyset$.

- **Asignación, asignación de campo, declaración, declaración de arreglo, invocación a método.** El siguiente esquema se usa para comandos que pueden ser expresiones: $x = e$, $x.f = e$, $\texttt{super}.f = e$, $\texttt{decl } x = e$, $\texttt{decl } C[\,]^1...[\,]^n\ x = e$, $e.m(e_1, ..., e_n)$, $\texttt{super}.m(e_1, ..., e_n)$, $C.m(e_1, ..., e_n)$.

$$\frac{E; \mathcal{M}_1; l \vdash c_e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma_1}{E; \mathcal{M}_1; l \vdash c_e \quad \Rightarrow_c \quad \mathcal{M}_2}$$

- **IfThenElse.** El análisis del comando if consiste en obtener el supremo de los valores obtenidos al analizar independientemente cada rama.

  (C-IfThenElse)

$$\frac{\begin{array}{c}E; \mathcal{M}_1; l \vdash e \ \Rightarrow_e \ \mathcal{M}_2; \sigma \\ E; \mathcal{M}_2; B_1; BL_1; l \sqcup \sigma \vdash c_1 \ \Rightarrow_c \ \mathcal{M}_3; B_2; BL_2 | \mathcal{R}_1 | \mathcal{E}_1 \\ E; \mathcal{M}_2; B_1; BL_1; l \sqcup \sigma \vdash c_2 \ \Rightarrow_c \ \mathcal{M}_4; B_3; BL_3 | \mathcal{R}_2 | \mathcal{E}_2\end{array}}{\begin{array}{ll}E; \mathcal{M}_1; B_1; BL_1; l \vdash \texttt{if } (e)\ c_1\ \texttt{else } c_2 \ \Rightarrow_c & (\mathcal{M}_3; B_2; BL_2) \\ & \sqcup\ (\mathcal{M}_4; B_3; BL_3) \\ & |\mathcal{R}_1 \sqcup \mathcal{R}_2 | \mathcal{E}_1 \sqcup \mathcal{E}_2\end{array}}$$

  Consideremos el siguiente ejemplo:

```
if (secreto)
  x = 1;
else
  x = 0;
```

El nivel del contador de programa en las ramas del if es alto debido a que la condición está relacionada con un dato secreto. De este modo, el análisis asigna un valor de seguridad T a la variable x. Esto refleja el hecho de que la variable x pasa a no ser confiable porque su valor (0 o 1) permite a un atacante saber el valor de la variable *secreto* ($falso$ o *verdadero*).

- **Secuencia.** En una secuencia se analizan los comandos respetando el orden. Además, los valores de seguridad de retorno y de las excepciones se obtienen realizando el supremo de los valores obtenidos en el análisis de cada comando.

  (C-Secuencia)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; B_1; BL_1; l \vdash c_1 \ \Rightarrow_c \ \mathcal{M}_2; B_2; BL_2 | \mathcal{R}_1 | \mathcal{E}_1 \\ E; \mathcal{M}_2; B_2; BL_2; l \vdash c_2 \ \Rightarrow_c \ \mathcal{M}_3; B_3; BL_3 | \mathcal{R}_2 | \mathcal{E}_2 \end{array}}{E; \mathcal{M}_1; B_1; BL_1; l \vdash c_1;\ c_2 \ \Rightarrow_c \ \mathcal{M}_3; B_3; BL_3 | \mathcal{R}_1 \sqcup \mathcal{R}_2 | \mathcal{E}_1 \sqcup \mathcal{E}_2}$$

- **Break.** Tenemos dos esquemas para el comando break. El primero simplemente registra el estado de memoria actual (i.e. el que existe en el punto en que el break es ejecutado).

$$\frac{}{E; \mathcal{M}; B \cdot b; l \vdash \texttt{break} \qquad \Rightarrow_c \qquad \emptyset; B \cdot (b \sqcup (\mathcal{M}, l))} \text{ C-Break}$$

  En el caso de un break con etiqueta debemos registrar el estado de memoria actual para esa etiqueta en particular.

  (C-Break-Et)

$$\frac{}{E; \mathcal{M}; BL; l \vdash \texttt{break}\ et \qquad \Rightarrow_c \qquad \emptyset; BL[et := (BL(et) \sqcup (\mathcal{M}, l))]}$$

  El siguiente ejemplo muestra por qué necesitamos una pila para considerar comandos `break` dentro de comandos `while`:

```
j = 0;
while (j < 2) {
  j ++;
  i = 0;
  while (i < 2) {
    i ++;
```

```
    if (secreto)
      break;
  }
}
// i vale 1 si secreto es true
```

En este ejemplo el nivel de seguridad del break en el while de más adentro no afecta la variable j en el while de más afuera.

El siguiente ejemplo muestra cómo `break` afecta el nivel de seguridad de las variables escritas en el cuerpo del comando while:

```
y = 5;
x = 0;
while (y > 0) {
  y = y - 1;
  if (secreto)
    break;
  x = x + 1;
}
// x vale 5 si secreto es false
// x vale 0 si secreto es true
```

A continuación hay un ejemplo de `break` usado con *etiqueta*, en este caso los niveles de seguridad de las variables de ambos cuerpos son afectados:

```
      i = 0;
etiq: while (i < 2) {
        i ++;
        j = 0;
        while (j < 2) {
          if (secreto)
            break etiq;
          j ++;
        }
      }
      \\ i vale 1 y j vale 0 si secreto es true
```

- **Return.** Para el comando return se analiza la expresión que contiene y se devuelve como valor de seguridad de retorno el valor de esa expresión (σ) y el estado de memoria resultante sin incluir el ambiente de variables locales (H_2, S_2). Como es una salida del método, devuelve un estado de memoria con el valor neutro del supremo de estructuras

($\emptyset$), lo mismo para las excepciones ya que es una salida del método que no es excepcional.

$$\frac{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash \texttt{return}\ e \quad \Rightarrow_c \quad \emptyset \mid (H_2, S_2, \sigma) \mid \emptyset}\ \text{C-RETURN-EXP}$$

Para la información de la tabla de seguridad se considera el σ como uno de los valores de retorno posibles. CAMPOSRET devuelve los campos del objeto retornado y campos de los objetos contenidos dentro de esos campos, y así siguiendo. Estos campos se agregan a H_{out} debido a que son visibles fuera del método.

$$\frac{\mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2; \sigma}{\mathcal{T}_1 \vdash \texttt{return}\ e \quad \Rightarrow_e \quad \mathcal{T}_2\ [\sigma_{return} \sqcup = \ \sigma] \ [H_{out} \cup = \text{CAMPOSRET}()]}$$

- **Try.** El análisis del comando try-catch consiste en analizar el comando del cuerpo (c) con la lista L de pares (E_i, c_i) que se corresponde con los manejadores de excepciones. Esta lista se tiene en cuenta en el comando throw.

  (C-TRY)

$$\frac{E; \mathcal{M}_1; \mathcal{C} \cdot L; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E}}{\begin{array}{l} E; \mathcal{M}_1; \mathcal{C}; l \vdash \texttt{try}\ c \quad \texttt{catch}\ (E_1\ e)\ c_1 \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E} \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad \texttt{catch}\ (E_n\ e)\ c_n \\ \qquad\qquad\qquad\qquad [\texttt{finally}\ c_f] \end{array}}$$

- **Throw.** A continuación se presentan varios esquemas de análisis para el comando throw que consideran los distintos contextos, si hay un manejador para la excepción y si la cláusula finally fue incluida.

  - No hay ningún manejador ni cláusula finally. (C-THROW)

$$\frac{E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2)}{\begin{array}{l} E; \mathcal{M}_1; \emptyset; l_1 \vdash \texttt{throw}\ e \ \Rightarrow_c \ \emptyset \\ \qquad\qquad\qquad\qquad\quad \mid \emptyset \\ \qquad\qquad\qquad\qquad\quad \mid \emptyset[Ex := (H_2, S_2, (A, Ex, l_2))] \end{array}}$$

  - No hay manejador ni cláusula finally en el comando try-catch inmediato. (C-THROW-NC-NF)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ Ex \notin L \\ E; \mathcal{M}_2; \mathcal{C}; l_1 \sqcup l_2 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- No hay manejador pero sí una cláusula finally en el comando try-catch inmediato. La función BuscarFinally devuelve el comando c_f que se encuentra en la cláusula finally.
  (C-Throw-NC-F)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ Ex \notin L \\ c_f = \text{BuscarFinally}(L) \\ E; \mathcal{M}_2; \mathcal{C}; l_1 \sqcup l_2 \vdash c_f;\ \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- Existe un manejador pero no hay cláusula finally en el comando try-catch inmediato. x_{Ex} es la variable que aparece en la cláusula catch de la excepción Ex.
  (C-Throw-C-NF)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ (Ex, c_{Ex}) \in L \\ E; \Gamma_2[x_{Ex} := (A, Ex, l_2)]; H_2; S_2; \mathcal{C}; l_1 \sqcup l_2 \vdash c_{Ex} \ \Rightarrow_c \ \mathcal{M}_3 | \mathcal{R} | \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- Existen ambos un manejador y una cláusula finally en el comando try-catch inmediato.
  (C-Throw-C-F)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ (Ex, c_{Ex}) \in L \\ c_f = \text{BuscarFinally}(L) \\ E; \Gamma_2[x_{Ex} := (A, E, l_2)] \ \vdash c_{Ex};\ c_f \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \\ ; H_2; S_2; \mathcal{C}; l_1 \sqcup l_2 \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

2.6. Tablas de seguridad

Esta sección describe las tablas de seguridad que guardan información sobre el comportamiento de métodos.

2.6.1. Análisis de un método

Para analizar una invocación a un método m se busca en la tabla de seguridad de m para ver cómo se modifica el estado actual de acuerdo al comportamiento de seguridad de m.

Las tablas de seguridad tienen una entrada por cada pre-estado de análisis considerado hasta el momento. Cuando falta una entrada en una tabla

analizamos el método con los esquemas extendidos con información sobre la tabla, presentados en recuadros en la sección 2.5. En estos esquemas extendidos se usan el ambiente de variables locales, el de variables estáticas y la heap simbólica con el nombre *in* para guardar la información que usa el método del pre-estado de análisis. Las mismas estructuras se usan con el nombre *out* para guardar las variables modificadas durante el análisis del método.

Una vez finalizado el análisis de seguridad del método se crea la nueva entrada en la tabla con la información de las estructuras *in* y *out*. Las estructuras *out* guardan el último valor asignado a las variables que fueron modificadas en el método. Las estructuras *in* se usan para buscar un entrada en la tabla de seguridad que coincida con un pre-estado de análisis como se describe en la próxima sección.

A continuación se describe un ejemplo.

```
1   void m (int y) {
2     x.f = 1;
3     x = new E(y);
4     x.f = 2;
5     if (true) {
6       z = new C();
7     }
8     w = z;
9   }
```

Asumimos que x entra con el valor $(\{a\}, A, \mathsf{U})$, $\{a\}.A.f$ con el valor T, y con el valor U, z con el valor $(\{b\}, C, \mathsf{U})$, y w entra con el valor $(\{g\}, C, \mathsf{T})$. Y además, en la línea 3 el objeto creado tiene el valor $(\{p\}, E, \mathsf{U})$, y el objeto creado en la línea 6 tiene el valor $(\{e\}, C, U)$. El nivel del contador de programa es U.

En la línea 2 agregamos por el esquema E-Variable-Estatica $x \mapsto (\{a\}, A, \mathsf{U})$ a S_{in}, y por el esquema E-AsignacionCampo $\{a\}.A.f$ a H_{out} y $\{a\}.A.f \mapsto \mathsf{T}$ a H_{in}. En el próximo comando agregamos $y \mapsto \mathsf{U}$ a Γ_{in} por el esquema E-Variable y x a S_{out} por E-Asignacion-Estatica. En la línea 4 agregamos $\{p\}.E.f$ a H_{out} con el esquema de análisis E-AsignacionCampo. En el comando if agregamos z a S_{out} por el esquema E-Asignacion-Estatica y además agregamos $z \mapsto (\{b\}, C, \mathsf{U})$ a S_{in} porque se lee implícitamente al hacer la operación de supremo sobre ambientes de variables estáticas en el if. En la línea 8 agregamos w a S_{out} y $w \mapsto (\{g\}, C, \mathsf{T})$ a S_{in} por E-Asignacion-Estatica. Y finalmente obtenemos la siguiente tabla llenando la información de las variables en las estructuras *out*:

Γ_{in}	H_{in}	S_{in}	l
...			
$x \mapsto (\{\alpha\}, A, \mathsf{U})$ $y \mapsto \mathsf{U}$	$\{\alpha\}.A.f \mapsto \mathsf{T}$	$z \mapsto (\{\beta\}, C, \mathsf{U})$ $w \mapsto (\{\gamma\}, C, \mathsf{T})$	U
...			

H_{out}	S_{out}
...	
$\{\alpha\}.A.f \mapsto \mathsf{U}$ $\{\pi\}.E.f \mapsto \mathsf{U}$	$x \mapsto (\{\pi\}, E, \mathsf{U})$ $z \mapsto (\{\beta, \eta\}, C, \mathsf{U})$ $w \mapsto (\{\beta, \eta\}, C, \mathsf{U})$
...	

Si se encuentra una vulnerabilidad la nueva entrada en la tabla también guarda esta información.

2.6.2. Comparación de entradas en tablas de seguridad

No necesitaremos analizar un método cuando su tabla tiene la respuesta para el estado de análisis actual. Para esto, tenemos que ver si el estado actual $< \Gamma, H, S, l >_s$ coincide con una entrada $< \Gamma, H, S, l >_t$ de la tabla. La comparación de cada parte de la tupla se hace de la siguiente manera:

- Los niveles de seguridad del contador de programa tiene que ser iguales, es decir, $l_t = l_s$.
- Para Γ usamos la función *match* que en caso de ser iguales los ambientes nos devuelve una sustitución que nos indica cómo reemplazar las variables de nombres de referencias simbólicas que aparecen en la tabla por los nombres de referencia que se corresponden con el estado de análisis actual:

$$\text{match} <>_t <>_s = id$$
$$\text{match} < (x : \mathsf{U}/\mathsf{T}) \cdot \Gamma >_t < (x : \mathsf{U}/\mathsf{T}) \cdot \Gamma >_s = \text{match} < \Gamma >_t < \Gamma >_s$$

$$\text{match} < (x : (\{\phi_1, .., \phi_n\}, C, l)) \cdot \Gamma >_t < (x : (\{b_1, .., b_n\}, C, l)) \cdot \Gamma >_s$$
$$=$$
$$\text{let } Subst = \{\rho \mid \rho(\{\phi_1, \ldots, \phi_n\}) = \{b_1, \ldots, b_n\}\}$$
$$\text{in}$$
$$\quad \text{if } Subst \text{ is empty}$$
$$\quad \text{then } fail$$
$$\quad \text{else}$$
$$\quad\quad \bigcup_{\rho \in Subst} \rho \circ (\text{match} < \rho\Gamma >_t < \Gamma >_s)$$

$$\text{match } \_ \_ = fail$$

donde:

$$\rho \circ fail = fail$$
$$\rho \circ \{\rho_1, \ldots, \rho_k\} = \{\rho \circ \rho_1, \ldots, \rho \circ \rho_k\}$$

y:

$$\phi = \{ \begin{array}{c} \alpha_i \\ a_i \end{array}$$

- La comparación para S es análoga a la de Γ.
- Asumimos que la información en H_{in} sólo puede ser accedida desde variables en Γ_{in} o S_{in}. Podemos tener entradas que se parecen a:

Γ_{in}	H_{in}	S_{in} l	H_{out}	S_{out}
...			...	
$x \mapsto (\{\alpha\}, C, \mathsf{U})$	$\{\alpha\}.C.y \mapsto (\{\beta\}, D, \mathsf{U})$ $\{\beta\}.D.f \mapsto (\{\gamma\}, \ldots)$ $\{\gamma\}.B.g \mapsto \ldots$	...	...	
...			...	

Una vez que tuvo éxito la comparación de Γ y S, aplicamos las sustituciones obtenidas a H_{in} y utilizamos la siguiente función para la comparación de heaps:

$$\begin{array}{l} \text{match} <>_t <>_s = \ id \\ \text{match} < H >_t < H >_s \\ \quad = \ \text{let } (x, < H' >_t) = \text{noVar}(< H >_t) \\ \qquad \text{in} \\ \qquad \text{match'} < x \cdot H' >_t < H >_s \end{array}$$

donde *match'* se define como:

$$\begin{array}{l} \text{match'} < (\{a_1, \ldots, a_n\}.C.f \mapsto (\{\phi_1, \ldots, \phi_m\}, D, l)) \cdot H >_t \\ \qquad\qquad < H[\{a_1, \ldots, a_n\}.C.f \mapsto (\{b_1, \ldots, b_m\}, D, l))] >_s \\ \quad = \\ \quad \text{let } Subst = \{\rho \mid \rho(\{\phi_1, \ldots, \phi_m\}) = \{b_1, \ldots, b_m\}\} \\ \quad \text{in} \\ \qquad \text{if } Subst \text{ is empty} \\ \qquad \text{then } fail \\ \qquad \text{else} \\ \qquad\quad \bigcup_{\rho \in Subst} \rho \circ (\text{match} < \rho H >_t < H >_s) \\ \\ \text{match'} \_ \_ \ = \ fail \end{array}$$

La función *noVar* retorna un elemento de H_{in} que no tiene variables de referencia en la parte izquierda y los elementos que quedan en H_{in}. Esta función siempre encuentra un elemento sin variables de referencia por la asunción de que H_{in} es sólo accesible desde Γ_{in} y S_{in}.

Si coinciden las entradas, estaremos interesados en una sustitución ρ. Para obtener la modificación de S y H por el método en cuestión, usamos S_{out} y H_{out} donde las variables de referencia que aparecen en ρ son reemplazadas, y las restantes variables de referencia se reemplazan con un nombre de referencia nuevo.

Capítulo 3

Prototipo

En este capítulo se presenta el prototipo que implementa el análisis descripto en el capítulo 2. En la primera sección se describen detalles relacionados con la implementación del prototipo y en la segunda sección se mencionan aplicaciones que fueron probadas con el mismo.

3.1. Implementación

Esta sección comienza explicando el lenguaje elegido para la implementación del prototipo. La segunda sección describe la arquitectura del prototipo. La tercera sección explica una sesión de análisis típica de una aplicación web. La última sección menciona algunas limitaciones de la implementación actual del análisis.

3.1.1. Lenguaje de implementación

Para implementar el análisis descripto en el capítulo anterior se desarrolló un prototipo en el lenguaje de programación Java. La elección de este lenguaje se debió principalmente a las facilidades que provee Eclipse para hacer análisis estático de código Java.

Eclipse es un framework de código abierto. En su forma por defecto es un ambiente de desarrollo para Java. Los usuarios pueden extender sus capacidades instalando plug-ins escritos para el framework Eclipse y pueden además escribir y contribuir sus propios módulos plug-in.

El framework Eclipse provee funciones para obtener y manipular árboles de sintaxis a partir del código Java. Esto nos permitió librarnos de desarrollar un parser del lenguaje objeto y concentrarnos en recorrer la estructura de árbol dada por Eclipse para realizar el análisis de seguridad.

La elección de Java como lenguaje de implementación también nos permitió desarrollar el prototipo como un plug-in de Eclipse para que en el mismo ambiente en que se desarrolla una aplicación se pueda analizar la seguridad de la misma. De este modo un programador no experto en auditoría de código tiene la posibilidad de hacer un análisis de seguridad de su programa.

3.1.2. Arquitectura del prototipo

En la figura 3.1 se muestra un diagrama que contiene las partes principales del prototipo implementado y sus relaciones.

Vistas tiene la funcionalidad relacionada con las vistas del plug-in de Eclipse. Las vistas que presenta el prototipo para que el usuario pueda interactuar con el sistema de análisis son las siguientes:

- *Vista*: Se utiliza para asignar valores de seguridad a las variables y agregar información a las tablas de métodos. Presenta en un navegador los diferentes proyectos que están disponibles para ser analizados.

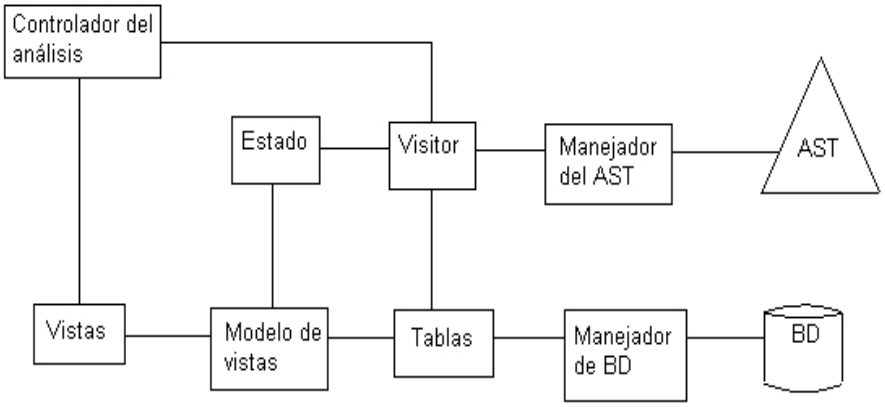

Figura 3.1: Arquitectura del prototipo

- *Métodos source*[1]: Muestra los métodos *source*. Estos métodos devuelven como resultado un valor ⊤ (no confiable). Los métodos usados para recibir entradas de usuarios de la aplicación se agregan en esta vista.
- *Métodos sink*: Muestra los métodos sink. Un método *sink* es aquel que no puede tener como parámetro un valor ⊤, por ejemplo, uno que se usa para hacer una consulta a una base de datos.
- *Debug*: Se usa para asignar los parámetros iniciales y ejecutar el análisis. Permite además seguir el análisis paso a paso, pudiendo inspeccionar y modificar el valor de seguridad de las variables durante el análisis.
- *Log*: Indica los eventos que surgen durante el análisis. Al seleccionar un evento que indica una vulnerabilidad se muestra la parte del código donde ésta se produce.

La información que muestran las vistas se encuentra en el *Modelo de vistas*.

Las vistas se comunican con el *Controlador del análisis* para iniciar el análisis de seguridad de la aplicación.

El código que implementa los esquemas de análisis vistos en el capítulo anterior se encuentra en el *Visitor*[2]. El Visitor se encarga de recorrer el *AST*[3] aplicando los esquemas de análisis.

[1]Source y sink son términos en inglés utilizados en la literatura.
[2]Nombre en inglés de un patrón de diseño usado para recorrer árboles.
[3]Por Abstract Syntax Tree que es el término en inglés para árbol de sintáxis.

En *Estado* se encuentran las implementaciones de las distintas estructuras que forman el estado en la ejecución simbólica, como ser los ambientes de variables y la heap simbólica.

Tablas es la parte del sistema que implementa las tablas de seguridad de métodos. Los valores seteados para las tablas antes del análisis o los calculados durante el análisis pueden guardarse en la *BD* (Base de Datos) para sesiones de análisis posteriores.

3.1.3. Descripción de una sesión típica

En esta sección se muestra una sesión de análisis típica usando PersonalBlog como la aplicación analizada.

El primer paso es cargar el proyecto `personalblog` en Eclipse y abrir las vistas del prototipo.

Luego, se indican los métodos source y sink en las vistas Métodos source y Métodos sink. Para el análisis de esta aplicación es relevante indicar como source el método `getParameter` y como sink el método `find`. El método `getParameter` lee los parámetros ingresados por un usuario de la aplicación. El método `find` permite realizar una consulta a la base de datos de la aplicación.

En la vista Debug se indica por qué método comienza el análisis de seguridad. Para PersonalBlog indicamos `executeSub` de la clase `ReadAction`.

Luego, en la vista Debug damos comienzo al análisis. Tenemos la opción de seguir el análisis paso a paso viendo cómo se modifica el estado simbólico de ejecución, o bien dejar que se analice automáticamente y ver los resultados en la vista Log.

Después de analizar PersonalBlog podemos ver que se detectaron 6 vulnerabilidades. Al seleccionar una vulnerabilidad se puede ver la parte del código donde se produce la misma (Figura 3.2).

En este caso, se invoca al método sink `find` con un argumento no confiable.

3.1.4. Limitaciones

El prototipo que se implementó para el análisis propuesto en este trabajo de grado tiene algunas limitaciones que se indican a continuación.

La funcionalidad de las tablas de seguridad respecto a la comparación de entradas no está implementada por el momento. En esta versión del prototipo cada vez que hay una llamada a método se analiza el cuerpo del método, aun cuando se podría usar el resultado del análisis de una llamada anterior al mismo método con una entrada similar. Esto además impide el análisis de métodos recursivos.

En esta versión del prototipo tampoco se implementaron los esquemas de análisis referidos a las excepciones. Algunas aplicaciones web por medio

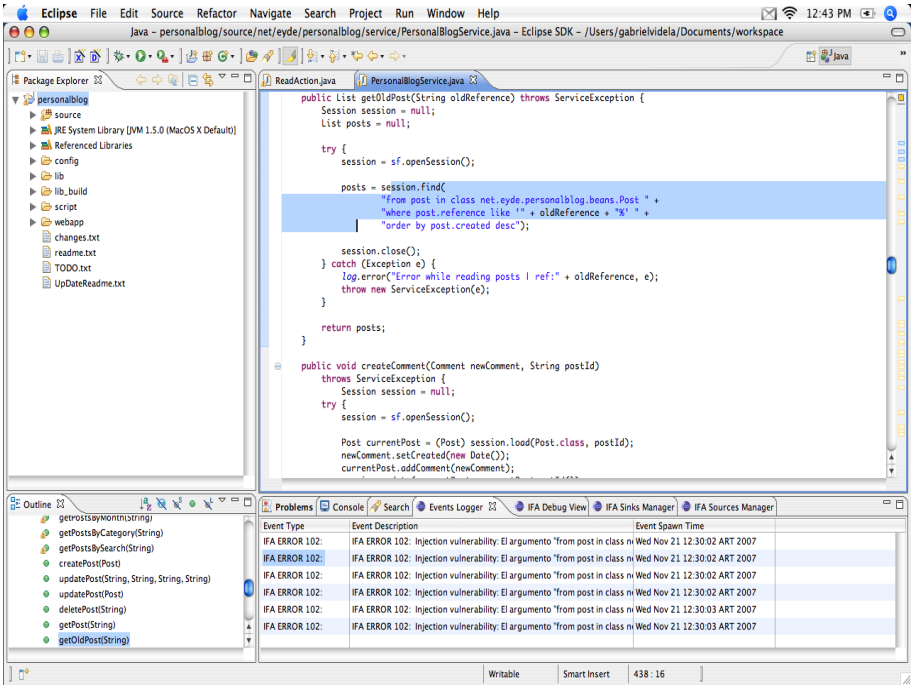

Figura 3.2: Vulnerabilidad en PersonalBlog

de excepciones pueden exponer información confidencial.

Una característica deseable sería poder indicar, al igual que se indican las funciones que son source o sink, las funciones que cambian su argumento a un valor de seguridad confiable. Esto permite, por ejemplo, no marcar como vulnerabilidad cuando un dato ingresado por un usuario es revisado por una función de seguridad de la aplicación y luego enviado a una consulta a una base de datos. Este efecto, conocido como *declasificación*, se puede lograr en la versión actual del prototipo haciendo un análisis paso a paso y modificando el estado actual de las variables.

3.2. Aplicaciones probadas

En esta sección se mencionan aplicaciones que fueron analizadas para probar que el análisis detecta vulnerabilidades.

3.2.1. PersonalBlog

PersonalBlog es una aplicación web pensada para hacer blogs personales. Está implementada en Java y es de código abierto. En esta aplicación el prototipo detectó 6 vulnerabilidades. A continuación se describe una de ellas mostrando porciones de código.

En esta aplicación tenemos parte del código en la que se leen datos del usuario con el método `getParameter` (Figura 3.3).

```
// Get request parameters
String reqDate = cleanNull(request.getParameter("caldate"));
String reqCategory = cleanNull(request.getParameter("cat"));
String reqMonth = cleanNull(request.getParameter("month"));
String reqPost = cleanNull(request.getParameter("post"));
String reqOldPost = cleanNull(request.getParameter("date"));
```

Figura 3.3: Lectura de datos del usuario

En las próximas instrucciones se invoca al método `getOldPost` con la variable `reqOldPost` que no es confiable porque proviene de un dato del usuario de la aplicación (Figura 3.4).

```
// Set Request Parameters
// Depending on the parameters, call the appropriate method
try {
    if (reqPost.length() > 1) {
        request.setAttribute("post", pblog.getPost(reqPost));
        request.setAttribute("posts", pblog.getPosts());
        forward = "readpost";
    } else if (reqOldPost.length() > 1) {
        request.setAttribute("posts", pblog.getOldPost(reqOldPost));
        forward = "readposts";
```

Figura 3.4: Pasaje de parámetro no confiable

Luego, se realiza una consulta a una base de datos con el parámetro formal `oldReference` que se corresponde con el parámetro real no confiable `reqOldPost` (Figura 3.5). En este punto un atacante podría ingresar un string que produzca una consulta no esperada por el sistema.

3.2.2. WebGoat

WebGoat es una aplicación web Java pensada para que sea insegura que es mantenida por OWASP [4] y diseñada para enseñar lecciones de seguridad de aplicaciones web. En cada lección, los usuarios tienen que demostrar su entendimiento de un tema de seguridad aprovechándose de una vulnerabilidad real en la aplicación WebGoat. Por ejemplo, en una lección el usuario tiene que usar inyección de SQL para robar números falsos de tarjetas de

[4]OWASP es una comunidad abierta que se centra en mejorar la seguridad de aplicaciones.

```
public List getOldPost(String oldReference) throws ServiceException {
    Session session = null;
    List posts = null;

    try {
        session = sf.openSession();

        posts = session.find(
                "from post in class net.eyde.personalblog.beans.Post " +
                "where post.reference like '" + oldReference + "%' " +
                "order by post.created desc");
```

Figura 3.5: Consulta a base de datos con parámetro no confiable

créditos. La aplicación es un ambiente de enseñanza realístico que provee a los usuarios de ayudas y código para enseñar la lección. A continuación mostramos cómo nuestro análisis detecta la vulnerabilidad en la lección sobre inyección SQL.

En la clase `SqlStringInjection` tenemos el método `injectableQuery` donde se detectó una vulnerabilidad al hacer una consulta a la base de datos con un parámetro no confiable (Figura 3.6).

```
protected Element injectableQuery(WebSession s)
{
ElementContainer ec = new ElementContainer();

try
{
    if (connection == null)
    {
    connection = DatabaseUtilities.makeConnection(s);
    }

    ec.addElement(makeAccountLine(s));

    String query = "SELECT * FROM user_data WHERE last_name = '"
        + accountName + "'";
    ec.addElement(new PRE(query));

    try
    {
    Statement statement = connection.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
    ResultSet results = statement.executeQuery(query);
```

Figura 3.6: Consulta a base de datos con parámetro no confiable

La lectura de los datos del usuario comienza en el llamado del método `makeAccountLine` (Figura 3.7).

Luego se llama al método `getRawParameter` (Figura 3.8) de la clase

```
protected Element makeAccountLine(WebSession s)
{
ElementContainer ec = new ElementContainer();
ec.addElement(new P().addElement("Enter your last name: "));

accountName = s.getParser().getRawParameter(ACCT_NAME, "Your Name");
Input input = new Input(Input.TEXT, ACCT_NAME, accountName.toString());
ec.addElement(input);

Element b = ECSFactory.makeButton("Go!");
ec.addElement(b);

return ec;

}
```

Figura 3.7: Lectura de parámetro no confiable (Parte 1)

`ParameterParser`.

```
public String getRawParameter(String name, String def)
{
try
{
    return getRawParameter(name);
}
catch (Exception e)
{
    return def;
}
}
```

Figura 3.8: Lectura de parámetro no confiable (Parte 2)

Y por último se llama al otro método `getRawParameter` que lee el parámetro no confiable por medio del método source `getParameterValues` (Figura 3.9).

3.2.3. Cofax

Cofax es una aplicación web Java de código abierto para el manejo de contenido (CMS). Fue diseñada para simplificar la presentación de periódicos en la web y para hacer publicaciones en tiempo real. Nuestro prototipo detectó 12 vulnerabilidades en esta aplicación de las cuales una de ellas se comenta a continuación.

En el método `navigate` de la clase `CofaxToolsNavigation` tenemos una parte del código en que se leen datos del usuario (Figura 3.10).

```
public String getRawParameter(String name)
    throws ParameterNotFoundException
{
String[] values = request.getParameterValues(name);

if (values == null)
{
    throw new ParameterNotFoundException(name + " not found");
}
else if (values[0].length() == 0)
{
    throw new ParameterNotFoundException(name + " was empty");
}

return (values[0]);
}
```

Figura 3.9: Lectura de parámetro no confiable (Parte 3)

Luego en el mismo método se llama al método `updatePublication` de la clase `CofaxToolsLifeCycle` con el valor leído (Figura 3.11).

Este valor está luego en el parámetro `pubName` del método `updatePublication` con el que se construye un string de SQL que es enviado al método `getPackageData` de la clase `MySqlDataStore` en el parámetro formal `tagData` (Figura 3.12). Por último se realiza una consulta a la base de datos con el valor no confiable (Figura 3.13).

```
HashMap htTime = CofaxToolsUtil.getDateInfo();
int year = Integer.parseInt((String) htTime.get("year"));
int month = Integer.parseInt((String) htTime.get("month"));
int day = Integer.parseInt((String) htTime.get("day"));
int hour = Integer.parseInt((String) htTime.get("hour"));
int minute = Integer.parseInt((String) htTime.get("minute"));
int second = Integer.parseInt((String) htTime.get("second"));
String amPm = (String) htTime.get("amPm");
CofaxToolsUser user = (CofaxToolsUser) session.getAttribute("user");
if (mode.equals("login_change_publication")) {
    if (user.userPubDescPubIDHash == null) {
        System.err.println("user.userPubDescPubIDHash NULL");
    }
    String selectPublication =
            CofaxToolsUtil.createSelect("PUBLICATION", user.userPubDescPubIDHash, (String) user.userInfoHash.get("HOMEPUB"));
    page.putGlossaryValue("system:message", CofaxToolsUtil.getI18NMessage(req.getLocale(),"choosepublication"));
    page.putGlossaryValue("system:highLightTab", "admin");
    page.putGlossaryValue("system:selectPublication", selectPublication);
    includeResource(page, "/toolstemplates/selectPublication.jsp",
            req, res, session);
    return ("");
} else if ((mode.equals("admin")) ||
```

Figura 3.10: Lectura de datos

```
} else if ((mode.equals("admin")) ||
        (mode.equals("admin_clear_cache")) ||
        (mode.equals("admin_reset_servlet")) ||
        (mode.equals("admin_update_lifecycles"))) {
    if (mode.equals("admin_clear_cache")) {
        //CofaxToolsServlet.dataStore.clearPackageCache();
        CofaxToolsServlet.dataStore.clearCache(DataStore.PACKAGE_TAG_RESULTS);
        CofaxToolsServlet.dataStore.clearCache(DataStore.CLOB_COLUMNS);
        CofaxToolsUtil.log("CofaxToolsNavigation navigate: Clearing database cache hash from admin_clear_cache.");
        page.putGlossaryValue("system:message", CofaxToolsUtil.getI18NMessage(req.getLocale(),"cacheCleared"));
    } else if (mode.equals("admin_reset_servlet")) {
        boolean reset = CofaxToolsUtil.resetServlet(db);
        page.putGlossaryValue("system:message", CofaxToolsUtil.getI18NMessage(req.getLocale(),"servletReset"));
    } else if (mode.equals("admin_update_lifecycles")) {
        CofaxToolsLifeCycle lf = new CofaxToolsLifeCycle();
        String rt = lf.updatePublication(db, user.workingPubName);
        int x1 = lf.count_articles_disabled;
        int x2 = lf.count_articles_enabled;
```

Figura 3.11: Pasaje de parámetro no confiable (Parte 1)

```
public String updatePublication(DataStore db, String pubName) {
    try {
        StringBuffer v_articles = new StringBuffer();
        v_articles.append("select A.itemID, A.pubStart, A.pubEnd ")
                  .append("from tblArticles AS A ")
                  .append("where A.pubname= '" + pubName + "' ")
                  .append("and   A.lifeCycle = 1");

        count_articles_disabled = 0;
        count_articles_enabled = 0;

        ArrayList articles = new ArrayList();
        HashMap ht = new HashMap();
        articles = (ArrayList)db.getPackageData(ht, "", v_articles.toString(), false);
```

Figura 3.12: Pasaje de parámetro no confiable (Parte 2)

```
    } else {
        // initialise le statement
        // puis execution
        sql = connection.createStatement();
        result = sql.executeQuery(tagData);
    }
```

Figura 3.13: Consulta a la base de datos

Capítulo 4

Conclusión

Este capítulo concluye el presente informe de trabajo de grado mencionando los resultados obtenidos en el mismo, trabajos futuros y trabajos relacionados.

4.1. Resultados obtenidos

En el informe final del trabajo de grado se comenzó realizando una introducción a la seguridad de aplicaciones web y a la técnica de análisis de programas basada en flujo de información. Luego, se describió en detalle el análisis de seguridad de aplicaciones web propuesto. Por último, se presentó un prototipo que implementa el mencionado análisis y con el cual se pudo probar que se detectaban vulnerabilidades en aplicaciones web.

El resultado de este trabajo es la aplicación de una técnica de análisis estático de flujo de información a Java con la finalidad de detectar vulnerabilidades en aplicaciones web. Se buscó un enfoque práctico para analizar código legacy Java sin la necesidad de anotaciones por parte de los usuarios. Se obtuvo además un prototipo que implementa el análisis. La interacción con el grupo encargado de la implementación del prototipo permitió mejorar el análisis y eliminar posibles errores.

Se comenzó el trabajo de grado haciendo un relevamiento de la literatura y aplicaciones existentes relacionadas con la detección de vulnerabilidades en aplicaciones web. Se analizaron ejemplos motivadores de vulnerabilidades que se deseaban detectar. Luego se determinó un subconjunto de interés del lenguaje Java. Para estas construcciones se diseñaron esquemas de análisis basados en la técnica de flujo de información. El análisis obtenido fue probado contra ejemplos para comprobar que se detectan vulnerabilidades.

Algunas limitaciones del análisis están relacionadas con el tratamiento de concurrencia y reflection. Estos temas decidimos no abordarlos por ser lo suficientemente complejos como para requerir un estudio por separado. El hecho de que hilos de ejecución secuenciales compartan memoria hace que entren en juego nuevos canales de información. Por ejemplo, consideremos los siguientes comandos de hilos de ejecución donde la variable l es pública y la variable h es privada y *secreto* representa un valor confidencial:

$$c_1 : h = 0; l = h \qquad c_2 : h = secreto$$

El hilo c_1 es seguro porque el valor final de l es siempre 0. El hilo c_2 es seguro porque ambos h y *secreto* están en el mismo nivel de seguridad. Sin embargo, la composición paralela $c_1||c_2$ de los dos hilos de ejecución no es necesariamente segura. El scheduler puede indicar que se ejecute c_2 después de la asignación $h = 0$ y antes de que se ejecute $l = h$ en c_1. Como resultado, *secreto* se copia en l.

Otra limitación del análisis se encuentra en el tratamiento de aliasing. Si consideramos el siguiente condicional donde la variable y tiene la etiqueta

de seguridad $(\{a\}, C, \mathsf{U})$ y la variable z tiene la etiqueta $(\{b\}, C, \mathsf{U})$:

```
if (e)
  x = y;
else
  x = z;
```

El valor con el que queda la variable x luego de ejecutar el condicional es $(\{a, b\}, C, \mathsf{U})$. De este modo, el valor de seguridad de un campo del objeto x se obtiene del supremo entre el valor del campo en la posición a de la heap y el valor del campo en la posición b de la heap. Esta unión de referencias simbólicas que se realiza en cada condicional influye en la escalabilidad del análisis.

El presente trabajo se realizó en el marco del proyecto *Plataforma híbrida de seguridad para protección de aplicaciones web* entre Core SA y LIFIA. El equipo de Core aportó su conocimiento y experiencia respecto a seguridad de aplicaciones web. Los que formamos el equipo del laboratorio de LIFIA estuvimos a cargo del diseño e implementación del análisis. Este trabajo permitió mostrar que se pueden obtener resultados interesantes a partir de la interacción entre la industria y la academia.

4.2. Trabajos futuros

Como trabajo futuro se pueden abordar los temas concurrencia [9, 10, 11] y reflection [12], lo cual haría más completo al análisis de seguridad presentado. Otro tema interesante para considerar es la ampliación del reticulado de valores de seguridad [19] para que el análisis sea más preciso respecto a las vulnerabilidades detectadas.

En cuanto a la implementación, la detección de vulnerabilidades relacionadas con excepciones que fue considerada en el análisis no se implementó aún en el prototipo. Sería deseable además que un usuario pueda indicar cuáles funciones cambian su argumento a un nivel de seguridad confiable. Este efecto, conocido como declasificación, se puede lograr en la versión actual del prototipo haciendo un análisis paso a paso y modificando el estado actual de las variables. La funcionalidad de las tablas de seguridad respecto a la comparación de entradas no fue implementada, lo cual impide el reuso de análisis previos sobre métodos.

4.3. Trabajos relacionados

En esta sección se mencionan trabajos de investigación relacionados con la detección de vulnerabilidades en aplicaciones web.

Myers [6] diseñó un análisis basado en tipos y en flujo de información para el lenguaje Jif que extiende parte de Java con información de seguridad.

El enfoque que tiene su trabajo es en un modelo de programación práctico y realístico. Si bien tiene inferencia para parte de los tipos de seguridad, requiere anotaciones por parte de los usuarios. Su análisis provee formas para desclasificar los valores de seguridad de variables. Algunas limitaciones de su análisis es que no abarca todo Java y no es aplicable a programas Java sin ninguna anotación. La salida del compilador para Jif es un programa Java.

Banerjee y Naumann [7] hicieron un análisis estático de seguridad basado en flujo de información para un lenguaje reducido similar a Java. Al igual que el análisis de Myers, requiere anotaciones para los tipos de seguridad. El objetivo de su trabajo es manejar la alocación dinámica de memoria y construcciones de lenguaje orientadas a objetos. A pesar de que consideran un fragmento de Java más pequeño que el de Myers, justifican sus reglas de análisis con un resultado de no interferencia. Junto con Sun [13] especificaron un algoritmo de inferencia de tipos de seguridad para el lenguaje en cuestión.

Bonelli et al. [14, 15] presentan un lenguaje assembly tipado con una pila de ejecución polimórfica. Definen un sistema de tipos para garantizar un flujo de información seguro. Además acompañan la presentación con un resultado de no-interferencia. Bonelli y Bavera [16] presentan un análisis estático basado en tipos de flujo de información para un lenguaje similar al bytecode de Java. Estos análisis requieren anotaciones de tipos por parte de los usuarios además de no ser lenguajes utilizados en la industria.

Livshits y Lam [17] desarrollaron un análisis estático para detectar vulnerabilidades en aplicaciones web escritas en Java. Los usuarios proveen especificaciones de vulnerabilidades en el lenguaje PQL [18] que son traducidas a analizadores estáticos. Una ventaja de su análisis es que se hace a nivel de bytecode, lo cual es útil para bibliotecas cuyo código fuente no está disponible. Una desventaja del análisis es que no considera flujos de información implícitos. Para este análisis desarrollaron la herramienta LAPSE que es un plugin de Eclipse.

Huang et al. [19] presentan un análisis de seguridad para aplicaciones web que mezcla técnicas estáticas y dinámicas. El análisis se basa en flujo de información y, a diferencia del análisis de Myers que es más estricto, permite que las variables tengan distintos valores de seguridad en distintos puntos del programa. Durante el análisis, en las secciones de código vulnerables se introducen rutinas para proteger la aplicación web en tiempo de ejecución. Esto último se hace de forma automática sin requerir la intervención del usuario. Para probar su algoritmo crearon la herramienta WebSSARI que analiza aplicaciones escritas en el lenguaje PHP.

Xie y Aiken [20] diseñaron un algoritmo de análisis estático para detectar vulnerabilidades en PHP. Su análisis se basa en una arquitectura que les permite considerar características dinámicas únicas de lenguajes de scripting, como ser la inclusión de código y el tipado dinámico. Estas últimas características no son consideradas en WebSSARI. El análisis tiene la ventaja de

requerir poca intervención por parte del usuario.

Jovanovic et al. [21] diseñaron un análisis estático para detectar vulnerabilidades en aplicaciones web escritas en PHP. A diferencia del análisis de Huang et al., consideran inclusiones de archivo por medio de un algoritmo de dos fases. Una ventaja respecto del análisis de Xie y Aiken es que consideran llamadas a funciones recursivas. El análisis de arreglos y características orientadas a objetos de PHP es limitado. Para probar el análisis desarrollaron la herramienta Pixy.

Futoransky y Waissbein [22] presentan un análisis dinámico para la protección de aplicaciones web. El mismo consiste en la modificación del ambiente de ejecución para que los valores del programa estén acompañados de una etiqueta de seguridad. La etiqueta de seguridad permite establecer diferentes políticas de seguridad. Hicieron una implementación del análisis para proteger aplicaciones PHP.

Buehrer et al. [23] diseñaron un análisis de seguridad en tiempo de ejecución para evitar la manipulación de una sentencia SQL por parte de un atacante. La técnica consiste en comparar el árbol sintáctico de una sentencia SQL antes de que se suministre una entrada de usuario con el árbol sintáctico que resulta luego de insertar la entrada. Para un análisis empírico, implementaron su solución para programas escritos en Java.

Halfond y Orso [24] desarrollaron un análisis en parte estático y en parte dinámico para proteger aplicaciones web de ataques por medio de la modificación de sentencias SQL. Durante el análisis estático por cada sentencia SQL se crea un modelo conservativo usando un autómata de las sentencias previstas en el programa. Luego, en ejecución, se comparan las sentencias SQL que se crean con las que son aceptadas por el modelo obtenido anteriormente.

Apéndice A

Listado de esquemas de análisis

En este apéndice se presenta el listado completo de los esquemas de análisis diseñados para expresiones y comandos de Java.

A.0.1. Esquemas de análisis para expresiones

Recordemos que el juicio de análisis para expresiones toma la siguiente forma:

$$\boxed{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}$$

El siguiente juicio de análisis auxiliar se usa para secuencias de expresiones.

$$\boxed{E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_2; (\sigma_1, \ldots, \sigma_n)}$$

Y se define por los siguientes dos esquemas. El primero pertenece a una secuencia no vacía de expresiones mientras que el segundo se aplica a secuencias vacías.

$$\frac{\begin{array}{rcl} E; \mathcal{M}_1; l \vdash e_1 & \Rightarrow_e & \mathcal{M}_2; \sigma_1 \\ E; \mathcal{M}_2; l \vdash (e_2, \ldots, e_n) & \Rightarrow_{\overline{e}} & \mathcal{M}_3; (\sigma_2, \ldots, \sigma_n) \end{array}}{E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_3; (\sigma_1, \sigma_2, \ldots, \sigma_n)}$$

$$\frac{}{E; \mathcal{M}; l \vdash () \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}; ()}$$

- **Asignación.** Hay dos esquemas para una asignación $x = e$ dependiendo si x se refiere a una variable local o estática. Estos esquemas simplemente actualizan el ambiente correspondiente con el valor de seguridad que resulta del análisis de e. Quizás vale aclarar que este valor tiene al menos el nivel l del contador de programa.

(E-Asignacion)

$$\frac{\begin{array}{c}x \text{ es una variable local o parámetro} \\ E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma\end{array}}{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \Gamma_2[x := \sigma]; H_2; S_2; \sigma}$$

(E-Asignacion-Estatica)

$$\frac{\begin{array}{c}x \text{ es estática} \\ E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma\end{array}}{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \Gamma_2; H_2; S_2[x := \sigma]; \sigma}$$

Las variables locales a un método sólo existen durante la ejecución del mismo, por esto, no se consideran en las tablas de seguridad. El caso de las variables estáticas sí se considera porque existen independientemente del método.

Para la asignación de una variable estática x se agrega la variable en la estructura S_{out} para indicar su modificación. Si es la primera vez que se lee esta variable ($x \notin S_{in}$) se guarda su valor en S_{in}.

$$\frac{\begin{array}{c}x \text{ es estática} \\ \mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2\end{array}}{\mathcal{T}_1 \vdash x = e \quad \Rightarrow_e \quad \mathcal{T}_2 \;\; \begin{array}{l}[S_{out} \cup = \; \{x\}] \\ [S_{in}(x) = S(x)] \quad (x \notin S_{in})\end{array}}$$

- **Asignación de campo.** A diferencia de la asignación de una variable local o estática, la asignación de un campo $e_1.f = e_2$ implica actualizar la heap simbólica dado que los objetos residen en la heap.

  Primero, se analiza la expresión e_1 que representa un objeto con campo f, obteniendo la referencia A para actualizar la heap. Luego, el análisis de e_2 nos da el nuevo valor de seguridad para el campo f.

  En la cadena formada por la clase de e_1 y sus superclases podría haber varias definiciones de f. Para saber cuál f se actualiza, Java realiza una búsqueda estática, es decir, a partir de la clase del tipo asignado a e_1 en tiempo de compilación. Para obtener el tipo que se le asigna a e_1 en tiempo de compilación usamos la función Tipo. La búsqueda de la clase que define el campo la realizamos con la función Buscar.

  La actualización $H[a.D.f := \sigma]$ para cada referencia $a \in A$ la escribimos con la notación más corta $H[A.D.f := \sigma]$.

  (E-AsignacionCampo)

$$\frac{\begin{array}{c}E; \mathcal{M}_1; l_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ E; \mathcal{M}_2; l_1 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{M}_3; \sigma \\ \textsc{Buscar}(\textsc{Tipo}(e_1), f) = D\end{array}}{E; \mathcal{M}_1; l_1 \vdash e_1.f = e_2 \quad \Rightarrow_e \quad \Gamma_3; H_3[A.D.f := \sigma]; S_3; \sigma}$$

Cuando se modifica un campo, el cambio se ve fuera del método sólo si la referencia al objeto que tiene el campo sigue existiendo fuera del método ($A \in \Gamma_{in} \cup H_{in} \cup S_{in}$). En este caso, se guarda en H_{out} la posición $A.D.f$ que se actualizó. Además, si es la primera vez que se analiza este campo en el método ($A.D.f \notin H_{in}$) se guarda su valor de seguridad en H_{in}.

El siguiente es un ejemplo en el que el campo f no es visible fuera del método:

```
void m() {
  C x = new C();
  x.f = 0;
}
```

$$\frac{\begin{array}{c}\mathcal{T}_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{T}_2; (A, C, l_2) \\ \mathcal{T}_2 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{T}_3 \\ \textsc{Buscar}(\textsc{Tipo}(e_1), f) = D\end{array}}{\begin{array}{ll}\mathcal{T}_1 \vdash e_1.f = e_2 \quad \Rightarrow_e \quad \mathcal{T}_3 & [H_{out} \cup = \{A.D.f\}] \\ & (A \in \Gamma_{in} \cup H_{in} \cup S_{in}) \\ & \\ & [H_{in}(A.D.f) = H(A.D.f)] \\ & (A \in \Gamma_{in} \cup H_{in} \cup S_{in} \\ & \wedge \\ & A.D.f \notin H_{in})\end{array}}$$

- **Asignación de campo de la superclase.** Cuando un campo f tiene como prefijo la palabra `super`, se refiere a un campo del objeto *this*. Para obtener la referencia A del objeto *this* se busca en el ambiente Γ. El campo f a ser modificado se busca estáticamente a partir de la superclase de la clase host E.

  (E-AsignacionCampo-Super)

$$\frac{\begin{array}{c}E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma \\ \Gamma(this) = (A, C, l_2) \\ \textsc{Buscar}(\textsc{Super}(E), f) = D\end{array}}{E; \mathcal{M}_1; l_1 \vdash \texttt{super}.f = e \quad \Rightarrow_e \quad \Gamma_2; H_2[A.D.f := \sigma]; S_2; \sigma}$$

  La modificación de la tabla es similar a la modificación para la asignación de campo. Además, considera el hecho de que se lee *this*, si es la primera vez que se lee ($this \notin \Gamma_{in}$) se agrega a Γ_{in}.

$$\frac{\begin{array}{c}\mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2 \\ \Gamma(this) = (A, C, l_2) \\ \textsc{Buscar}(\textsc{Super}(E), f) = D\end{array}}{\begin{array}{l}\mathcal{T}_1 \vdash \texttt{super}.f = e \quad \Rightarrow_e \quad \mathcal{T}_2 \ [H_{out} \cup = \ \{A.D.f\}] \\ \qquad\qquad\qquad\qquad\qquad\qquad [H_{in}(A.D.f) = H(A.D.f)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (A.D.f \notin H_{in}) \\ \qquad\qquad\qquad\qquad\qquad\qquad [\Gamma_{in}(this) = (A, C, l_2)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad (this \notin \Gamma_{in})\end{array}}$$

- **Asignación de arreglo.** La forma en que diseñamos los arreglos en el análisis fue como un objeto de la clase $t[\,]$, donde t es la clase o tipo primitivo de los elementos que contiene el arreglo, y un campo *cont* que guarda el valor de seguridad de sus elementos.

  Para la expresión $e_1[e_2] = e_3$ se analiza primero el indice e_2, luego la expresión e_1 que representa el arreglo para obtener su referencia, y por último, la expresión e_3 para obtener el nuevo valor de seguridad para los elementos del arreglo.

  (E-AsignacionArr)

$$\frac{\begin{array}{c}E; \mathcal{M}_1; l_1 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{M}_2; l_2 \\ E; \mathcal{M}_2; l_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_3; (A, t[\,], l_3) \\ E; \mathcal{M}_3; l_1 \vdash e_3 \quad \Rightarrow_e \quad \mathcal{M}_4; \sigma\end{array}}{E; \mathcal{M}_1; l_1 \vdash e_1[e_2] = e_3 \quad \Rightarrow_e \quad \Gamma_4; H_4[A.cont := \sigma]; S_4; \sigma}$$

- **Literal.** En el caso de un literal de un tipo primitivo, el estado de memoria permanece igual y el valor de seguridad de la expresión es el del contador de programa.

$$\frac{}{E; \mathcal{M}; l \vdash \underline{p} \quad \Rightarrow_e \quad \mathcal{M}; l}\ \text{E-Primitivo}$$

  Cuando tenemos un literal que representa un objeto como es el caso de un string, creamos un objeto nuevo de la clase correspondiente. El esquema para un string es el siguiente:

$$\frac{E; \mathcal{M}_1; l \vdash \texttt{new}\ String() \quad \Rightarrow_e \quad \mathcal{M}_2; (\{a\}, String, l)}{E; \mathcal{M}_1; l \vdash \underline{s} \quad \Rightarrow_e \quad \mathcal{M}_2; (\{a\}, String, l)}\ \text{E-String}$$

  Para el caso de un inicializador de arreglo usamos el siguiente esquema que crea un nuevo arreglo (ver el esquema de declaración de arreglos

para más detalle). $\underline{a}$ representa un inicializador de un arreglo de n dimensiones con elementos del tipo t.

(E-InicializadorArreglo)

$$\frac{a \; fresh}{E; \mathcal{M}_1; l \vdash \texttt{arr} < a, t[\,]^1 \ldots [\,]^{n-1} > \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash \underline{a} \quad \Rightarrow_e \quad \mathcal{M}_2; (\{a\}, t[\,]^1 \ldots [\,]^n, l)}$$

- **This.** El valor de seguridad para *this* es el almacenado en el ambiente de variables locales actualizado con el nivel del contador de programa.

$$\frac{}{E; \mathcal{M}; l \vdash \texttt{this} \quad \Rightarrow_e \quad \mathcal{M}; \Gamma(this) \sqcup l} \text{ E-This}$$

Si es la primera vez que se lee *this* ($this \notin \Gamma_{in}$) entonces se guarda su valor de seguridad en Γ_{in}.

$$\boxed{\frac{}{\mathcal{T} \vdash \texttt{this} \quad \Rightarrow_e \quad \mathcal{T}[\Gamma_{in}(this) = \Gamma(this)] \quad (this \notin \Gamma_{in})}}$$

- **Variable.** El valor de seguridad de una variable es el almacenado en el ambiente apropiado y actualizado para incluir el nivel del contador de programa. Hay dos esquemas dependiendo si la variable es estática o no.

Para la tabla de seguridad, se verifica si es la primera vez que se lee x, en cuyo caso se agrega a Γ_{in} o S_{in} según corresponda.

$$\frac{x \text{ es una variable local o parámetro}}{E; \mathcal{M}; l \vdash x \quad \Rightarrow_e \quad \mathcal{M}; \Gamma(x) \sqcup l} \text{ E-Variable}$$

$$\boxed{\frac{x \text{ es un parámetro}}{\mathcal{T} \vdash x \quad \Rightarrow_e \quad \mathcal{T}[\Gamma_{in}(x) = \Gamma(x)] \quad (x \notin \Gamma_{in})}}$$

$$\frac{x \text{ es estática}}{E; \mathcal{M}; l \vdash x \quad \Rightarrow_e \quad \mathcal{M}; S(x) \sqcup l} \text{ E-Variable-Estatica}$$

$$\boxed{\frac{x \text{ es estática}}{\mathcal{T} \vdash x \quad \Rightarrow_e \quad \mathcal{T}[S_{in}(x) = S(x)] \quad (x \notin S_{in})}}$$

- **Acceso a campo.** Hay dos esquemas dependiendo del receptor del acceso a campo, que puede ser *super* o una expresión e diferente de *super*.

  Si tenemos la expresión $e.f$, primero analizamos e para obtener la referencia A. Como mencionamos antes, la búsqueda del campo f se hace de forma estática. Luego, se devuelve el supremo de los valores de cada referencia ($\bigsqcup_{a \in A} H(a.D.f)$) que lo notamos de forma corta $H(A.D.f)$, actualizado con el nivel de seguridad de la referencia al objeto e.

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ \textsc{Buscar}(\textsc{Tipo}(e), f) = D \end{array}}{E; \mathcal{M}_1; l_1 \vdash e.f \quad \Rightarrow_e \quad \mathcal{M}_2; H_2(A.D.f) \sqcup l_2} \text{ E-}\textsc{Campo}$$

  Para la tabla de seguridad se verifica si es la primera vez que se lee el campo.

$$\frac{\begin{array}{c} \mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2; (A, C, l_2) \\ \textsc{Buscar}(\textsc{Tipo}(e), f) = D \end{array}}{\mathcal{T}_1 \vdash e.f \quad \Rightarrow_e \quad \mathcal{T}_2 \;\; \begin{array}{c} [H_{in}(A.D.f) = H(A.D.f)] \\ (A.D.f \notin H_{in}) \end{array}}$$

  Cuando se tiene el prefijo *super* se busca el campo a partir de *this* y se hace el supremo de las referencias ($H(A.D.f)$) junto con los niveles de seguridad de la referencia a *this* y del contador de programa.

  (E-Campo-Super)

$$\frac{\begin{array}{c} \Gamma(this) = (A, C, l_2) \\ \textsc{Buscar}(\textsc{Super}(E), f) = D \end{array}}{E; \mathcal{M}_1; l_1 \vdash \texttt{super}.f \quad \Rightarrow_e \quad \mathcal{M}; H(A.D.f) \sqcup l_1 \sqcup l_2}$$

  Para la tabla de seguridad se considera además si es la primera vez que se lee *this*.

$$\frac{\begin{array}{c} \Gamma(this) = (A, C, l_2) \\ \textsc{Buscar}(\textsc{Super}(E), f) = D \end{array}}{\mathcal{T} \vdash \texttt{super}.f \quad \Rightarrow_e \quad \mathcal{T} \;\; \begin{array}{c} [H_{in}(A.D.f) = H(A.D.f)] \\ (A.D.f \notin H_{in}) \\ \\ [\Gamma_{in}(this) = (A, C, l_2)] \\ (this \notin \Gamma_{in}) \end{array}}$$

- **Acceso a arreglo.** El valor de seguridad de un acceso a un arreglo está dado por el supremo entre los valores de seguridad del contenido del arreglo, la referencia al arreglo y el resultante de evaluar el índice.

  (E-Arreglo)

$$\frac{\begin{array}{cccc} E; \mathcal{M}_1; l_1 \vdash e_2 & \Rightarrow_e & \mathcal{M}_2; l_2 \\ E; \mathcal{M}_2; l_1 \vdash e_1 & \Rightarrow_e & \mathcal{M}_3; (A, t[\,], l_3) \end{array}}{E; \mathcal{M}_1; l_1 \vdash e_1[e_2] \quad \Rightarrow_e \quad \mathcal{M}_3; H_3(A.cont) \sqcup l_2 \sqcup l_3}$$

- **Invocación a método.** Existen tres esquemas para analizar un método dependiendo del receptor, que puede ser una expresión e que representa un objeto, *super* o una clase C. Cuando el receptor es una clase tenemos un método estático.

  Cuando tenemos una expresión e que representa un objeto al que se le envía un mensaje m, se analiza primero esta expresión para obtener su valor de seguridad que es el que corresponde al parámetro *this*. Luego, se obtienen los valores de seguridad de los parámetros reales. El método m se busca dinámicamente a partir de la clase del receptor subiendo por sus superclases. Esto es porque Java tiene binding dinámico para métodos. Observar que para el acceso a campos esto no se cumple. Finalmente, se obtiene a partir de los datos anteriores y la tabla de seguridad del método m el estado de memoria resultante en el caso de terminación normal $(H_4; S_4; \sigma)$ y los estado de memoria que resultan de una terminación excepcional $(H^{E_i}, S^{E_i}, \sigma^{E_i})$. Las E_i son las excepciones que declara el método m. El valor de seguridad de la invocación está dado por el valor de seguridad del valor de retorno (σ).

  Las invocaciones a métodos que no devuelven un valor no pueden ser expresiones. Para estos métodos el σ no está presente.

  (E-Invocacion)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, C, l_2) \\ E; \mathcal{M}_2; l_1 \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_3; (\sigma_1, \ldots, \sigma_n) \\ \textsc{Buscar}(C, m) = D \\ \left(\begin{array}{l} (H_4, S_4, \sigma), \\ \\ \{(H^{E_i}, S^{E_i}, \sigma^{E_i})\} \end{array} \right) = \mathcal{T} \begin{array}{l} (D.m[\textsc{Tipo}(e_1), \ldots, \textsc{Tipo}(e_n)]) \\ (\{\ x_i : \sigma_i\ \} \quad \cup \quad \{\ this : (A, C, l_2)\ \} \\ \quad ; H_3 \\ \quad ; S_3) \\ (l_1) \end{array} \end{array}}{E; \mathcal{M}_1; l_1 \vdash e.m(e_1, \ldots, e_n) \quad \Rightarrow_e \quad \Gamma_3; H_4; S_4; \sigma}$$

  La diferencia cuando la invocación es a partir de *super* es que la

búsqueda del método comienza a partir de la superclase del receptor que en este caso es *this*.

(E-Invocacion-Super)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_2; (\sigma_1, \ldots, \sigma_n) \\ \textsc{Buscar}(\textsc{Super}(E), m) = D \\ \left(\begin{array}{l} (H_3, S_3, \sigma), \\ \\ \{(H^{E_i}, S^{E_i}, \sigma^{E_i})\} \end{array} \right) = \mathcal{T} \begin{array}{l} (D.m[\textsc{Tipo}(e_1), \ldots, \textsc{Tipo}(e_n)]) \\ (\{\ x_i : \sigma_i\ \} \quad \cup \quad \{\ this :\ \Gamma_2(this)\ \} \\ \quad ; H_2 \\ \quad ; S_2) \\ (l) \end{array} \end{array}}{E; \mathcal{M}_1; l \vdash \texttt{super}.m(e_1, \ldots, e_n) \quad \Rightarrow_e \quad \Gamma_2; H_3; S_3; \sigma}$$

En el caso de métodos estáticos $C.m$, se busca a partir de la clase C.

(E-Invocacion-Estatica)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_2; (\sigma_1, \ldots, \sigma_n) \\ \textsc{Buscar}(C, m) = D \\ \left(\begin{array}{l} (H_3, S_3, \sigma), \\ \\ \{(H^{E_i}, S^{E_i}, \sigma^{E_i})\} \end{array} \right) = \mathcal{T} \begin{array}{l} (D.m[\textsc{Tipo}(e_1), \ldots, \textsc{Tipo}(e_n)]) \\ (\{\ x_i : \sigma_i\ \}) \\ \quad ; H_2 \\ \quad ; S_2) \\ (l) \end{array} \end{array}}{E; \mathcal{M}_1; l \vdash C.m(e_1, \ldots, e_n) \quad \Rightarrow_e \quad \Gamma_2; H_3; S_3; \sigma}$$

Cuando estamos analizando un método m_1 y encontramos una invocación a un método m_2, obtenemos la entrada que coincide en la tabla de m_2 para modificar el estado actual de la tabla de m_1. Por ejemplo, si tenemos el código:

```
int m1(A x) {
  ...
  this.m2(x);
  ...
}

void m2(A z) {
  ...
}
```

y la entrada de la tabla de m_2 que coincide con los elementos del análisis al momento de evaluar la invocación:

Γ_{in}	H_{in}	S_{in}	l	H_{out}	S_{out}
...				...	
$z \mapsto (\{\alpha\}, A, \mathsf{U})$		$w \mapsto \mathsf{U}$	U		$w \mapsto \mathsf{T}$
...				...	

haremos las siguientes modificaciones al estado actual de la tabla de m_1:

$$\mathcal{T}[\Gamma_{in}(x) = \Gamma(x)] \quad (x \notin \Gamma_{in})$$

$$\mathcal{T}[S_{out} \cup = \ \{w\}]$$

El primer cambio es para indicar que se lee el parámetro x y el segundo indica que la variable estática w fue modificada.

- **New.** Un `new` se analiza de forma similar a una invocación a método con la diferencia de que la variable *this* que recibe como parámetro es un objeto nuevo con los campos seteados con la función ET.

  (E-NEW)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l \vdash (e_1, \ldots, e_n) \quad \Rightarrow_{\overline{e}} \quad \mathcal{M}_2; (\sigma_1, \ldots, \sigma_n) \\ a \text{ fresh} \quad \textsc{Campos}(C) = \{C_i.f_j\} \\ \left(\begin{array}{l} (H_3, S_3, \sigma), \\ \\ \{(H^{E_i}, S^{E_i}, \sigma^{E_i})\} \end{array} \right) = \mathcal{T} \ \begin{array}{l} (C.C[\textsc{Tipo}(e_1), \ldots, \textsc{Tipo}(e_n)]) \\ (\{\ x_k : \sigma_k\ \} \quad \cup \quad \{this : (\{a\}, C, l)\} \\ \quad ; H_2[a := [\{C_i.f_j : \textsc{Et}(C_i.f_j)\}]]) \\ \quad ; S_2) \\ (l) \end{array} \end{array}}{E; \mathcal{M}_1; l \vdash \texttt{new}\ C(e_1, \ldots, e_n) \quad \Rightarrow_e \quad \Gamma_2; H_3; S_3; \sigma}$$

- **Declaración.** Una declaración de variable con inicialización se analiza de la misma manera que una asignación.

$$\frac{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash \texttt{decl}\ x = e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}\ \text{E-DECLARACION}$$

  Cuando una declaración no incluye la inicialización se analiza como si tuviera una inicialización con el valor dado por ET para esa variable. La función ET devuelve un valor de seguridad que setea el usuario para esa variable o, si el usuario no seteó este valor, un valor dado por PD. La función PD devuelve un nivel de seguridad establecido por el usuario antes de comenzar el análisis.

- **Declaración de arreglo.** Una declaración de una variable arreglo se analiza de la misma manera que una asignación.

(E-Decl-Arreglo-Ini)

$$\frac{E; \mathcal{M}_1; l \vdash x = e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash \texttt{decl}\ t[\,]^1 \ldots [\,]^n\ x = e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}$$

Para una declaración sin inicialización se usan los siguientes tres esquemas para crear la estructura del arreglo en la heap. Estos esquemas usan una estructura auxiliar `arr` para llevar la cuenta de cuántas dimensiones tiene el arreglo que se está creando.

Si x es estática, en el esquema E-Decl-Arreglo se actualiza el ambiente S_2 en vez de Γ_2.

(E-Decl-Arreglo)

$$\frac{\begin{array}{c} a\ fresh \\ E; \mathcal{M}_1; l \vdash \texttt{arr} < a, t[\,]^1 \ldots [\,]^{n-1} > \Rightarrow_e\ \mathcal{M}_2; \sigma \end{array}}{\begin{array}{ll} E; \mathcal{M}_1; l \vdash \texttt{decl}\ t[\,]^1 ... [\,]^n\ x\ \Rightarrow_e & \Gamma_2[x := (\{a\}, t[\,]^1 ... [\,]^n, l)] \\ & ; H_2; S_2; (\{a\}, t[\,]^1 ... [\,]^n, l) \end{array}}$$

(E-Arr)

$$\frac{\begin{array}{c} b\ fresh \\ E; \mathcal{M}_1; l \vdash \texttt{arr} < b, t[\,]^1 \ldots [\,]^{n-1} > \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma \end{array}}{\begin{array}{ll} E; \mathcal{M}_1; l \vdash \texttt{arr} < a, t[\,]^1 \ldots [\,]^n > & \Rightarrow_e \\ & \Gamma_2 \\ & ; H_2[\ a := \\ & [cont : \\ & (\{b\}, t[\,]^1 \ldots [\,]^n, l)]\ ] \\ & ; S_2; \mathsf{U} \end{array}}$$

(E-Arr-Vacio)

$$\frac{}{E; \mathcal{M}_1; l \vdash \texttt{arr} < a, t > \quad \Rightarrow_e \quad \Gamma_2; H_2[a := [cont : \mathrm{PD}()]]; S_2; \mathsf{U}}$$

- **Operaciones binarias.** Para analizar una operación binaria se toma el supremo de las dos expresiones.

$$\frac{\begin{array}{lll} E; \mathcal{M}_1; l \vdash e_1 & \Rightarrow_e & \mathcal{M}_2; \sigma_1 \\ E; \mathcal{M}_2; l \vdash e_2 & \Rightarrow_e & \mathcal{M}_3; \sigma_2 \end{array}}{E; \mathcal{M}_1; l \vdash e_1\ op\ e_2 \quad \Rightarrow_e \quad \mathcal{M}_3; \sigma_1 \sqcup \sigma_2}\ \text{E-OperacionBin}$$

- **Operciones unarias.** El valor de seguridad de una expresión con una operación unaria es el mismo que el de la expresión sin la operación.

$$\frac{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash op\ e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}\ \text{E-OperacionUn-Pre}$$

$$\frac{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}; l \vdash e\ op \quad \Rightarrow_e \quad \mathcal{M}_1; \sigma}\ \text{E-OperacionUn-Pos}$$

- **Casting.** A continuación presentamos dos situaciones a contemplar en el análisis de expresiones que involucran un cast. Los casts para tipos primitivos no influyen en el análisis.

  - *Cast válido.* El siguiente esquema se usa cuando un objeto se castea a su propia clase o a una de sus superclases. Habrá un error en tiempo de ejecución si casteamos un objeto a una de sus subclases o a una clase que no tiene relación con el objeto.

    $$\frac{\begin{array}{c}\text{C es D o una de sus superclases} \\ E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2)\end{array}}{E; \mathcal{M}_1; l_1 \vdash (C)e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2)}\ \text{E-Cast-Arriba}$$

  - *Métodos del sistema.* Como los métodos del sistema no son analizados, creamos como valor de retorno un objeto de la clase de retorno. Pero el verdadero resultado del método podría ser un objeto de una subclase de la clase de retorno. Por esto, en la ejecución simbólica del programa podría haber un cast a una subclase del objeto. Por ejemplo:

```
1   m(Vector<String> files) {
2     Iterator iter = files.iterator();
3     while(iter.hasNext()) {
4       String file = (String) iter.next();
5       this.m2(file);
6     }
7   }
8
9   m2(String file) {
10    file = file.trim();
11    ...
12  }
```

    El valor que retorna el análisis para `iter.next()`, un método del sistema, en la línea 4 es un objeto nuevo de la clase `Object`, el tipo de retorno del método. El programa luego castea este objeto a `String` para enviarlo como parámetro al método $m2$.

    Para resolver este problema, creamos un objeto de la clase que se requiere, `String` en este caso, como lo indica el siguiente esquema:

(E-Cast-Abajo)

$$\frac{\begin{array}{c}\text{C es una subclase de D}\\ E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, D, l_2) \\ E; \mathcal{M}_2; l_1 \vdash \texttt{new } C() \quad \Rightarrow_e \quad \mathcal{M}_3; (\{b\}, C, l_3)\end{array}}{E; \mathcal{M}_1; l_1 \vdash (C)e \quad \Rightarrow_e \quad \mathcal{M}_3; (\{b\}, C, l_3)}$$

- **Null.** El análisis de `null` simplemente retorna el nivel de seguridad del contador de programa y deja el estado de memoria inalterado.

$$\frac{}{E; \mathcal{M}; l \vdash \texttt{null} \quad \Rightarrow_e \quad \mathcal{M}; l}\ \text{E-Null}$$

- **InstanceOf.** El análisis de `instanceOf` analiza la expresión y, dado que el resultado es booleano, devuelve el nivel de seguridad del contador de programa.

$$\frac{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash e \texttt{ instanceOf } C \quad \Rightarrow_e \quad \mathcal{M}_2; l}\ \text{E-InstanceOf}$$

- **IfThenElse.** Como mencionamos anteriormente, analizamos una expresión condicional analizando cada rama independientemente y luego tomamos el supremo de ambas.

  (E-IfThenElse)

$$\frac{\begin{array}{c}E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma_1 \\ E; \mathcal{M}_2; l \sqcup \sigma_1 \vdash e_1 \quad \Rightarrow_e \quad \mathcal{M}_3; \sigma_2 \\ E; \mathcal{M}_2; l \sqcup \sigma_1 \vdash e_2 \quad \Rightarrow_e \quad \mathcal{M}_4; \sigma_3\end{array}}{E; \mathcal{M}_1; l \vdash e \ ?\ e_1 : e_2 \quad \Rightarrow_e \quad (\mathcal{M}_3; \sigma_2) \sqcup (\mathcal{M}_4; \sigma_3)}$$

A.0.2. Esquemas de análisis para comandos

En esta sección se detallan los esquemas que definen el juicio de análisis para comandos:

$$\boxed{E; \mathcal{M}_1; B_1; BL_1; \mathcal{C}; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2; B_2; BL_2 \mid \mathcal{R} \mid \mathcal{E}}$$

Varios de estos esquemas se basan en los de expresiones. En estos casos omitimos más explicaciones y en cambio remitimos al lector a la explicación dada más arriba para las expresiones correspondientes.

Para simplificar la notación mencionamos B, BL, $\mathcal{C}$, $\mathcal{R}$ y $\mathcal{E}$ sólo en los esquemas que los modifican directamente. Cuando $\mathcal{R}$ y $\mathcal{E}$ no están presentes, sus valores se asumen $\emptyset$.

- **Asignación, asignación de campo, declaración, declaración de arreglo, invocación a método.** El siguiente esquema se usa para comandos que pueden ser expresiones: $x = e$, $x.f = e$, `super`.$f = e$, `decl` $x = e$, `decl` $C[\,]^1...[\,]^n\ x = e$, $e.m(e_1, ..., e_n)$, `super`.$m(e_1, ..., e_n)$, $C.m(e_1, ..., e_n)$.

$$\frac{E; \mathcal{M}_1; l \vdash c_e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma_1}{E; \mathcal{M}_1; l \vdash c_e \quad \Rightarrow_c \quad \mathcal{M}_2}$$

- **IfThenElse.** El análisis del comando if al igual que el de la expresión if consiste en obtener el supremo de los valores obtenidos al analizar independientemente cada rama.

  (C-IfThenElse)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l \vdash e \ \Rightarrow_e \ \mathcal{M}_2; \sigma \\ E; \mathcal{M}_2; B_1; BL_1; l \sqcup \sigma \vdash c_1 \ \Rightarrow_c \ \mathcal{M}_3; B_2; BL_2 | \mathcal{R}_1 | \mathcal{E}_1 \\ E; \mathcal{M}_2; B_1; BL_1; l \sqcup \sigma \vdash c_2 \ \Rightarrow_c \ \mathcal{M}_4; B_3; BL_3 | \mathcal{R}_2 | \mathcal{E}_2 \end{array}}{\begin{array}{ll} E; \mathcal{M}_1; B_1; BL_1; l \vdash \texttt{if}\ (e)\ c_1\ \texttt{else}\ c_2 \ \Rightarrow_c & (\mathcal{M}_3; B_2; BL_2) \\ & \sqcup\ (\mathcal{M}_4; B_3; BL_3) \\ & |\mathcal{R}_1 \sqcup \mathcal{R}_2 | \mathcal{E}_1 \sqcup \mathcal{E}_2 \end{array}}$$

  Consideremos el siguiente ejemplo:

```
if (secreto)
  x = 1;
else
  x = 0;
```

  El nivel del contador de programa en las ramas del if es alto debido a que la condición está relacionada con un dato secreto. De este modo, el análisis asigna un valor de seguridad T a la variable x. Esto refleja el hecho de que la variable x pasa a no ser confiable porque su valor (0 o 1) permite a un atacante saber el valor de la variable *secreto* ($falso$ o *verdadero*).

- **IfThen.** El análisis del comando if sin else es similar al caso con else con la diferencia de que la alternativa a la rama then en este caso es el estado del análisis anterior a la ejecución de la rama then, para considerar el caso en que no se ejecuta el then.

  (C-IfThen)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma \\ E; \mathcal{M}_2; B_1; BL_1; l \sqcup \sigma \vdash c \quad \Rightarrow_c \quad \mathcal{M}_3; B_2; BL_2 \mid \mathcal{R} \mid \mathcal{E} \end{array}}{\begin{array}{ll} E; \mathcal{M}_1; B_1; BL_1; l \vdash \texttt{if}\ (e)\ c \quad \Rightarrow_c & (\mathcal{M}_2; B_1; BL_1) \\ & \sqcup (\mathcal{M}_3; B_2; BL_2) \\ & \mid \mathcal{R} \mid \mathcal{E} \end{array}}$$

- **While.** El análisis de un while consiste en analizar reiteradas veces su cuerpo hasta que se obtiene un punto fijo, es decir, no hay más modificaciones en los valores de seguridad de las variables. Para verificar que no hay cambios en el estado de memoria luego de una iteración, se usan las siguientes operaciones de estructuras que no consideran las referencias simbólicas. Esto es necesario porque, si hubiera un `new` en el cuerpo del while, en cada iteración el estado de memoria contendría distintas referencias simbólicas dado que el `new` crea una referencia nueva.

  - *Operación para eliminar referencias de ambientes de variables locales.*

$$\begin{array}{lcl} \widehat{\epsilon} & = & \epsilon \\ \widehat{x : l, \Gamma} & = & x : l, \widehat{\Gamma} \\ \widehat{x :< A, C, l >, \Gamma} & = & x :< C, l >, \widehat{\Gamma} \end{array}$$

  - *Operación para eliminar referencias de ambientes de variables estáticas.* Esta operación es anóloga a la de ambientes de variables locales.
  - *Comparación de heaps.*

$$\widehat{H_1} =_T \widehat{H_2} \quad \Leftrightarrow \quad \widehat{H_1}(A, C) = \widehat{H_2}(B, C), \quad \forall((A, B), C) \in T$$

$$T = (\Gamma_1, S_1) \oplus (\Gamma_2, S_2) = \{((A, B), \textsc{Tipo}(x)) \mid \begin{array}{l} x : (A, C, l_1) \\ \in \Gamma_1 \cup S_1 \\ \wedge \\ x : (B, D, l_2) \\ \in \Gamma_2 \cup S_2\} \end{array}$$

    El símbolo $\uparrow^C$ se usa para indicar que se desea considerar, del objeto en una posición dada por una referencia a, sólo los campos de la clase C y sus superclases.

$$\begin{array}{lcl} \widehat{H}(A, C) & = & \bigsqcup_j \widehat{H}(a_j, C) \quad , \quad a_j \in A \\ \widehat{H}(a, C) & = & [f_i : \widehat{H}(\sigma_i, \textsc{Tipo}(f_i))] \uparrow^C \\ \widehat{H}(l, C) & = & l \\ \widehat{H}((A, D, l), C) & = & < \widehat{H}(A, C), l > \end{array}$$

    Si tenemos $\Gamma_1(x) = (\{a, b\}, C, l)$ podríamos tener $H(a) = [f : \mathsf{U}]$ y $H(b) = [f : \mathsf{T}]$. Cuando accedemos al campo f de la variable x obtenemos el valor T que resulta de $[f : \mathsf{T}] = H(a) \sqcup H(b)$. Por esto, para $\widehat{H}(\{a, b\}, C)$ computamos el supremo entre $\widehat{H}(a, C)$ y $\widehat{H}(b, C)$.

A continuación se muestra cómo se implementa el esquema para el comando `while` (e) c. La función VarsE(c) devuelve las variables en el cuerpo c modificadas por una asignación o por un método.

$$
\begin{array}{l}
\textit{El estado inicial es} : E; \mathcal{M}_i; B_i; BL_i; l_i \\
\textit{El esquema } \text{C-While} \textit{ se implementa como sigue} : \\
\quad E; \mathcal{M}_i; l_i \vdash e \qquad \Rightarrow_e \qquad \mathcal{M}_p; l_p \\
\quad E; \mathcal{M}_p; B_i \cdot \emptyset; BL_i; l_p \vdash c \;\Rightarrow_c\; \mathcal{M}_f; B_i \cdot b_f; BL_f | \mathcal{R}_f | \mathcal{E}_f \\
\quad b_{sup} = b_f \\
\quad \mathcal{R}_{sup} = \mathcal{R}_f \\
\quad \mathcal{E}_{sup} = \mathcal{E}_f \\
\quad \textit{while } (\textit{not } (\widehat{\Gamma_i} = \widehat{\Gamma_f} \wedge \widehat{H_i} =_T \widehat{H_f} \wedge \widehat{S_i} = \widehat{S_f}))\{ \\
\qquad \mathcal{M}_i = \mathcal{M}_f \\
\qquad BL_i = BL_f \\
\qquad E; \mathcal{M}_i; l_i \vdash e \qquad \Rightarrow_e \qquad \mathcal{M}_e; l_e \\
\qquad E; \mathcal{M}_e; B_i \cdot \emptyset; BL_i; l_e \vdash c \;\Rightarrow_c\; \mathcal{M}_f; B_i \cdot b_f; BL_f | \mathcal{R}_f | \mathcal{E}_f \\
\qquad b_{sup} = b_{sup} \sqcup b_f \\
\qquad \mathcal{R}_{sup} = \mathcal{R}_{sup} \sqcup \mathcal{R}_f \\
\qquad \mathcal{E}_{sup} = \mathcal{E}_{sup} \sqcup \mathcal{E}_f \\
\quad \} \\
\textit{El estado final es} : \; (\\
\qquad \Gamma_f|_{\overline{\text{VarsE}(c)}} \quad \cup \\
\qquad (\Gamma_f|_{\text{VarsE}(c)} \sqcup \text{Nivel}(b_{sup})) \\
\qquad ; H_f \\
\qquad ; S_f|_{\overline{\text{VarsE}(c)}} \quad \cup \\
\qquad (S_f|_{\text{VarsE}(c)} \sqcup \text{Nivel}(b_{sup})) \\
\qquad) \sqcup \mathcal{M}_p \sqcup b_{sup} \\
\qquad ; B_i; BL_f \mid \mathcal{R}_{sup} \mid \mathcal{E}_{sup}
\end{array}
$$

El siguiente es un ejemplo de por qué no es suficiente con analizar sólo una vez el cuerpo de un while:

```
y = new D();
z = 0;
while (condicion) {
  z = y.m();
  y = new C();
}
```

El problema es que el método m en D podría ser inofensivo mientras que el método m en C podría ser peligroso. Por eso en este ejemplo otro análisis del cuerpo del while sería necesario.

El while con etiqueta se analiza como un while pero además se actualizan las variables modificadas en el cuerpo del while con el valor de seguridad de la etiqueta.

(C-While-E)

$$\frac{E; \mathcal{M}_1; BL_1[et := \emptyset]; l \vdash \texttt{while}\ (e)\ c\ \Rightarrow_c\ \mathcal{M}_2; BL_2 | \mathcal{R} | \mathcal{E}}{\begin{array}{l} E; \mathcal{M}_1; BL_1; l \vdash et :\ \texttt{while}\ (e)\ c\ \Rightarrow_c \\ \quad (\ \Gamma_2|_{\overline{\text{VarsE}(c)}}\ \cup \\ \quad (\ \Gamma_2|_{\text{VarsE}(c)} \\ \quad \sqcup \text{Nivel}(BL_2(et))\ ) \\ \quad ; H_2 \\ \quad ; S_2|_{\overline{\text{VarsE}(c)}}\ \cup \\ \quad (\ S_2|_{\text{VarsE}(c)} \\ \quad \sqcup \text{Nivel}(BL_2(et)))\ ) \\ \\ \quad \sqcup BL_2(et) \\ \quad ; BL_2 \mid \mathcal{R} \mid \mathcal{E} \end{array}}$$

El do while se analiza del mismo modo que un while.

(C-DoWhile)

$$\frac{E; \mathcal{M}_1; BL_1; l \vdash \texttt{while}\ (e)\ c \quad \Rightarrow_c \quad \mathcal{M}_2; BL_2 \mid \mathcal{R} \mid \mathcal{E}}{E; \mathcal{M}_1; BL_1; l \vdash \texttt{do}\ c\ \texttt{while}\ (e) \quad \Rightarrow_c \quad \mathcal{M}_2; BL_2 \mid \mathcal{R} \mid \mathcal{E}}$$

- **Secuencia.** En una secuencia se analizan los comandos respetando el orden. Además, los valores de seguridad de retorno y de las excepciones se obtienen realizando el supremo de los valores obtenidos en el análisis de cada comando.

  (C-Secuencia)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; B_1; BL_1; l \vdash c_1\ \Rightarrow_c\ \mathcal{M}_2; B_2; BL_2 | \mathcal{R}_1 | \mathcal{E}_1 \\ E; \mathcal{M}_2; B_2; BL_2; l \vdash c_2\ \Rightarrow_c\ \mathcal{M}_3; B_3; BL_3 | \mathcal{R}_2 | \mathcal{E}_2 \end{array}}{E; \mathcal{M}_1; B_1; BL_1; l \vdash c_1;\ c_2\ \Rightarrow_c\ \mathcal{M}_3; B_3; BL_3 | \mathcal{R}_1 \sqcup \mathcal{R}_2 | \mathcal{E}_1 \sqcup \mathcal{E}_2}$$

- **Break.** Tenemos dos esquemas para el comando break. El primero simplemente registra el estado de memoria actual (i.e. el que existe en el punto en que el break es ejecutado).

$$\frac{}{E; \mathcal{M}; B \cdot b; l \vdash \texttt{break} \quad \Rightarrow_c \quad \emptyset; B \cdot (b \sqcup (\mathcal{M}, l))}\ \text{C-Break}$$

En el caso de un break con etiqueta debemos registrar el estado de memoria actual para esa etiqueta en particular.

(C-Break-Et)

$$\frac{}{E; \mathcal{M}; BL; l \vdash \texttt{break}\ et \qquad \Rightarrow_c \qquad \emptyset; BL[et := (BL(et) \sqcup (\mathcal{M}, l))]}$$

El siguiente ejemplo muestra por qué necesitamos una pila para considerar comandos `break` dentro de comandos `while`:

```
j = 0;
while (j < 2) {
  j ++;
  i = 0;
  while (i < 2) {
    i ++;
    if (secreto)
      break;
  }
}
// i vale 1 si secreto es true
```

En este ejemplo el nivel de seguridad del break en el while de más adentro no afecta la variable j en el while de más afuera.

El siguiente ejemplo muestra cómo `break` afecta el nivel de seguridad de las variables escritas en el cuerpo del comando while:

```
y = 5;
x = 0;
while (y > 0) {
  y = y - 1;
  if (secreto)
    break;
  x = x + 1;
}
// x vale 5 si secreto es false
// x vale 0 si secreto es true
```

A continuación hay un ejemplo de `break` usado con *etiqueta*, en este caso los niveles de seguridad de las variables de ambos cuerpos son afectados:

```
      i = 0;
etiq: while (i < 2) {
        i ++;
        j = 0;
```

```
  while (j < 2) {
    if (secreto)
      break etiq;
    j ++;
  }
}
\\ i vale 1 y j vale 0 si secreto es true
```

- **Continue.** Se requiere cierto cuidado cuando se trata con comandos `continue` dado que también pueden cambiar el flujo de control. A continuación se presenta el esquema para este comando. La función VarsC() devuelve las variables escritas que le siguen a un `continue` dentro de un bloque.

  (C-Continue)

$$\frac{}{E;\mathcal{M};l \vdash \texttt{continue} \quad \Rightarrow_c \quad \begin{array}{l}\Gamma|_{\overline{\text{VarsC}()}} \cup (\Gamma|_{\text{VarsC}()} \sqcup l) \\ ;H \\ ;S|_{\overline{\text{VarsC}()}} \cup (S|_{\text{VarsC}()} \sqcup l)\end{array}}$$

  El siguiente ejemplo muestra cómo `continue` afecta el nivel de seguridad de las variables escritas después de `continue`:

```
x = 0;
while (e) {
  System.out.print(x);
  if (secreto) {
    continue;
  }
  x = 1;
}
// Imprime 1 en pantalla si secreto es false
```

- **For.** El comando for se puede ver como un while donde antes se hace la inicialización y al final del bloque se ejecutan los comandos que modifican la forma de iterar. Por lo tanto, se reusa el análisis visto para el while.

  (C-For)

$$\frac{E;\mathcal{M}_1;l \vdash \{e_1;...;e_i;\texttt{while}(e)\ \{c;\ e_{i+1};...;e_n\}\} \ \Rightarrow_c\ \mathcal{M}_2|\mathcal{R}|\mathcal{E}}{E;\mathcal{M}_1;l \vdash \texttt{for}\ (e_1,...,e_i;\ e;\ e_{i+1},...,e_n)\ c\ \Rightarrow_c\ \mathcal{M}_2|\mathcal{R}|\mathcal{E}}$$

- **Switch.** En el comando switch se presenta el hecho de tener distintas ramas posibles para continuar la ejecución. Al igual que en el if, se

toma el supremo de los valores de seguridad que resultan de analizar independientemente cada rama.

El esquema de análisis principal que se presenta a continuación hace uso de otros esquemas auxiliares que contemplan las distintas configuraciones de un switch.

(C-Switch)

$$\frac{\begin{array}{c} E;\mathcal{M}_1;l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_s;\sigma_s \\ E;\mathcal{M}_s;l \sqcup \sigma_s \vdash (\mathcal{M}_s, \texttt{sw} < c_1,\ldots,c_{k+1} >) \quad \Rightarrow_c \quad \mathcal{M}_2|\mathcal{R}|\mathcal{E} \end{array}}{\begin{array}{l} E;\mathcal{M}_1;l \vdash \texttt{switch}\ (e)\ \{ \quad \Rightarrow_c \quad \mathcal{M}_2|\mathcal{R}|\mathcal{E} \\ \quad \texttt{case}\ e_1^1 :\ \ldots\ \texttt{case}\ e_{n_1}^1 :\ c_1 \\ \quad \vdots \\ \quad \texttt{case}\ e_1^k :\ \ldots\ \texttt{case}\ e_{n_k}^k :\ c_k \\ \quad \texttt{default} :\ c_{k+1} \\ \} \end{array}}$$

Las expresiones e_i^j no son analizadas porque representan valores constantes.

Los esquemas auxiliares definen el significado del siguiente juicio de análisis auxiliar:

$$\boxed{E;\mathcal{M}_1;l \vdash (\mathcal{M}_s, \texttt{sw} < c_1,\ldots,c_n >) \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E}}$$

Un comando c termina en `break`, `return`, `throw` si todas las hojas en el grafo de flujo de c contienen un comando `break`, `return` o `throw`. Esto se relaciona con la semántica del switch, se usa para saber si después de un comando se sale del switch o se continúa con el siguiente comando.

Los esquemas auxiliares son:

- Para el caso de una lista de comandos donde el primero (c_1) no termina en `break`, `return`, `throw`, tenemos dos esquemas, uno para la lista de más de un comando y otro para la lista unitaria.
  - c_1 no es el último comando.
    (C-Sw)

$$\frac{\begin{array}{c} E;\mathcal{M}_1;l \vdash c_1 \ \Rightarrow_c \ \mathcal{M}_2|\mathcal{R}_1|\mathcal{E}_1 \\ E;\mathcal{M}_2;l \vdash (\mathcal{M}_s, \texttt{sw} < c_2,...,c_n >) \ \Rightarrow_c \ \mathcal{M}_3|\mathcal{R}_2|\mathcal{E}_2 \\ E;\mathcal{M}_s;l \vdash (\mathcal{M}_s, \texttt{sw} < c_2,...,c_n >) \ \Rightarrow_c \ \mathcal{M}_4|\mathcal{R}_3|\mathcal{E}_3 \end{array}}{\begin{array}{l} E;\mathcal{M}_1;l \vdash (\mathcal{M}_s, \texttt{sw} < c_1,...,c_n >) \ \Rightarrow_c \quad \mathcal{M}_3 \sqcup \mathcal{M}_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad |\mathcal{R}_2 \sqcup \mathcal{R}_3|\mathcal{E}_2 \sqcup \mathcal{E}_3 \end{array}}$$

- c_1 es el último comando.
  (C-Sw-Ultimo)

$$\frac{E; \mathcal{M}_1; l \vdash c_1 \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R}_1 \mid \mathcal{E}_1 \qquad E; \mathcal{M}_s; l \vdash (\mathcal{M}_s, \texttt{sw} <>) \quad \Rightarrow_c \quad \mathcal{M}_3 | \mathcal{R}_2 | \mathcal{E}_2}{E; \mathcal{M}_1; l \vdash (\mathcal{M}_s, \texttt{sw} < c_1 >) \quad \Rightarrow_c \quad \begin{array}{l} \mathcal{M}_2 \sqcup \mathcal{M}_3 \\ \mid \mathcal{R}_1 \sqcup \mathcal{R}_2 \\ \mid \mathcal{E}_1 \sqcup \mathcal{E}_2 \end{array}}$$

- Cuando el primer comando (c_1) de la lista de comandos termina en `break`, `return`, `throw`, usamos el siguiente esquema:
  (C-Sw-BRT)

$$\frac{E; \mathcal{M}_1; B \cdot \emptyset; l \vdash c_1 \quad \Rightarrow_c \quad \emptyset; B \cdot (\mathcal{M}_b, l_b) \mid \mathcal{R}_1 \mid \mathcal{E}_1 \qquad E; \mathcal{M}_s; l \vdash (\mathcal{M}_s, \texttt{sw} < c_2, \ldots, c_n >) \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R}_2 \mid \mathcal{E}_2}{E; \mathcal{M}_1; B; l \vdash (\mathcal{M}_s, \texttt{sw} < c_1, \ldots, c_n >) \quad \Rightarrow_c \quad \begin{array}{l} \mathcal{M}_b \sqcup \mathcal{M}_2; \\ B \\ \mid \mathcal{R}_1 \sqcup \mathcal{R}_2 \\ \mid \mathcal{E}_1 \sqcup \mathcal{E}_2 \end{array}}$$

- Cuando no quedan más comandos para analizar tenemos:
  - Si la etiqueta `default` está presente:
    (C-Sw-Vacio-Default)

$$\frac{}{E; \mathcal{M}; l \vdash (\mathcal{M}_s, \texttt{sw} <>) \quad \Rightarrow_c \quad \emptyset}$$

  - Si la etiqueta `default` está ausente consideramos cuando la expresión del switch no coincide con ningún case:

$$\frac{}{E; \mathcal{M}; l \vdash (\mathcal{M}_s, \texttt{sw} <>) \quad \Rightarrow_c \quad \mathcal{M}_s} \text{ C-Sw-Vacio}$$

- **Bloque de código.** El análisis de un bloque de código consiste en agregar las variables declaradas en el bloque al ambiente de variables locales, analizar el comando dentro del bloque y luego quitar las variables del bloque del ambiente de variables locales.

  (C-Bloque)

$$\frac{E; \Gamma_1 \cup \textsc{Decl}(c); H_1; S_1; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E}}{E; \mathcal{M}_1; l \vdash \{\ c\ \} \quad \Rightarrow_c \quad \Gamma_2 - \Gamma_2|_{\textsc{Decl}(c)}; H_2; S_2 \mid \mathcal{R} \mid \mathcal{E}}$$

- **Return.** Para el comando return se analiza la expresión que contiene y se devuelve como valor de seguridad de retorno el valor de esa expresión (σ) y el estado de memoria resultante sin incluir el ambiente

de variables locales (H_2, S_2). Como es una salida del método, devuelve un estado de memoria con el valor neutro del supremo de estructuras ($\emptyset$), lo mismo para las excepciones ya que es una salida del método que no es excepcional.

$$\frac{E; \mathcal{M}_1; l \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; \sigma}{E; \mathcal{M}_1; l \vdash \texttt{return}\ e \quad \Rightarrow_c \quad \emptyset \mid (H_2, S_2, \sigma) \mid \emptyset}\ \text{C-RETURN-EXP}$$

Para la información de la tabla de seguridad se considera el σ como uno de los valores de retorno posibles. CAMPOSRET devuelve los campos del objeto retornado y campos de los objetos contenidos dentro de esos campos, y así siguiendo. Estos campos se agregan a H_{out} debido a que son visibles fuera del método.

$$\frac{\mathcal{T}_1 \vdash e \quad \Rightarrow_e \quad \mathcal{T}_2; \sigma}{\begin{array}{ll}\mathcal{T}_1 \vdash \texttt{return}\ e & \Rightarrow_e \quad \mathcal{T}_2\ [\sigma_{return} \sqcup = \ \sigma] \\ & \quad\ [H_{out} \cup = \text{CAMPOSRET}()]\end{array}}$$

- **Try.** El análisis del comando try-catch consiste en analizar el comando del cuerpo (c) con la lista L de pares (E_i, c_i) que se corresponde con los manejadores de excepciones. Esta lista se tiene en cuenta en el comando throw.

  (C-TRY)

$$\frac{E; \mathcal{M}_1; \mathcal{C} \cdot L; l \vdash c \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E}}{\begin{array}{ll}E; \mathcal{M}_1; \mathcal{C}; l \vdash \texttt{try}\ c & \texttt{catch}\ (E_1\ e)\ c_1 \quad \Rightarrow_c \quad \mathcal{M}_2 \mid \mathcal{R} \mid \mathcal{E} \\ & \vdots \\ & \texttt{catch}\ (E_n\ e)\ c_n \\ & [\texttt{finally}\ c_f]\end{array}}$$

- **Throw.** A continuación se presentan varios esquemas de análisis para el comando throw que consideran los distintos contextos, si hay un manejador para la excepción y si la cláusula finally fue incluida.

  - No hay ningún manejador ni cláusula finally. (C-THROW)

$$\frac{E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2)}{\begin{array}{ll}E; \mathcal{M}_1; \emptyset; l_1 \vdash \texttt{throw}\ e & \Rightarrow_c \quad \emptyset \\ & \quad\ \mid \emptyset \\ & \quad\ \mid \emptyset[Ex := (H_2, S_2, (A, Ex, l_2))]\end{array}}$$

- No hay manejador ni cláusula finally en el comando try-catch inmediato. (C-Throw-NC-NF)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ Ex \notin L \\ E; \mathcal{M}_2; \mathcal{C}; l_1 \sqcup l_2 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- No hay manejador pero sí una cláusula finally en el comando try-catch inmediato. La función BuscarFinally devuelve el comando c_f que se encuentra en la cláusula finally.
  (C-Throw-NC-F)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ Ex \notin L \\ c_f = \text{BuscarFinally}(L) \\ E; \mathcal{M}_2; \mathcal{C}; l_1 \sqcup l_2 \vdash c_f;\ \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- Existe un manejador pero no hay cláusula finally en el comando try-catch inmediato. x_{Ex} es la variable que aparece en la cláusula catch de la excepción Ex.
  (C-Throw-C-NF)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ (Ex, c_{Ex}) \in L \\ E; \Gamma_2[x_{Ex} := (A, Ex, l_2)]; H_2; S_2; \mathcal{C}; l_1 \sqcup l_2 \vdash c_{Ex} \ \Rightarrow_c \ \mathcal{M}_3 | \mathcal{R} | \mathcal{E} \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

- Existen ambos un manejador y una cláusula finally en el comando try-catch inmediato.
  (C-Throw-C-F)

$$\frac{\begin{array}{c} E; \mathcal{M}_1; l_1 \vdash e \quad \Rightarrow_e \quad \mathcal{M}_2; (A, Ex, l_2) \\ (Ex, c_{Ex}) \in L \\ c_f = \text{BuscarFinally}(L) \\ E; \Gamma_2[x_{Ex} := (A, E, l_2)] \ \vdash c_{Ex};\ c_f \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E} \\ ; H_2; S_2; \mathcal{C}; l_1 \sqcup l_2 \end{array}}{E; \mathcal{M}_1; \mathcal{C} \cdot L; l_1 \vdash \texttt{throw}\ e \quad \Rightarrow_c \quad \mathcal{M}_3 \mid \mathcal{R} \mid \mathcal{E}}$$

Bibliografía

[1] C. Kiran. *The web development industry is expected to grow over 20 by 2010*. URL:

`http://www.articler.com/23205/The-Web-Development-Industry-Is-Expected-To-Grow-Over-20-By-2010.html`

[2] A. van der Stock, J. Williams y D. Wichers. *The ten most critical web application security vulnerabilities*. OWASP technical report. 2007.

[3] G. McGraw. *Software security: Building security in*. Addison-Wesley. 2006.

[4] A. Sabelfeld y A. C. Myers. *Language-based information-flow security*. IEEE Journal on Selected Areas in Communications. 2003.

[5] J. Gosling, B. Joy, G. Steele y G. Bracha. *The Java language specification, third edition*. 2005.

[6] A. Myers. *Practical mostly-static information flow control*. Proceedings of the 26th ACM Symposium on Principles of Programming Languages, páginas 228–241. 1999.

[7] A. Banerjee y D. A. Naumann. *Secure information flow and pointer confinement in a Java-like language*. Proceedings of 15th IEEE Computer Security Foundations Workshop, páginas 253-267. IEEE Computer Society. 2002.

[8] J. C. King. *Symbolic execution and program testing*. Communications of the ACM, 19(7), páginas 385–394. 1976.

[9] G. Smith y D. Volpano. *Secure information flow in a multi-threaded imperative language*. ACM Symposium on Principles of Programming Languages, páginas 355–364. 1998.

[10] A. Sabelfeld y D. Sands. *Probabilistic noninterference for multi-threaded programs*. Proceedings of 13th IEEE Computer Security Foundations Workshop, páginas 200–215. 2000.

[11] A. Russo. *Language support for controlling timing-based covert channels*. Chalmers University of Technology, Doctoral thesis ISBN/ISSN: 978-91-7385-171-8. 2008.

[12] V. B. Livshits y M. S. Lam. *Detecting security vulnerabilities in Java applications with static analysis*. Technical report. Stanford University. 2005. URL:

`http://suif.stanford.edu/~livshits`
`/papers/tr/webappsec_tr.pdf`

[13] Q. Sun, A. Banerjee y D. A. Naumann. *Modular and constraint-based information flow inference for an object-oriented language*. Proceedings of the 11th International Static Analysis Symposium, volume 3148 of Lecture Notes in Computer Science, páginas 84–99. Springer. 2004.

[14] E. Bonelli, A. B. Compagnoni y R. Medel. *Information flow analysis for a typed assembly language with polymorphic stacks*. Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Second International Workshop, volume 3956 of Lecture Notes in Computer Science, páginas 37–56. Springer. 2005.

[15] R. Medel, A. B. Compagnoni y E. Bonelli. *A typed assembly language for non-interference*. Theoretical Computer Science, 9th Italian Conference, volume 3701 of Lecture Notes in Computer Science, páginas 360–374. Springer. 2005.

[16] F. Bavera y E. Bonelli. *Typed-based information flow analysis for bytecode languages with variable object field policies*. Proceedings of the 2008 ACM Symposium on Applied Computing, páginas 347–351. ACM. 2008.

[17] V. B. Livshits y M. S. Lam. *Finding security errors in Java programs with static analysis*. Proceedings of the 14th Usenix Security Symposium, páginas 271–286. 2005.

[18] M. Martin, V. B. Livshits y M. S. Lam. *Finding application errors and security flaws using PQL: a program query language*. Proceedings of the 20th annual ACM SIGPLAN conference on object oriented programming languages and applications, páginas 365–383. 2005.

[19] Y. Huang, F. Yu, C. Hang, C. Tsai, D. Lee y S. Kuo. *Securing web application code by static analysis and runtime protection*. Proceedings of the 13th international conference on World Wide Web, páginas 40-52. ACM Press New York, NY, USA. 2004.

[20] Y. Xie y A. Aiken. *Static detection of security vulnerabilities in scripting languages*. Proceedings of the 15th USENIX Security Symposium. 2006.

[21] N. Jovanovic, C. Kruegel y E. Kirda *Precise alias analysis for static detection of web application vulnerabilities*. ACM SIGPLAN Workshop on Programming Languages and Analysis for Security. 2006.

[22] A. Futoransky y A. Waissbein. *Enforcing privacy in web applications*. 3rd Annual Conference on Privacy, Security and Trust. 2005.

[23] G. T. Buehrer, B. W. Weide y P. A. G. Sivilotti. *Using parse tree validation to prevent SQL injection attacks*. Proceedings of the 5th international workshop on software engineering and middleware, páginas 106–113. 2005.

[24] W. G. J. Halfond y A. Orso. *Combining static analysis and runtime monitoring to counter SQL-injection attacks*. Workshop on Dynamic Analysis (WODA), páginas 1–7. 2005.

Índice alfabético