Trabajo de Grado

Especialización de GBA para seguridad en la web

Bontempi, María Paula
Bormape, Sabrina

Director
Dr. Pablo E. Martínez López

5 de noviembre de 2007

Agradecimientos

En primer lugar queremos agradecerle a Fidel por tenernos en cuenta a la hora de formar su equipo de investigación confiando ciegamente en nosotras y permitiéndonos crecer profesionalmente, y especialmente por dirigirnos en este trabajo de grado, acompañándonos y guiándonos en todo momento con mucha calidad y responsabilidad.

Le agradecemos a LIFIA por formarnos y capacitarnos profesionalmente.

A la empresa CORE, por permitirnos desarrollar este trabajo de grado incentivando la investigación y por instruirnos en el área de seguridad.

Le agradecemos también a los jurados, Claudia Pons, Javier Díaz y Fernando Tinetti, por el tiempo dedicado a leer y corregir este trabajo.

A nuestros compañeros de facultad por permitirnos compartir mucho más que millones de horas de estudio y por hacernos dar cuenta de que no somos las únicas locas en este mundo como así lo creen nuestros familiares y amigos.

A todos aquellos profesores que con su responsabilidad e interés en la educación marcaron un antes y un después en nuestra carrera.

A nuestras familias, en especial a nuestros padres, por apoyarnos y alentarnos durante todo el transcurso de la carrera tanto en los buenos como en los malos momentos, soportando nuestras locuras y respetando nuestros momentos de concentración.

A todos nuestros amigos que estuvieron a nuestro lado durante todos estos años por escucharnos, acompañarnos, distraernos y bancarnos incondicionalmente.

A nuestros compañeros de trabajo, por enseñarnos y escucharnos en todo momento.

Personalmente, yo Sabrina, le quiero agradecer a Lucas, mi novio, por acompañarme y apoyarme durante estos últimos años de la carrera y por estar siempre a mi lado alentándome y dándome fuerzas hasta en los peores momentos. A Jorgito, que junto con mi mamá, me apoyaron y alentaron desde el principio. También le agradezco a mi compañera y amiga del alma Paula por compartir conmigo todos estos años, no sólo en lo que respecta a la carrera sino también a mi vida personal. Gracias Pau!

Finalmente, yo Paula, le quiero agradecer a mis abuelos, a Luciano y a su familia por tantos años de aliento. A Sandra, por el apoyo y el aguante

de cada mañana. A vos Sabri, por todos los momentos que compartimos, imposibles de expresar con palabras, pero que forman parte del verdadero sentido de tantos años de esfuerzo. A Gustavo, mi jefe, por permitirme crecer profesionalmente. Y a todas aquellas personas que siempre estuvieron presentes y que vivieron conmigo el transcurso de mi carrera.

Índice general

Capítulo 1

Introducción

En este trabajo presentaremos una técnica de análisis estático y una propuesta de modificaciones a la misma para utilizarla en el área de seguridad de aplicaciones web. Para esto, primero vamos a describir el problema de seguridad y privacidad existente en las aplicaciones web. Luego introduciremos las vulnerabilidades de seguridad y nos concentraremos principalmente en explicar y ejemplificar las vulnerabilidades de inyección, que son un tipo particular de vulnerabilidades de seguridad que aparecen en los códigos de las aplicaciones web con mucha frecuencia y por esta razón estamos interesados en detectarlas. A continuación daremos una idea general sobre la teoría de Sistemas de Tipos, para luego explicar una de las técnicas basadas en Sistemas de Tipos, Análisis de Expresiones String Basado en Gramáticas (llamada GBA por su traducción al inglés) y las modificaciones necesarias de la misma para poder utilizarla en la detección de vulnerabilidades de inyección.

1.1. Descripción del Problema

Las aplicaciones web han experimentado un crecimiento exponencial durante los últimos años y se han convertido en un estándar para brindar servicios online, desde foros de discusión hasta áreas sensibles de seguridad como ventas y actividades bancarias y gubernamentales. Si bien la facilidad de accederlas con un simple navegador web y una conexión a Internet es uno de los grandes beneficios que aporta el medio en el que se encuentran, la exposición al mundo entero aumenta la posibilidad de ataques que atenten contra la seguridad y comprometan al sistema. Las vulnerabilidades en el código de la aplicación, es decir los defectos en la implementación que podrían ser explotados por un atacante para obtener beneficios salteando la lógica de la aplicación, representan una amenaza creciente tanto para los proveedores como para los usuarios de estos servicios, tal como se explica sintéticamente en la sección 1.2 y más profundamente en el capítulo 2.

El propósito de este Trabajo de Grado es estudiar la utilización de una técnica de análisis estático, llamada Análisis de Expresiones String Basado en Gramáticas (GBA, por su nombre en inglés) para detectar vulnerabilidades de seguridad de tipo inyección en una aplicación web; la misma se presenta brevemente en la sección 1.4 y totalmente en el capítulo 3.

El trabajo se enmarca dentro del proyecto "Plataforma híbrida de seguridad para protección de aplicaciones web", del cual formamos parte. Este proyecto fue desarrollado entre LIFIA y la sección Corelabs de la empresa Core Security Technologies desde el mes de marzo del año 2006 hasta el mes de junio del año 2007. El mismo consideró diversas técnicas de Análisis Estático combinadas con Análisis Dinámico (de allí el término "híbrida" incluído en el nombre del mismo) con el objetivo de descubrir vulnerabilidades de inyección, como fuera descripto. Entre las técnicas consideradas se incluyó GBA, dado que la misma se basa en el análisis de strings, los cuales son de gran importancia al momento de analizar la seguridad de una aplicación web.

1.2. Vulnerabilidades

En los últimos años hemos visto un constante incremento en la importancia de la seguridad a nivel de aplicación, es decir en aquellas vulnerabilidades que afectan a la aplicación en lugar de al sistema operativo o al middleweare de los sistemas de computación. Dichas vulnerabilidades son defectos de programación que permiten a un atacante alterar el comportamiento de la aplicación en beneficio propio. De manera más específica, son imperfecciones de código de las cuales se pueden valer usuarios mal intencionados para revelar información confidencial, realizar operaciones privilegiadas, etc.

En el contexto de aplicaciones web, se producen típicamente por errores en la validación de la entrada ingresada por el usuario y manejo incorrecto de los requerimientos enviados. Existen diferentes tipos de vulnerabilidades relacionadas con la validación de la entrada: Buffer overflow, Integer overflow, vulnerabilidades de inyección, etc., todas ellas provenientes de suposiciones implícitas hechas por el desarrollador sobre diferentes características que presentan las entradas. Las vulnerabilidades de buffer overflow resultan de suposiciones inválidas hechas sobre el tamaño máximo de la entrada. Los ataques de integer overflow resultan de suposiciones inválidas hechas sobre el rango de la entrada. De manera similar, las vulnerabilidades de inyección resultan de suposiciones inválidas hechas sobre la presencia de cierto contenido sintáctico en la entrada de la aplicación. El contenido es considerado sintáctico cuando influye en la forma o la estructura de una expresión resultando en una alteración de la semántica de la misma.

En este trabajo nos concentramos en las vulnerabilidades de tipo inyección las cuales plantean una gran amenaza a nivel de seguridad de la aplica-

ción. Dichas vulnerabilidades permiten a un atacante alterar la semántica de una expresión cuando la entrada proporcionada posee contenido sintáctico como fuera mencionado. Algunos de los tipos más comunes son vulnerabilidades SQL injection, vulnerabilidades cross-site scripting y vulnerabilidades shell injection, las cuales se describen y ejemplifican en la sección 2.2.

Las consecuencias de los ataques de inyección, es decir, ataques que explotan este tipo de vulnerabilidades, consisten en omitir la lógica de autenticación, acceder o modificar información de una base de datos o ejecutar comandos arbitrarios en el servidor.

Existen dos soluciones posibles al problema de las vulnerabilidades en el código fuente de una aplicación: utilizar técnicas de construcción de programas que garanticen la ausencia de vulnerabilidades o realizar un análisis del código, posterior a la codificación, para detectar las mismas con el fin de eliminarlas. Dado que la primera alternativa es durante la codificación y que la empresa Core requiere un análisis posterior a la misma por el tipo de tarea que pretende automatizar, la auditoría de código, nosotros nos concentraremos en la segunda solución.

1.3. Análisis Estático

El análisis estático tiene como principal objetivo la obtención de propiedades del funcionamiento de un programa sin necesidad de ejecutarlo. Los Sistemas de Tipos Estáticos obtienen información de un programa estáticamente para determinar propiedades de los mismos antes de ejecución (por ejemplo, la ausencia de algunos errores de ejecución) y para proveer documentación rudimentaria cuando se codifican soluciones en un lenguaje de programación. La principal motivación para el uso de estos sistemas es que cada programa con un tipo calculado estáticamente (es decir, basado solamente en el texto del programa y no en su computación) está libre de algunas clases de errores durante la ejecución. Esto garantiza cierta corrección del código y también ayuda a omitir chequeos específicos en el código ejecutable, obteniendo un programa más eficiente.

Existen dos líneas de investigación que dependen del compromiso existente entre expresividad y decibilidad del sistema de tipos. Esto se debe a que la computación prescripta para un sistema de tipos es limitada; puede ocurrir que dado un sistema de tipos no exista un algoritmo de inferencia decidible. Por lo tanto, las opciones son diseñar un sistema decidible descartando un conjunto de programas correctos o diseñar un sistema preciso pero con una noción no computable de asignación de tipos. Cuando se elije la primer alternativa, el objetivo es maximizar la expresividad del sistema, esto es lograr que el número de programas aceptados correctos sea lo más grande posible, mientras que se minimiza el número de los incorrectos sin perder decibilidad. En esta línea de investigación aparecen los sistemas

de tipos polimórficos [Milner, 1978; Damas and Milner, 1982; Jim, 1996; Reynolds, 1983] y sus extensiones (como sistemas con subtipos [Mitchell, 1991; Palsberg *et al.*, 1997; Henglein and Rehof, 1997] o tipos recursivos [Tiuryn and Wand, 1993]). En el segundo caso, la idea es diseñar herramientas semi-automáticas para ayudar en la construcción de programas, maximizando la habilidad de ejecutar inferencia de tipo automática. En esta línea de investigación aparece el sistema F de Girard [Girard, 1989; Reynolds, 1974] y sistemas de tipos dependientes como la teoría de tipos de Martin Löf [Nordström *et al.*, 1990] y el Cálculo de Construcciones de Coquand [Coquand and Huet, 1988].

En este trabajo nos concentraremos en la primer alternativa dado que la misma corresponde a una solución de análisis y se adecua por ello a las características de la empresa Core como fuera descripto en la sección anterior (la segunda alternativa corresponde a un marco de construcción de programas seguros, pero requiere programadores con habilidades especiales).

Los Sistemas de Tipos son una parte integral de la definición de un lenguaje, de manera similar a la gramática que define la sintaxis de los programas. Además, así como hay una herramienta que implementa el parser de gramática, hay una herramienta que implementa el sistema de tipos. Esta última puede ser un comprobador de tipos (*typechecker*) o un inferidor de tipos. Un comprobador de tipos chequea que una construcción del lenguaje coincida con el tipo previsto en su contexto. Un inferidor de tipos determina el tipo de una construcción del lenguaje (quizás ayudado por anotaciones de tipo en el programa).

Existe una forma de presentar sistemas de inferencia de tipos mediante el uso de restricciones o contraints. En estos sistemas hay dos fases para la inferencia de tipos; la primera cuando se recorre el término y se recolectan las variables de tipo y una segunda fase de resolución de constraints, donde se calcula una solución para todos los constraints recolectados. Esta forma de presentación permite más flexibilidad para el uso de heurísticas durante la resolución de constraints. Esta variante fue usada, por ejemplo, en la teoría de tipos calificados para sobrecarga [Jones, 1996] y en la variante de Martínez López [Martínez López and Hughes, 2002] del sistema de especialización de tipos introducido por John Hughes [Hughes, 1996b].

Se han concebido Sistemas de Tipos para casi todos los tipos de paradigmas de programación y para casi todas las características; por ejemplo registros [Rèmy, 1989; Gaster and Jones, 1996], arreglos, variantes, sobrecarga [Jones, 1994a; Jones, 1994b; Odersky *et al.*, 1995; Jones, 1996], sistemas de tipos para características imperativas [Hoang *et al.*, 1993; Wright, 1992; Wright, 1995], lenguajes orientados a objetos [Abadi and Cardelli, 1996; Wand, 1987; von Oheimb and Nipkow, 1999; Syme, 1999; Nordlander, 1999; Nordlander, 2000], herencia [Breazu-Tannen *et al.*, 1991; Wand, 1994; Wand, 1991], persistencia [Atkinson *et al.*, 1983; Atkinson and Morrison, 1988; Dearle, 1988; Morrison *et al.*, 1993], tipos de datos abstractos [Mitchell and

Plotkin, 1988], sistemas reactivos [Hughes *et al.*, 1996; Nordlander, 1999], sistemas móviles [Knabe, 1995; Freund and Mitchell, 1998; Igarashi and Kobayashi, 2001] y otros [Pottier, 2000; Shields and Meijer, 2001] sólo en el campo de características de lenguaje.

Las técnicas basadas en sistemas de tipos han sido la inspiración de muchos desarrollos en el campo de análisis de programas y técnicas de validación y verificación. Estas técnicas de tipos han sido modificadas para ser aplicadas al cálculo de diferentes clases de efectos (sistemas de tipos y efectos [Talpin and Jouvelot, 1992; Jouvelot and Gifford, 1991]) –como por ejemplo, tipar referencias [Wright, 1992], alocación de registros [Agat, 1997], etc.–, análisis de control de flujo [Mossin, 1996; Volpano *et al.*, 1996], compilación [Wand, 1997], transformación de programas [Romanenko, 1990; Hannan and Hicks, 1988; Hughes, 1996b; Hughes, 1996a; Danvy, 1998], splicing de código [Thiemann, 1999], seguridad [Volpano and Smith, 1997] y diversas formas de análisis de programas [O'Callahan, 1998], entre otros.

1.4. Análisis de Strings Basado en Gramáticas

El Análisis de Expresiones String Basado en Gramáticas (GBA, por su nombre en inglés) es una técnica de análisis estático basada en Sistemas de Tipos que fuera presentada por Peter Thiemann en el paper "Grammar-Based Analysis of String Expressions" [Thiemann, 2005]. GBA analiza los strings determinando para cada ocurrencia de un tipo string en una derivación de tipos, un conjunto de strings tal que cada valor computado por la expresión correspondiente sea un miembro de este conjunto. El análisis es expresado como un sistema de tipos donde el tipo string es refinado con una estructura indexada apropiada.

El motivo que impulsó la creación de esta técnica está basado en el frecuente uso de los strings en contextos inapropiados. Particularmente en los lenguajes de scripting, los strings sirven como una estructura de datos universal para comunicar valores y comandos entre diferentes componentes de programas. Por ejemplo, los strings son usados para construir salida en formato XHTML, para formar sentencias SQL y para contruir expresiones XPath y JavaScript que son pasadas a sus intérpretes respectivos o almacenadas en un atributo de evento. Por lo tanto, los strings frecuentemente llevan un significado oculto que no está reflejado en su tipo (en un sistema tradicional).

En todos los casos anteriores, los strings deben respetar un formato externo fijo. Algunos ejemplos son los strings XHTML, las sentencias SQL, las sentencias XPath y JavaScript. En cada caso, hay una gramática libre de contexto que define el lenguaje de strings admisibles. Además, en cada caso se produce un error en tiempo de ejecución si el valor string actual no está correctamente formado (o sea no respeta las reglas de la gramática).

Los strings XHTML mal formados pueden hacer que el navegador web o un procesador XML usado posteriormente deje de funcionar. Las sentencias SQL mal formadas son rechazadas por la base de datos o conducen a resultados incorrectos. Finalmente, las expresiones ilegales pasadas a un procesador XPath o a un intérprete JavaScript causan excepciones en tiempo de ejecución.

El propósito de esta técnica es analizar expresiones strings de programas existentes de manera de aliviar el mantenimiento y debbuging asegurando que todos estos strings construidos sin una metodología apropiada respetan un formato deseado. GBA permite comprobar si todos los valores de una expresión string son elementos de una lenguaje libre de contexto dado por alguna gramática de referencia.

Este análisis de expresiones string está expresado como un sistema de tipos restringido donde las restricciones (o constraints) son parseadas con respecto a la gramática de referencia G. Intuitivamente, un constraint parseado expresa la relación de derivación de G, la cual se define en términos de símbolos terminales y no terminales de G.

Thiemann especifica un sistema de tipos polimórfico para un lambda cálculo aplicado que refina el tipo string con una jerarquía de subtipos derivada de la inclusión de lenguajes. Esto nos permite encontrar un lenguaje para cada expresión de tipo string tal que el valor de la expresión es un elemento de ese lenguaje. La inferencia de tipos para estos sistemas infiere constraints de inclusión del lenguaje que pueden ser vistos como una gramática libre de contexto con un no terminal para cada expresión evaluada a string. Thiemann luego presenta dos algoritmos que resuelven constraints de inclusión del lenguaje con respecto a una gramática de referencia libre de contexto. Las soluciones son correctas pero incompletas debido a que el problema general de inclusion de lenguaje libre de contexto es indecidible. Ambos algoritmos se derivan del algoritmo de parsing de Earley [Earley, 1970] para lenguajes libres de contexto.

Tomar estas dos partes juntas nos permite responder preguntas como "¿El valor de una expresión de tipo string es derivable de un no terminal dado en la gramática de referencia?" Este tipo de preguntas es interesante porque permite validar que la salida de un programa respeta ciertos formatos, como por ejemplo HTML, evitando de esa forma los problemas derivados del uso incorrecto de dicha salida.

1.5. Objetivos y Logros del Trabajo de Grado

Luego de haber analizado el problema y considerado diversas alternativas de solución (teniendo en cuenta la escasez de herramientas existentes para resolverlo), nos concentramos en analizar la técnica de GBA como posible solución.

El objetivo de este trabajo fue estudiar la técnica de GBA con el propósito de descubrir y proponer las modificaciones necesarias para aplicarla en la detección de vulnerabilidades de seguridad de tipo inyección como se describió anteriormente.

Para lograr el objetivo propuesto fue esencial estudiar y comprender adecuadamente tanto el problema en cuestión como la técnica utilizada. Respecto al estudio de la técnica, además de intentar comprender su funcionamiento de manera analítica, buscamos implementarlo para así integrar las ideas en un entorno operativo.

Logramos la implementación del algoritmo GBA modificado con una eficiencia razonable que permite el tratamiento de programas que sean representativos del problema, utilizando un lenguaje simple pero expresivo (basado en el propuesto por Thiemman).

El resultado consiste en la descripción de los conceptos aprendidos (tanto respecto del problema como de la técnica de solución) y en la descripción de los primeros pasos dados hacia la capacidad de utilizar la técnica en este problema.

1.6. Overview

El trabajo está organizado en siete capítulos. El capítulo 2 describe y ejemplifica las vulnerabilidades que fuimos aprendiendo durante la primera parte del proyecto antes mencionado. El capítulo 3 explica los sistemas de tipos y la técnica de análisis estático GBA presentada por Peter Thiemann en el paper “Grammar-Based Analysis of String Expressions”. El capítulo 4 explica nuestra propuesta de extensiones que permitan aplicar GBA para detectar vulnerabilidades. El capítulo 5 describe como realizamos la implementación del prototipo. El capítulo 6 explica como se usa el prototipo. El capítulo 7 describe técnicas existentes de análisis estático para detectar vulnerabilidades. Finalmente, en el capítulo 8 presentamos las conclusiones. Además se adjunta electrónicamente el código del prototipo de GBA escrito en Haskell.

Capítulo 2

Vulnerabilidades

En este capítulo presentaremos el problema de seguridad existente en las aplicaciones web [Xie and Aiken, 2006], concentrándonos principalmente en las vulnerabilidades a nivel de aplicación y dentro de ellas en las vulnerabilidades de validación de entrada, las cuales han sido presentadas brevemente en el capítulo 1.2. Luego presentaremos el problema de privacidad que afecta también a las aplicaciones web [Futoransky and Waissbein, 2005], y describiremos en detalle el proceso de provisión de privacidad necesario al momento de desarrollar una de las mismas. Finalmente, explicaremos las vulnerabilidades que estamos interesadas en detectar con nuestra herramienta que son las vulnerabilidades de inyección dando varios ejemplos de las mismas para su mejor comprensión y explicando sus causas [Huang *et al.*, 2004; OWASP Organization, 2003; Pietraszek and Berghe, 2005].

2.1. Seguridad y Privacidad en aplicaciones web

Las aplicaciones web han experimentado un crecimiento exponencial durante los últimos años y se han convertido en el estándar para brindar servicios en línea desde foros de discusión hasta áreas sensibles en seguridad como bancos y ventas. Por esta razón, las vulnerabilidades de seguridad en este tipo de aplicaciones representan una gran amenaza tanto para los proveedores como para los usuarios de estos servicios.

En los últimos años hemos visto un constante incremento en la importancia de la seguridad a nivel de aplicación, es decir en aquellas vulnerabilidades que afectan a la aplicación en lugar de al sistema operativo o al middleweare de los sistemas de computación. Hace unos años Symantec catalogó 670 vulnerabilidades que afectan a las aplicaciones web, observando un incremento del 81 % en el transcurso de un año [Symantec, 2005]. De acuerdo al mismo reporte, estas vulnerabilidades son causadas típicamente por errores de programación en la validación de la entrada y manejo incorrecto de los requerimientos enviados.

Debido a que las vulnerabilidades usualmente están profundamente embebidas en la lógica del programa, la defensa tradicional a nivel de red, como firewalls, no ofrece una protección adecuada contra estos ataques. El testing también es poco efectivo porque los atacantes usan típicamente la entrada menos esperada para explotar estas vulnerabilidades y comprometer el sistema.

Por todo lo expresado, la seguridad sigue siendo el principal obstáculo para la aceptación universal de la Web en muchos tipos de transacciones.

Entre las vulnerabilidades a nivel de aplicación, la clase de vulnerabilidades de validación de entrada es la más predominante y merece particular atención. Las vulnerabilidades de validación de entrada son defectos resultantes de suposiciones implícitas hechas por el desarrollador de la aplicación sobre la entrada de la aplicación. Más específicamente, las vulnerabilidades de validación de entrada existen cuando se puede aprovechar la invalidez de estas suposiciones usando entrada maliciosa para efectuar un cambio en el comportamiento de la aplicación que es beneficioso para el atacante.

Existen diferentes tipos de vulnerabilidades de validación de entrada, dependiendo de la suposicón inválida que se haya hecho. Las vulnerabilidades de *buffer overflow* resultan de suposiciones inválidas hechas sobre el tamaño máximo de la entrada. Los ataques de *integer overflow* resultan de suposiciones inválidas hechas sobre el rango de la entrada. De manera similar, las vulnerabilidades de inyección resultan de suposiciones inválidas hechas sobre la presencia de contenido sintáctico en la entrada de la aplicación. El contenido es considerado sintáctico cuando influye en la forma o la estructura de una expresión resultando en una alteración de la semántica de la misma. Este trabajo se enfoca en esta última clase de vulnerabilidades y en los ataques que las explotan. En estos ataques, llamados ataques de inyección, el atacante provee entrada maliciosa que incluye contenido sintáctico que cambia la semántica de una expresión en la aplicación. Los resultados dependen de la aplicación, pero típicamente conducen a un filtrado de información, escalado de privilegios o ejecución de comandos arbitrarios.

Las aplicaciones web son servicios brindados a los usuarios finales por un servidor web, típicamente a través de Internet y accedidos mediante un navegador web, por ejemplo, Gmail o Amazon.com. Éstas manejan grandes cantidades de datos que involucran diferentes usuarios web empleando diversas políticas de privacidad. El desarrollo de las mismas se hace típicamente olvidando las precauciones de privacidad. Esto se debe principalmente a la falta de conocimientos técnicos y herramientas apropiadas para garantizar la privacidad. Como resultado la información personal de los usuarios web está constantemente en riesgo.

En el contexto de las aplicaciones web, una "política de privacidad" es un acuerdo entre los usuarios y la organización que provee la aplicación web sobre como debería menejarse la información de los usuarios. Por ejemplo, los usuarios web ingresarían su número de tarjeta de crédito en un formulario

web si confían en que éste se mantendrá privado, o compartirían sus registros médicos para propósitos de investigación si se les garantizase privacidad.

Consideramos el problema de privacidad desde la perspectiva de ambas partes involucradas, por un lado los usuarios web y por el otro la persona responsable de garantizar la privacidad (llamado de aquí en adelante oficial de seguridad u OS para abreviar) junto con el grupo de desarrolladores web. Las aplicaciones web plantean un problema tanto para los usuarios web que deben confiar en el compromiso de privacidad de los dueños de la aplicación, como para los oficiales de seguridad que deben hacer cumplir las políticas de seguridad eficientemente con recursos limitados [Rezgui *et al.*, 2002]. En la mayoría de los casos la información personal de los usuarios web está débilmente protegida, a veces conscientemente y a veces por descuido. Los atacantes constantemente atentan contra las aplicaciones web accediéndolas como usuarios comunes y enviando entrada mal formada [OWASP Organization, 2003]. Un ataque exitoso puede exponer información privada, causar severos daños a la organización atacada y a los usuarios web expuestos [Zeller Jr., 2005].

La única protección posible para el tipo que hemos discutido antes es garantizar la privacidad de manera adecuada, forzando su existencia de alguna manera. Podemos describir el proceso de aplicación de privacidad en tres etapas: especificación, desarrollo y garantía de calidad. Primero, el OS define los requerimientos (idealmente durante la especificación de la aplicación web) y envía este documento a los desarrolladores para su implementación.

Los desarrolladores utilizan alguno de los tantos lenguajes para el diseño de una aplicación web, desde plataformas complejas y poderosas como Java o .NET hasta lenguajes de script fáciles de usar como PHP, Perl o ASP. Estos lenguajes o plataformas ayudan a los desarrolladores a producir rápidamente páginas generadas dinámicamente a través de facilidades de la programación de alto nivel que reducen la dificultad de acceder a funcionalidades complejas, como agentes de mail y back ends de base de datos. Sin embargo, estas facilidades carecen usualmente de seguridad y privacidad [OWASP Organization, 2003; Anupam and Mayer, 1998]; por ejemplo, no se puede establecer en forma simple que los datos recibidos a través de un cierto canal no deben ser expuestos a los usuarios.

Debido a que los lenguajes de script tiene un umbral de aprendizaje bajo, muchos proyectos son encomendados a desarrolladores web inexpertos, con escasos conocimientos de seguridad que no pueden implemenar las políticas de privacidad correctamente con las tecnologías disponibles. Los nuevos productos de seguridad no son efectivos debido a que apuntan a reducir los riesgos de seguridad y no a garantizar la privacidad. La única opción restante es que los desarrolladores implementen los requerimientos de privacidad [OWASP Organization, 2003] modificando sistemáticamente el código asociado con la entrada de los usuarios web en toda la aplicación, es decir, introduciendo chequeos de seguridad y comandos de políticas de validación.

Está de más decir que esta tarea requiere un gran consumo de tiempo y una habilidad técnica especial [Sandhu, 2003] y que su éxito depende de la cobertura del proceso: un error puede ser decisivo.

Después que el desarrollo ha terminado, el OS debe contratar a un equipo de auditores de seguridad para identificar y ayudar a eliminar las vulnerabilidades en la aplicación web, mejorando de esta manera la garantía global de privacidad. Una de las principales tareas de la sección Corelabs de la empresa Core Security Technologies, con la cual desarrollamos el proyecto mencionado en la sección 1.1, es la auditoria de código. Opcionalmente, el OS publica las políticas de privacidad utilizadas [1]. Una vez hecho esto, la aplicación web es publicada.

Por su parte, los usuarios web deben tomar precauciones de privacidad y seguridad cuando acceden a las aplicaciones web, es decir, revisar la configuración SSL, la configuración de cookies, chequear certificados [Herzberg and others, 2005] y leer las políticas de privacidad de la organización [Cranor *et al.*, 2002]. Los usuarios deben decidir si confían en que las políticas alegadas se cumplen antes de enviar información personal, o en caso contrario, salir del sitio.

En general, la evidencia muestra que la garantía de privacidad en las aplicaciones web sufren grandes problemas que no están siendo manejados; por ello es necesario desarrollar herremientas que permitan garantizar la privacidad y la seguridad.

2.2. Vulnerabilidades de Inyección

Las vulnerabilidades de inyección plantean una grave amenaza a nivel de seguridad de las aplicaciones. Éstas son fallas de programación que permiten a un atacante alterar la semántica de una expresión en una aplicación, proveyendo entrada que incluye contenido sintáctico. Algunos de los tipos más comunes son las vulnerabilidades SQL injection, cross-site scripting y shell injection, que se desciben a continuación.

2.2.1. Vulnerabilidades SQL injection

Las vulnerabilidades SQL injection ocurren cuando valores no confiables se usan para construir comandos SQL, resultando en la ejecución de comandos SQL arbitrarios dados por un atacante.

El siguiente código muestra un ejemplo real de una parte de una aplicación PHP, responsable de la autenticación a través de una dirección de e-mail (`$email`) y un código de pin numérico (`$pincode`) contra credenciales almacenadas en una base de datos. El usuario es autenticado exitosamente si el

[1] Los usuarios web deberían poder informarse de las políticas de seguridad y estar garantizados de que las mismas están actualizadas y se cumplen.

resultado es un conjunto no vacío.

```
$query = "SELECT *
          FROM users
          WHERE email='"  . $email . "' AND pincode=" . $pincode;
$result = mysql_query($query);
```

Este código es propenso a varios ataques SQL injection. Si el atacante provee `"alice@host' OR '0'='1"` (notar el uso de las comillas simples "desbalanceadas") como dirección de e-mail, la aplicación ejecuta una consulta cuyo resultado es independiente del código de pin provisto, pues el string de entrada enviado a la función `mysql_query` será

```
"SELECT *
FROM users
WHERE email='alice@host' OR '0'='1' AND pincode="
```

Debido a la precedencia de operadores, en este caso OR sobre AND, esta consulta será equivalente a una con una sola condición `"email='alice@host'"`, permitiendo de esta manera al atacante omitir la lógica de autenticación. Ataques similares realizados usando la variable pincode, la cual es usada en un contexto numérico, no requieren comillas simples en la entrada del usuario. Por ejemplo, usando la dirección de e-mail válida `"alice@host"` y `"0 OR 1=1"` como código de pin, el atacante podría nuevamente autenticarse sin las credenciales apropiadas.

2.2.2. Vulnerabilidades shell injection

Las vulnerabilidades shell injection ocurren cuando se usan datos no confiables para invocar funciones que manipulan los recursos del sistema, por ejemplo en PHP: `fopen()`, `rename()`, `copy()`, `unlink()`, etc. o para invocar procesos, por ejemplo, `exec()`.

Un ejemplo para demostrar una vulnerabilidad shell injection se vé en el siguiente código que envía un e-mail de confirmación a una dirección de e-mail provista.

```
$handle = popen("/usr/bin/mail $email");
fputs($handle); #escribe el mensaje
```

En este caso, cualquiera de los metacaracteres (`, `&&`, `;`, `newline`) en el campo de la dirección de e-mail puede ser usado para ejecutar comandos arbitrarios sobre el servidor. Por ejemplo, si el atacante usa como dirección de e-mail `"alice@host && rm -rf ."`, el servidor web, podría tratar de eliminar todos los archivos del directorio actual, además de enviar el e-mail.

2.2.3. Vulnerabilidades cross-site-scripting

Las vulnerabilidades cross-site-scripting, a veces llamadas XSS, ocurren cuando un atacante usa una aplicación web para enviar código malicioso, generalmente JavaScript, a otro usuario final. Cuando la aplicación web usa sin filtrar la entrada de un usuario en la salida que genera, un atacante puede insertar un ataque en esa entrada y la aplicación web envía el ataque a otros usuarios. El usuario final confía en la aplicación web y el ataque tiene éxito haciendo cosas que normalmente no estarían permitidas. Los atacantes frecuentemente usan una gran variedad de métodos para codificar la porción maliciosa de la tarea, como usar Unicode[2], por lo que el requerimiento resulta menos sospechoso para el usuario.

Los ataques XSS pueden ser clasificados en dos categorías: permanentes (*stored*) y no permanentes (*reflected*). Los ataques no permanentes son aquellos en los que el código inyectado es retornado por el servidor web, en un menseje de error, en un resultado de búsqueda o en cualquier otra respuesta que incluya alguna o toda la entrada enviada al servidor como parte del requerimiento. Es decir, el código inyectado es embebido en la página retornada al navegador del usuario inmediatamente después del requerimiento. Estos ataques son enviados a la víctima a través de otra ruta, como un mensaje de e-mail o sobre algún otro servidor. Cuando un usuario clickea un link o envía un formulario, el código inyectado viaja al servidor web vulnerable el cual envía el ataque de vuelta al navegador del usuario. Luego el navegador ejecuta el código debido a que proviene de un servidor "confiable" y se produce el ataque. Los ataques permanentes son aquellos donde el código inyectado se almacena permanentemente en el servidor, en una base de datos, en un foro de mensajes, etc. En este caso el ataque no se producirá inmediatamente, sino algún tiempo después, por ejemplo, en el momento que el usuario lea la entrada infectada del foro.

Las consecuencias de un ataque XSS son las mismas independientemente si es permanente o no. Los ataques XSS pueden causar una gran variedad de problemas al usuario final; los más severos incluyen la revelación de la cookie de sesión del usuario, permitiendo a un atacante robar la sesión del usuario y apoderarse de su cuenta. Otros ataques incluyen la revelación de los archivos del usuario final, la instalación de programas troyanos, la redirección del usuario a otra página y la modificación de la presentación del contenido. Por ejemplo, una vulnerabilidad XSS en un sitio farmacéutico podría permitir a un atacante modificar la información de dosificación produciendo una dosis excesiva.

El siguiente código muestra un ejemplo de una vulnerabilidad cross-site

[2]Unicode es un estándar industrial cuyo objetivo es proporcionar el medio por el cual un texto en cualquier forma e idioma pueda ser codificado para el uso informático. Unicode proporciona un número único para cada carácter, sin importar la plataforma, sin importar el programa y sin importar el idioma. Ejemplos: UTF-8, UCS-2

scripting:

```
$month = $_GET['month'];
$year = $_GET['year'];
$day = $_GET['day'];
echo "<a href=\"day.php?year=$year&amp;"
echo "month=$month&amp;day=$day\">";
```

Los valores para las variables `$month`, `$day` y `$year` provienen de requerimientos HTTP y se usan para contruir salida HTTP para el usuario. Suponiendo que este código se encuentra en la página `event_delete.php` del servidor `http://www.target.com`, un ejemplo de ataque podría ser que el usuario la acceda tipeando en el navegador:

```
http://www.target.com/event_delete.php?year=
><script>script_malicioso();</script>
```

lo cual equivale a enviar `"><script>script_malicioso();</script>"` en el parámetro year (notar el uso del tag de cierre al principio del valor del parámetro).

Los atacantes deben encontrar la forma de que las víctimas abran esta URL. Una estrategia es enviar un e-mail conteniendo código JavaScript que secretamente abra una ventana oculta que abra esta URL. Otra opción es embeber ese mismo código JavaScript dentro de una página web: cuando las víctimas abran la página, el script se ejecuta y secretamente abre la URL. Una vez que el código mostrado anteriormente recibe un requerimiento HTTP para la URL, se genera la siguiente salida HTML:

```
<a href="day.php?year=><script>script_malicioso();</script>
```

En esta estrategia, la salida contiene un script malicioso preparado por un atacante y enviado en nombre del servidor web. Por lo tanto, se pierde la integridad de la salida HTML y se violan las políticas de los navegadores sobre mismo origen de fuente[3]. Debido a que el script malicioso es enviado en nombre del servidor web, este garantiza el mismo nivel de confianza que el servidor web, el cual mínimamente permite que el script lea cookies de usuarios setadas por el servidor. Esto frecuentemente revela claves de acceso o permite el robo de sesiones; si el servidor web está registrado en el dominio confiable del navegador de la víctima, puede que otros derechos sean cedidos, como por ejemplo acceso al sistema de directorios local.

Los problemas XSS también pueden presentarse en los servidores web o de aplicaciones. La mayoría de los servidores generan páginas web simples

[3]Las políticas sobre el mismo origen de fuente establecen que en un sitio web sólo deberían accederse recursos del mismo sitio ya que los recursos de otro sitio web podrían ser maliciosos. El término "origen" se define usando el nombre de dominio, el protocolo y el puerto. Dos páginas pertenecen al mismo origen si y sólo si estos tres valores coinciden.

para mostrar en el caso de varios errores, como 404 “página no encontrada” o 500 “error interno del servidor”. Si estas páginas reflejan alguna información del requerimiento del usuario, como la URL que están intentando acceder, podrían ser vulnerables a un ataque XSS no permanente. Por ejemplo, si el usuario ingersa la URL `http://www.example.com/FILENAME.html` y la página `"FILENAME.html"` no existe, el servidor del sitio web podría retornar un mensaje de error de la forma:

```
<HTML> 404 pagina no encontrada: FILENAME.html .... </HTML>
```

Notar que `"FILENAME.html"` es un string ingresado por el usuario y que el sito web lo ha incluído en la página retornada al navegador. Si en lugar del nombre de la página un atacante ingresa un tag HTML o un script como `http://www.example.com/<script>codigo_malicioso();</script>` el servidor retornará la siguiente página de error:

```
<HTML> 404 pagina no encontrada:
<script>codigo_malicioso();</script> .... </HTML>
```

la cual ejecutará el código ingresado en el script por el atacante.

2.2.4. Análisis de las causas de las vulnerabilidades de inyección

En todos nuestros ejemplos, la entrada maliciosa incluye contenido sintáctico. El contenido es considerado sintáctico cuando influye en la forma o la estructura de una expresión. Este cambio de estructura resulta en una alteración de la semántica de la expresión. Los caracteres clasificados como contenido sintáctico dependen del contexto en el cual se usa la expresión (comandos SQL o comandos shell). Además, el contexto también depende de cómo se usa la entrada dentro de la expresión (constantes strings vs. códigos de pin númericos en una sentencia SQL en nuestro primer ejemplo). Identificar todo el contenido sintáctico para los diferentes contextos es por lo tanto el mayor desafío.

La eliminación de comillas simples y espacios de la entrada podría prevenir algunos de los ataques descriptos, pero seguramente no podría evitar todos los ataques. Otros caracteres peligrosos incluyen secuencias de comentarios (–, /*, */) y puntos y comas (;), pero tampoco es una lista exhaustiva [Maor and Shulman, 2004].

Además, los servidores de base de datos comúnmente extienden el estándar ANSI SQL con características propietarias y ayudan a corregir errores sintácticos (por ejemplo, permiten el uso de comillas dobles (”) en lugar de simples (’) para delimitar constantes strings). Debido a que los chequeos necesarios son específicos de cada base de datos, una aplicación se puede volver vulnerable por un mero cambio de la misma.

Las vulnerabilidades de inyección son comúnmente clasificadas como vulnerabilidades de validación de entrada. Sin embargo, la tarea de validar la entrada del usuario para prevenir estos ataques no es trivial y es propensa a errores. Por lo tanto, tratar a estas vulnerabilidades como vulnerabilidades de validación de entrada es una sobresimplificación.

Una propiedad común de las vulnerabilidades de inyección es el uso de representación textual en las expresiones de salida construídas usando la entrada provista por el usuario. Las representaciones textuales son representaciones en una forma de texto legible por los humanos. Las expresiones de salida son expresiones manejadas por un componente externo, como un servidor de base de datos o un intérprete de comandos.

La entrada del usuario es típicamente usada en las partes de datos de las expresiones de salida. Por lo tanto, la entrada del usuario no debería incluir contenido sintáctico. En un ataque de inyección, la entrada del usuario influencia la sintaxis produciendo un cambio en la semántica de la expresión de salida. Consecuentemente, para que una vulnerabilidad de inyección esté presente en una aplicación se necesitan dos requisitos. El primero es que la aplicación tiene que usar una expresión de salida creada usando serialización ad-hoc de variables. La segunda es que la expresion de salida dependa de los datos de entrada provistos por el usuario, para que pueda ser influenciada por el atacante.

2.2.5. Soluciones posibles a las vulnerabilidades de inyección

Existen dos soluciones posibles al problema visto, el de las vulnerabilidades de inyección, que dependen del momento en el que se requiera:

- Si aún no se ha realizado la codificación, se pueden estudiar técnicas de construcción de programas que garanticen la ausencia de estas vulnerabilidades.

- En caso contrario, se debe realizar un análisis del código posterior a la codificacíon para detectar dichas vulnerabilidades con el fin de eliminarlas del mismo.

En este trabajo nos concentramos en la segunda alternativa dado que como se mencionó anteriormente, una de las principales tareas de la sección Corelabs de la empresa Core Security Technologies es la auditoria de código y el objetivo del proyecto realizado con dicha empresa fué la automatización de esta tarea utilizando alguna técnica de análisis estático.

En el capítulo siguiente presentamos las técnicas de análisis estático de manera general para luego especializarlas.

Capítulo 3

Sistemas de Tipos y GBA

En este capítulo presentaremos la técnica llamada Grammar-Based Analysis (GBA). GBA es una instancia de un framework general para sistemas de tipos estilo Hindley-Milner llamado HM(X). Debido a que muchas de las ideas de GBA fueron tomadas de dicho framework, comenzaremos describiendo las principales características del sistema de tipos Hindley-Milner y del framework HM(X).

El material presentado sobre GBA está tomado del paper "Grammar-Based Analysis of String Expressions" [Thiemann, 2005]. Incluimos gran parte de los conceptos mencionados en dicho paper para una mejor comprensión de nuestro trabajo.

3.1. Sistemas de Tipos

El sistema Hindley-Milner [Hindley, 1969; Milner, 1978; Damas and Milner, 1982] es un sistema de tipos adoptado como base por la mayoría de los lenguajes funcionales tipados estáticamente, como por ejemplo Hope [Burstall *et al.*, 1980], Standard ML [Milner *et al.*, 1990], Miranda [Turner, 1986] y Haskell [Hudak *et al.*, 1992]. Entre las principales razones se encuentra su flexibilidad, permitiendo que un único valor sea usado de diferentes formas, y su practicidad, ya que libera al programador de la necesidad de suministrar información de tipo explícitamente.

El sistema Hindley-Milner es un sistema polimórfico en el sentido que permite que a una expresión que puede ser tipada de infinitas maneras, se le asigne un tipo que sea más general que todos ellos, y tal que en cada uso pueda transformarse en uno particular. A esta clase de polimorfismo se lo conoce como polimorfismo paramétrico. A modo de ejemplo podemos considerar la función polimórfica $\mathsf{id} :: a \to a$ donde el tipo de su argumento puede ser instanciado de diferentes maneras en diferentes usos:

$(\mathsf{id}\ 3) :: Int$ y aquí $\mathsf{id} :: Int \to Int$

(id True) :: $Bool$ y aquí id :: $Bool \to Bool$

Otra característica importante del sistema polimórfico de Hindley-Milner es que existe para el mismo un algoritmo de inferencia automática que puede ser usado para determinar que una expresión dada está bien tipada y además para calcular su tipo más general sin requerir niguna información de tipo en el código del programa.

El sistema utiliza algún conjunto de tipos básicos (usualmente Int, $Char$, $Bool$, etc.) y además posee tipos compuestos entre los cuales los más destacados son las funciones ($A \to B$). El sistema cuenta con 4 reglas fundamentales; las mismas pueden observarse a continuación:

$$\frac{\Gamma, x :: \tau_2 \vdash e :: \tau_1}{\Gamma \vdash \lambda x.e :: \tau_2 \to \tau_1}\ Funciones$$

$$\frac{\Gamma \vdash e :: \tau_2 \to \tau_1 \qquad \Gamma \vdash e' :: \tau_2}{\Gamma \vdash ee' :: \tau_1}\ App$$

$$\frac{\Gamma \vdash e :: \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e :: \forall \alpha.\tau}\ Gen$$

$$\frac{\Gamma \vdash e :: \forall \alpha.\tau}{\Gamma \vdash e :: \tau[\alpha := \tau']}\ Inst$$

Existen diversas extensiones del sistema de Hindler-Milner; por ejemplo sistemas para tipar registros, overloading (polimorfismo ad-hoc) y subtyping. Todas ellas son presentadas mediante el uso de restricciones o constraints y desarrollan teorías de restricciones y algoritmos de inferencia similares. En este tipo de sistemas hay dos fases principales. La primera es la inferencia de tipos que calcula un conjunto de restricciones y la segunda es la resolución de dichas restricciones para alcanzar una solución. Los juicios de tipo $\Gamma \vdash e :: \sigma$ del sistema Hindley-Milner original, se extienden con una hipótesis restringida del lado izquierdo escrita

$$C, \Gamma \vdash e :: \sigma$$

Además se extienden los esquemas de tipo

$$\forall \alpha.\tau$$

del sistema Hindley-Milner con un componente de restricción:

$$\forall \alpha.C \Rightarrow \tau$$

para expresar que el constraint C restringe los tipos que pueden ser legalmente sustituídos por las variables ligadas α.

Aunque estos sistemas de tipos usan diferentes dominios de constraints, son semejantes en sus aspectos teóricos de tipos. Existe un framework general llamado HM(X) para sistemas de tipos estilo Hindley-Milner con restricciones [Odersky *et al.*, 1999]. Diversos sistemas de tipos pueden ser obtenidos instanciando el parámetro X a un sistema de restricciones específico. El framework HM(X) es presentado junto con un algoritmo de inferencia de tipos genérico; bajo condiciones suficientes sobre el dominio de constraints X, la inferencia de tipos siempre computará el tipo principal de un término.

Por mucho tiempo, el tratamiento de retricciones en sistemas de tipos ha sido sintáctico: las restricciones eran consideradas como conjuntos de fórmulas, frecuentemente de una forma específica. Sin embargo, la programación con restricciones actual usa generalmente una definición semántica de los sistemas de restricciones, considerando un sistema de restricciones como un álgebra cilíndrica con alguna propiedades adicionales. Las álgebras cilíndricas definen un operador de proyección $\exists\widetilde{\alpha}$ que liga algún subconjunto de variables de $\widetilde{\alpha}$ en el constraint. En el caso usual donde los constraints son álgebras booleanas, la proyección corresponde al cuantificador existencial. Siguiendo la dirección de la programación de constraints, HM(X) considera que X es un álgebra cilíndrica con un operador de proyección. La proyección es considerada útil para este propósito por dos razones: la primera, la proyección nos permite formular una regla lógicamente favorable y pragmáticamente útil (∀ Intro).

$$\frac{C \wedge D, \Gamma \vdash e :: \tau \quad \widetilde{\alpha} \notin FV(C) \cup FV(\Gamma)}{C \wedge \exists\widetilde{\alpha} D, \Gamma \vdash e :: \forall\widetilde{\alpha}.D \Rightarrow \tau}\, \forall Intro$$

donde C y D son constraints sobre las variables de tipos en el contexto de tipo Γ y el esquema de tipo σ. Segundo, la proyección es una fuente importante de oportunidades para simplificación de constraints. En HM(X), simplificar significa cambiar la representación sintáctica de un constraint sin cambiar su denotación. Por ejemplo, el constraint de subtipo $\exists\beta.(\alpha \leq \beta) \wedge (\beta \leq \gamma)$ se puede simplificar a $(\alpha \leq \gamma)$ debido a que la denotación es la misma para ambos constraints. Sin el operador de proyección, los dos constraints serían diferentes debido a que uno limita la variable β y el otro no.

En el sistema de tipos de HM(X) se admite solamente un subconjunto S de constraints de X, denominados constraints en *solved form*. Los dominios sintácticos básicos para un sistema de constraints X son:

Valores	v	::=	$x \mid \lambda x.e$
Expresiones	e	::=	$v \mid ee \mid \text{let } e = x \text{ in } e$
Tipos	τ	::=	$\alpha \mid \tau \to \tau$
Esquemas de tipos	σ	::=	$\tau \mid \forall\alpha.C \Rightarrow \sigma$

Los esquemas de tipo $\forall\alpha.C \Rightarrow \sigma$ ahora incluyen un componente de constraint $C \in S$, el cual restringe los tipos que pueden ser sustituídos por la variable de tipo α. El lenguaje de términos es el mismo que en [DM82].

La inferencia de tipos en HM(X) se realiza en dos pasos. Primero, un problema de tipado es traducido en un constraint D en el sistema de constraints X. Luego el constraint D es normalizado. Normalizar significa computar una sustitución ψ y un constraint residual C en X tal que ψC implica a D donde ψC es un constraint en el conjunto de constraints en solved form.

3.2. Grammar-Based Analysis (GBA)

Como se mencionó en el Capítulo 2, los valores strings han perdido su inocencia y están siendo usados en muchos contextos imprevistos. En todos los casos los strings deben respetar un formato externo fijo (strings XHTML, sentencias SQL, etc.)

Juzgando por la cantidad de errores de este tipo encontrados, estamos enfrentando un serio problema de mantenimiento de software. ¿Cómo vamos a estar seguros de que todos estos strings construídos sin una metodología apropiada respetan un formato deseado?

Una buena respuesta a esta pregunta es la utilización de un análisis de programas y en particular de análisis de expresiones string. El análisis de expresiones string ha sido considerado anteriormente por dos grupos de investigadores [Tabuchi *et al.*, 2003; Christensen *et al.*, 2003]. Ambos modelaron los valores posibles de una expresión string con un lenguaje regular y confiaron en la decibilidad de la inclusión para lenguajes regulares. Los lenguajes regulares producen resultados altamente útiles pero no pueden garantizar que un valor string es un elemento de un lenguaje libre de contexto. Por ejemplo, los strings XHTML, las sentencias SQL y las expresiones XPath y JavaScript son todos definibles por gramáticas libres de contexto y no pueden ser descriptos por lenguajes regulares de manera completamente satisfactoria.

Sin embargo, los enfoques anteriores no generalizan a los lenguajes libres de contexto. Aunque una gramática libre de contexto puede ser construída para que describa los valores de todas las expresiones string [Christensen *et al.*, 2003], no es posible testear el lenguaje de esa gramática contra un lenguaje de referencia libre de contexto describiendo los valores deseados porque la inclusión de lenguaje es indecidible para lenguajes libres de contexto [Hopcroft, 1969]

El presente capítulo describe un novedoso framework basado en tipos para el análisis de expresiones string que garantiza que todos los valores de una expresión string son elementos de un lenguaje libre de contexto dado por alguna gramática de referencia G. El framework es correcto pero incompleto porque reemplaza la propiedad de inclusión (indecidible) por la propiedad de derivación (decidible).

El análisis de expresiones string está expresado como un sistema de tipos restringido, donde las restricciones o constraints son parseados con respec-

to a una gramática de referencia G. Intuitivamente, un constraint parseado expresa la relación de derivación de G la cual es definida en términos de símbolos terminales y no-terminales de G. Como la derivación es decidible para gramáticas libres de contexto, los constraints pueden ser resueltos efectivamente. En particular, se exhibe una generalización del algoritmo de parseo para gramáticas libres de contexto de Earley [Earley, 1970] para resolver los constraints. Gracias a la formulación basada en tipos, el análisis es polimórfico y se aplica a un lenguaje polimórfico.

Hay dos advertencias sobre este framework. Primero, mucho depende de la selección apropiada de la gramática para asegurar que el parser de Earley puede resolver todos los constraints. En particular, las gramáticas que sean recursivas a derecha o a izquierda no parsearán los constraints exitosamente si el programa construye los strings de la otra manera. Las posibilidades de un parseo exitoso son mucho más altas con una gramática ambigua que no prefiera ninguna dirección particular de construir listas.

Segundo, debido a que el análisis trabaja directamente a nivel de caracteres, la gramática debe estar dada al mismo nivel. Una gramática que asume un análisis léxico precedente y usa tokens como símbolos terminales no funcionará. En cambio, la gramática debe especificar la sintaxis bajo el nivel caracter, similar a la gramática para scannerless parsing.

3.2.1. Análisis de expresiones string polimórficas

La tarea del análisis de strings es determinar para cada ocurrencia de un tipo string en una derivación de tipo, un conjunto de strings tal que cada valor computado por una expresión de ese tipo es miembro de ese conjunto. Esta sección desarrolla una especificación de referencia para un análisis de strings que escencialmente infiere una gramática libre de contexto de un programa dado tal que el valor del string en el programa puede ser derivado de la gramática. El análisis es expresado como un sistema de tipos donde el tipo string es refinado con una estructura indexada apropiada.

El lenguaje considerado es un lambda cálculo aplicado con funciones recursivas, un condicional que chequea si un string está vacío y operaciones para construcciones de strings (constantes strings y concatenación de strings). Los strings son secuencias de símbolos terminales pertenecientes a un conjunto finito T llamado alfabeto, con ϵ denotando al string vacío. La Figura 3.1 define la sintaxis y la semántica del cálculo. La semántica está dada en un estilo operacional con contextos de evaluación [Wright and Felleisen, 1994]. Los contextos no fijan el orden de evaluación completamente: admiten reducción por valor y reducción por nombre. Como es usual, $\xrightarrow{*}$ denota la clausura reflexivo transitiva de la relación de reducción $\rightarrow$. Todas las operaciones entre strings (condicional, constantes strings y concatenación de strings) están incluídas en el cálculo a través de constantes con reducciones delta.

Alfabeto	T		
Símbolos	a, b	$\in$	T
Palabras	ω	$\in$	T^*
Constantes	c	$\in$	$\{\omega \in T^*\} \cup \{\cdot, \mathsf{if}\}$
Expresiones	e	$::=$	$c \mid x \mid e\ (e) \mid \mathsf{rec}\ f(x)\ e \mid$ $\mathsf{let}\ x = e\ \mathsf{in}\ e$
Valores	υ	$::=$	$\mathsf{rec}\ f(x)\ e \mid c$
Contextos de evaluación	E	$::=$	$[\,] \mid E(e) \mid e(E)$

Beta reducción

$$(\mathsf{rec}\ f(x)\ e)\ (\upsilon) \quad \longrightarrow \quad e\,[x \mapsto \upsilon, f \mapsto \mathsf{rec}\ f(x)\ e]$$

Delta reducción

$$\begin{array}{lcl} \mathsf{if}(ab_1...bm)\ e_1\ e_2 & \longrightarrow & e_1 \\ \mathsf{if}\ \epsilon\ e_1\ e_2 & \longrightarrow & e_2 \\ a_1...a_n \cdot b_1...b_m & \longrightarrow & a_1...a_n b_1...b_m \end{array}$$

Reducción

$$\frac{e \longrightarrow e'}{E\,[e] \mapsto E\left[e'\right]}$$

Figura 3.1: Sintaxis y semánticas dinámicas

Tipos	τ	::=	$\alpha \mid Str(\varphi) \mid \tau \to \tau$
Variables de lenguaje	φ	$\in$	Φ
Tipos restringidos	ρ	::=	$C \Rightarrow \tau$
Constraints	C	::=	$true \mid \tau \dot{\leq} \tau' \mid r \dot{\subseteq} \varphi \mid C \wedge C$
Índices del tipo string	r	::=	$\varphi \mid \epsilon \mid a \mid r \cdot r$
Esquemas de tipos	σ	::=	$\forall \widetilde{\varphi} \widetilde{\alpha}.\rho$
Ambientes de tipos	Γ	::=	$\emptyset \mid \Gamma(x : \sigma)$

Figura 3.2: Lenguaje de tipos

$$L_\Theta(\varphi) = \Theta(\varphi) \qquad L_\Theta(\epsilon) = \{\epsilon\} \qquad L_\Theta(a) = \{a\}$$
$$L_\Theta(r_1 \cdot r_2) = \{\omega_1 \cdot \omega_2 \mid \omega_1 \in L_\Theta(r_1), \omega_2 \in L_\Theta(r_2)\}$$

Figura 3.3: Significado de los índices del tipo string

La Figura 3.2 define la sintaxis de tipos. Un tipo puede ser una variable de tipo, un tipo string o un tipo función. Cada tipo string está indexado por una *variable de lenguaje* φ. Una variable de lenguaje representa un lenguaje sobre T. Los tipos pueden estar restringidos por una conjunción de constraints de subtipos $\tau \dot{\leq} \tau'$ y constraints de inclusión $r \dot{\subseteq} \varphi$ donde r es una secuencia de símbolos terminales y variables de lenguaje y la operación $(\cdot)$ es considerada asociativa. Los tipos restringidos pueden ser abstraídos sobre una secuencia de variables de tipo y variables de lenguaje (usando la notación $\widetilde{\varphi}$ para una secuencia φ_1 ... φ_n de longitud n). La función *fv* es aplicable a todos los tipos de constraints y expresiones de tipo para las cuales obtiene el conjunto de variables de tipo y variables de lenguaje libres, con la definición estándar.

El álgebra de tipos es many-sorted. Además de la clase usual de tipos hay otra clase de índices string. En esta visión, una variable de lenguaje es sólo una variable de tipo con un sort diferente; sin embargo usaremos la denominacion "tipo" para el sort original.

Hay una relación de subtipos que es estructural y completamente determinada por la inclusión de constraints sobre los índices del tipo string. Debido a que los índices del tipo string (y por lo tanto los tipos, los tipos restringidos, constraints y esquemas de tipos) pueden contener variables de lenguaje, el significado de un tipo expresado es relativo a una asignación $\Theta : \Phi \to \mathcal{P}(T^*)$ que mapea variables de lenguaje a un conjunto de strings. En particular, la Figura 3.3 define que $L_\Theta(r)$ es el conjunto de strings denotado por r bajo la asignación Θ.

Un modelo para un constraint C (denotado $S, \Theta \models C$) es un par (S, Θ) donde S es una sustitución y Θ una asignación sujeta a las siguientes condiciones:

$$S, \Theta \models true$$

$$S, \Theta \models \alpha \dot{\leq} \alpha$$

$$\frac{\Theta(\varphi_1) \subseteq \Theta(\varphi_2)}{S, \Theta \models Str(\varphi_1) \dot{\leq} Str(\varphi_2)}$$

$$\frac{S, \Theta \models \tau_2' \dot{\leq} \tau_2 \quad S, \Theta \models \tau_1 \dot{\leq} \tau_1'}{S, \Theta \models \tau_2 \to \tau_1 \dot{\leq} \tau_2' \to \tau_1'}$$

$$\frac{L_\Theta(r) \subseteq \Theta(\varphi)}{S, \Theta \models r \dot{\subseteq} \varphi}$$

$$\frac{S, \Theta \models C_1 \quad S, \Theta \models C_2}{S, \Theta \models C_1 \wedge C_2}$$

$$\frac{S, \Theta \models S(C)}{S, \Theta \models C}$$

Figura 3.4: Satisfacción de Constraints, parte 1

1. La sustitución S mapea variables de tipo a tipos y variables de lenguaje a variables de lenguaje. La sustitución S debe ser idempotente.

2. La asignación Θ mapea variables de lenguaje a conjuntos de strings.

3. Θ debe estar definida sobre todas las variables de lenguaje en C y $S(C)$ y debe ser compatible con S; o sea para todo φ, $\Theta(\varphi) = \Theta(S(\varphi))$.

4. $S, \Theta \models C$ debe ser derivable usando las reglas de inferencia de la Figura 3.4. La última regla (aplicación de sustitución) necesita usarse solamente a lo sumo una vez en una derivación porque las sustituciones son idempotentes.

Definimos constraint entailment $C \Vdash C'$ de la forma usual: $C \Vdash C'$ si y sólo si $(S, \Theta) \models C$ implica $(S, \Theta) \models C'$.

Las reglas en la Figura 3.5 definen el sistema de deducción para el juicio de tipado $C, \Gamma \vdash e : \sigma$ que relaciona un constraint, un ambiente de tipo y una expresión con un esquema de tipos. Intuitivamente, una expresión de tipo $Str(\varphi)$ bajo un constraint C sólo puede asumir valores strings de $\Theta(S(\varphi))$ para algún S, Θ donde $S, \Theta \models C$. Las reglas para el lambda cálculo y la regla de sustitución se toman de la presentación lógica del sistema HM(X) [Odersky *et al.*, 1999]. La única diferencia es la introducción de la regla para polimorfismo. Las reglas HM(X) exponen que el constraint abstraído D debe ser soluble requiriendo $\exists \widetilde{\alpha}\widetilde{\varphi}.D$. La omisión de este constraint efectivamente difiere el chequeo de solubilidad al punto de uso. Todas las operaciones para

$$\frac{(x:\sigma)\in\Gamma}{C,\Gamma\vdash x:\sigma} \qquad \frac{C,\Gamma\vdash e:\tau \quad C\Vdash\tau\dot{\leq}\tau'}{C,\Gamma\vdash e:\tau'}$$

$$\frac{C,\Gamma\vdash e_1:\tau_2\to\tau_1 \quad C,\Gamma\vdash e_2:\tau_2}{C,\Gamma\vdash e_1\ (e_2):\tau_1}$$

$$\frac{C,\Gamma(f:\tau_2\to\tau_1)(x:\tau_2)\vdash e:\tau_1}{C,\Gamma\vdash \mathsf{rec}\ f(x)\ e:\tau_2\to\tau_1}$$

$$\frac{C,\Gamma\vdash e:\sigma \quad C,\Gamma(x:\sigma)\vdash e':\tau'}{C,\Gamma\vdash \mathsf{let}\ x=e\ \mathsf{in}\ e':\tau'}$$

$$\frac{C\wedge D,\Gamma\vdash e:\tau \quad \{\widetilde{\alpha},\widetilde{\varphi}\}\cap(fv(C)\cup fv(\Gamma))=\emptyset}{C,\Gamma\vdash e:\forall\widetilde{\alpha},\widetilde{\varphi}.D\Rightarrow\tau}$$

$$\frac{\begin{array}{c}C,\Gamma\vdash e:\forall\widetilde{\alpha},\widetilde{\varphi}.D\Rightarrow\tau\\ S=\left[\widetilde{\alpha}\mapsto\widetilde{\tau},\widetilde{\varphi}\mapsto\widetilde{\varphi'}\right] \quad C\Vdash S(D)\end{array}}{C,\Gamma\vdash e:S(\tau)}$$

Figura 3.5: Reglas de tipos

strings (constantes ω, concatenación de strings $\cdot$ y el condicional if) se hacen disponibles a través de un ambiente de tipos inicial que define

$$\begin{array}{l}\omega:\forall\varphi.(\omega\ \dot{\subseteq}\ \varphi)\Rightarrow Str(\varphi)\\ \cdot\ :\forall\varphi_1,\varphi_2,\varphi.(\varphi_1\cdot\varphi_2\ \dot{\subseteq}\ \varphi)\Rightarrow Str(\varphi_1)\to Str(\varphi_2)\to Str(\varphi)\\ \mathsf{if}\ :\forall\alpha,\varphi.Str(\varphi)\to\alpha\to\alpha\to\alpha\end{array}$$

Observar que la concatenación restringe (mediante la condición $(\varphi_1\cdot\varphi_2\ \dot{\subseteq}\ \varphi)$) el lenguaje del string resultante $Str(\varphi)$ para ser la concatenación de los lenguajes de los strings de entrada $Str(\varphi_1)$ y $Str(\varphi_2)$.

Los subtipos son un ingrediente necesario del sistema. Se requiere aproximar las ramas de los condicionales, para definir funciones recursivas y para usar funciones en más de un contexto.

Nuestro uso de álgebra de tipos many-sorted es sólo una mera extensión de HM(X). Todos los resultados para HM(X) trasladados al caso many-sorted han sido observados y expresados por los autores de HM(X) [Odersky *et al.*, 1999].

Es un ejercicio estándar probar que este sistema es correcto, probando preservación de tipos y progreso.

Teorema 1 *Supongamos que* $C,\emptyset\vdash e:\tau$ *y que existen* S,Θ *tales que* $S,\Theta\models C$. *Luego, e diverge o hay algún* v *tal que* $e\xrightarrow{*}v$ *y* $C,\emptyset\vdash v:\tau$.

Una parte de la propiedad de progreso es el establecimiento de un lema en forma canónica. Aunque este lema es usualmente sólo de interés técnico,

aquí se muestra porque expresa la conexión entre los constraints, los tipos strings indexados y los valores resultantes.

Lema 1 *Supongamos que* $C, \Theta \vdash v : \tau$.

1. *Si* $\tau_1 = \tau_2$, *luego* $v = \mathsf{rec}\ f(x)\ e$, *para algún* f, x *y* e.
2. *Si* $\tau = Str(\varphi)$, *luego* $v = \omega$, *para algún* $\omega \in T^*$ *y* $\omega \in \Theta(S(\varphi))$, *para todo* S, Θ, *tales que* $S, \Theta \models C$.

Una vez especificado el sistema de tipos, en la Sección 3.2.2 se presenta una versión algorítmica del mismo. En esa versión se intenta reducir los constraints de subtipo tanto como sea posible, pero difiere todos los constraints de inclusión $r \dot{\subseteq} \varphi$. Escencialmente, extrae los constraints que corresponden a una gramática libre de contexto para describir todos los conjuntos de strings que ocurren en un tipo. La secciones siguientes investigan futuras reglas de simplificación para resolver los constraints de inclusión.

3.2.2. Algoritmo de análisis de expresiones

Para llegar a una gramática que describa los conjuntos de strings en los tipos, se transforma la presentación lógica del sistema de tipos de la sección previa en una formulación algorítmica guiada por sintaxis. Esta transformación es exactamente lo que prescribe HM(X) [Odersky *et al.*, 1999] y su resultado equivale a un algoritmo de reconstrucción de tipos para el análisis polimórfico de strings.

La formulación del sistema algorítmico requiere un sistema de constraints para expresiones. Por lo tanto, los constraints C reciben tres nuevas alternativas: dos para sustituciones y una para expresar falla, cada una de las cuales surgen durante la simplificación de constraints. La lista completa de alternativas para un constraint C es:

- $\alpha \doteq \tau$ (sustitución para variables de tipo)
- $\varphi \doteq \varphi'$ (sustitución para variables de lenguaje)
- $\tau \dot{\leq} \tau'$ (subtipo)
- $r \dot{\subseteq} \varphi$ (inclusión del lenguaje, $r \in (\Phi \cup T)^*$)
- $C' \wedge C''$ (conjunción de constraints)
- $true$ y $fail$

En consecuencia, la noción de un modelo de un constraint $S, \Theta \models C$ necesita ser extendida. La Figura 3.6 contiene las reglas de inferencia adicionales. Para preservar la forma de los constraints de sustitución definimos $S(\alpha \doteq \tau) = \alpha \doteq S(\tau)$, esto es, la sustitución se aplica solamente a la parte

$$\frac{S(\alpha) = \tau}{S, \Theta \models \alpha \doteq \tau} \qquad S, \Theta \models \varphi \doteq \varphi$$

Figura 3.6: Satisfacción de constraints, parte 2

$$\frac{\begin{array}{c}(x:\sigma) \in \Gamma \quad \sigma = \forall \widetilde{\alpha}\widetilde{\varphi}.C' \Rightarrow \tau' \quad \widetilde{\beta}, \widetilde{\varphi'} \textbf{ fresh} \\ S_0 = \left[\widetilde{\alpha} \mapsto \widetilde{\beta}, \widetilde{\varphi} \mapsto \widetilde{\varphi'}\right] \quad (S, C) = \mathit{normalize}(S_0(C'))\end{array}}{S|_{fv(\Gamma)}, C, \Gamma \vdash' x : S(\tau')}\ (var')$$

$$\frac{\begin{array}{c}S, C, \Gamma(f : \alpha \to \alpha')(x : \alpha) \vdash' e : \tau \quad \alpha, \alpha' \textbf{ fresh} \\ (S', C') = \mathit{normalize}(S \wedge C \wedge \tau \dot{\leq} \alpha')\end{array}}{S'|_{fv(\Gamma)}, C', \Gamma \vdash' \mathsf{rec}\ f(x)\ e : S'(\alpha \to \tau)}\ (rec')$$

$$\frac{\begin{array}{c}S, C, \Gamma \vdash' e : \tau \quad S', C', \Gamma \vdash' e' : \tau' \\ \alpha \textbf{ fresh} \quad (S'', C'') = \mathit{normalize}(S \wedge S' \wedge C \wedge C' \wedge \tau \dot{\leq} \tau' \to \alpha)\end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' e(e') : S''(\alpha)}\ (app')$$

$$\frac{\begin{array}{c}S_0, C_0 \wedge C_1, \Gamma \vdash' e : \tau \quad S', C', \Gamma(x : \forall \widetilde{\alpha}\widetilde{\varphi}.C_1 \Rightarrow \tau) \vdash' e' : \tau' \\ \{\widetilde{\alpha}, \widetilde{\varphi}\} = fv(\tau) \backslash fv(\Gamma) \quad fv(C_0) \subseteq fv(\Gamma) \\ (S'', C'') = \mathit{normalize}(S_0 \wedge S' \wedge C_0 \wedge C')\end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' \mathsf{let}\ x = e\ \mathsf{in}\ e' : S''(\tau')}\ (let')$$

Figura 3.7: Reglas de reconstrucción de tipos basadas en constraints

derecha de un constraint de sustitución. A pesar del símbolo simétrico $\doteq$, el constraint $\alpha \doteq \tau$ denota una sustitución de izquierda a derecha, es decir, los dos lados no deben intercambiarse.

El algoritmo de reconstrucción de tipos de la Figura 3.7 ha sido tomado casi literalmente del paper de HM(X) [Odersky *et al.*, 1999].

La única diferencia está en las reglas (rec') y (if'): HM(X) no considera recursión y condicional como características fundamentales sino que estipula que éstos están incluídos a través de constantes tipadas (como son aquí las constantes strings y la concatenación). El framework HM(X) define el juicio de reconstrucción de tipo:

$$S, C, \Gamma \vdash' e : \tau$$

donde el ambiente de tipos Γ y la expresión e son la entrada y la sustitución S, el constraint C y el tipo τ son la salida. La función *normalize* simplifica constraints y realiza todas las unificaciones necesarias. La Sección 3.2.2 explica esta función y expone algunas de sus propiedades. Si un constraint normaliza a $fail$ (quizás debido a un fallo de unificación), luego la regla de

inferencia llamada *normalize* también falla y no existe derivación. Para permitir un tratamiento unificado se considera equivalentes a una sustitución y a una conjunción de constraints de sustitución; convertimos entre las dos representaciones como sea conveniente.

La corrección y completitud de la reconstrucción de tipos con respecto la sistema lógico es inmediata de los correspondientes resultados para HM(X) [Odersky *et al.*, 1999].

Teorema 2 (Corrección de la reconstrucción) *Supongamos que* $S, C, \Gamma \vdash' e : \tau$. *Luego* $S(C), S(\Gamma) \vdash e : S(\tau)$

La formulación del teorema de completitud requiere nociones auxiliares adicionales que no hemos incluído en esta presentación pues dichos detalles no contribuyen a nuestro objetivo.

Normalización de constraints

La función *normalize* mapea un constraint a un par con una sustitución y un constraint. Esta función está definida por una aplicación exhaustiva de las reglas de normalización de la Figura 3.8. Las reglas están dadas en el estilo de reglas de manejo de constraints [Frühwirth, 1995]. La aplicación de una regla a un constraint C funciona de la siguiente manera:

- El constraint C se divide en $C = C_0 \wedge C_1$ tomando ventaja de la asociatividad y de la conmutatividad del $\wedge$ así como también del elemento unidad true.
- Una regla de la forma $C_1 \Leftrightarrow C_2$ se aplica reemplazando el constraint C_1 por el constrain C_2. El constraint final es $C_0 \wedge C_2$.
- Una regla de la forma $C_1 \Rightarrow C_2$ sólo se aplica si C_2 no está presente en C. En este caso la regla agrega el constraint C_2 y produce el constraint final $C_0 \wedge C_1 \wedge C_2$.
- Las reglas del segundo grupo se aplican a todo el constraint C (esto es $C_0 = \mathit{true}$). La notación $C@(C')$ es un as-pattern: liga todo el constraint a C y machea partes de C de acuerdo a C'.
- La condición al lado de la primera regla del segundo grupo es requerida para asegurar terminación. Es similar a un chequeo de ocurrencia para lo que utiliza la función $\mathit{reach}(C, \varphi)$ que produce el conjunto de variables de lenguaje alcanzables en uno o más pasos por φ en el grafo de dependencias inducido por el constraint C. El grafo de dependencias tiene un conjunto de nodos Φ y una arista dirigida $\varphi \rightarrow \varphi'$ para cada inclusión $r_1 \varphi' r_2 \dot{\subseteq} \varphi \in C$.

Cuando ninguna regla es aplicable, el constraint normalizado resultante se puede dividir en los constraints de sustitución de la forma $\alpha \doteq \tau$ y $\varphi \doteq \varphi'$ (que constituyen la sustitución) y el resto.

$$
\begin{array}{lcl}
Str(\varphi) \dot{\leq} Str(\varphi') & \Leftrightarrow & \varphi \dot{\subseteq} \varphi' \\
Str(\varphi) \dot{\leq} \tau \to \tau' & \Leftrightarrow & fail \\
Str(\varphi) \dot{\leq} \alpha & \Rightarrow & \alpha \doteq Str(\varphi') \ \ \textbf{si } \varphi' \textbf{ fresh} \\
\tau \to \tau' \dot{\leq} Str(\varphi') & \Leftrightarrow & fail \\
\tau \to \tau' \dot{\leq} \tau_1 \to \tau_1' & \Leftrightarrow & \tau_1 \dot{\leq} \tau \wedge \tau' \dot{\leq} \tau_1' \\
\tau \to \tau' \dot{\leq} \alpha & \Rightarrow & \alpha \doteq \alpha' \to \alpha'' \ \ \textbf{si } \alpha', \alpha'' \textbf{ fresh} \\
\alpha \dot{\leq} Str(\varphi') & \Rightarrow & \alpha \doteq Str(\varphi) \ \ \textbf{si } \varphi \textbf{ fresh} \\
\alpha \dot{\leq} \tau_1 \to \tau_1' & \Rightarrow & \alpha \doteq \alpha' \to \alpha'' \ \ \textbf{si } \alpha', \alpha'' \textbf{ fresh} \\
\alpha \dot{\leq} \alpha & \Leftrightarrow & true \\
\alpha \dot{\leq} \alpha' \wedge \alpha' \dot{\leq} \alpha'' & \Rightarrow & \alpha \dot{\leq} \alpha'' \\
\alpha \dot{\leq} \alpha' \wedge \alpha' \dot{\leq} \alpha & \Leftrightarrow & \alpha \doteq \alpha' \\
\alpha \doteq \alpha & \Leftrightarrow & true \\
\varphi \doteq \varphi & \Leftrightarrow & true \\
\varphi \dot{\subseteq} \varphi & \Leftrightarrow & true \\
\varphi \dot{\subseteq} \varphi' \wedge \varphi' \dot{\subseteq} \varphi & \Leftrightarrow & \varphi \doteq \varphi' \\
 & & \\
C@(\ C' \wedge & \Rightarrow & r_1 r r_2 \dot{\subseteq} \varphi' \ \ \textbf{si } \varphi \notin reach(C, \varphi) \\
\quad r \dot{\subseteq} \varphi \wedge & & \\
\quad r_1 \varphi r_2 \dot{\subseteq} \varphi') & & \\
C \wedge \varphi \doteq \varphi' & \Leftrightarrow & C\,[\varphi \mapsto \varphi'] \wedge \varphi \doteq \varphi' \\
C \wedge \alpha \doteq \tau & \Leftrightarrow & C\,[\alpha \mapsto \tau] \wedge \alpha \doteq \tau \ \ \textbf{si } \alpha \notin fv(\tau) \\
C \wedge \alpha \doteq \tau & \Leftrightarrow & fail \ \ \textbf{si } \alpha \in fv(\tau) \wedge \tau \neq \alpha \\
C \wedge true & \Leftrightarrow & C \\
C \wedge fail & \Leftrightarrow & fail
\end{array}
$$

Figura 3.8: Reglas de Normalización de Constraints

Lema 2 *Las reglas en la Figura 3.8 son correctas y completas. Supongamos que C reescribe a C'. Luego, $S, \Theta \models C$ si y sólo si hay extensiones S' de S y Θ' de Θ tales que $S', \Theta' \models C'$.*

Lema 3 *Las reglas en la Figura 3.8 son confluentes y terminantes.*

La aplicación exhaustiva de las reglas de propagación producen un constraint en forma normal.

Definición 1 *Un constraint C está en forma normal si C es true o fail o todas las conjunciones en C tienen una de las siguientes formas:*

1. $\alpha \doteq \tau$, $\alpha \notin fv(\tau)$ y α *no ocurre en el resto de* C,
2. $\alpha \dot{\leq} \alpha'$ *donde* α *y* α' *son diferentes,*
3. $\varphi \doteq \varphi'$ *donde* φ *y* φ' *son diferentes,*
4. $r \dot{\subseteq} \varphi$ *y para toda partición posible* $r_1\varphi' r_2 = r$, $reach(C, \varphi') = \emptyset$, $\varphi' \in reach(C, \varphi')$ *o existe algún* r' *tal que* $r_1 r' r_2 \dot{\subseteq} \varphi \wedge r' \dot{\subseteq} \varphi' \in C$.

Lema 4 *Si la aplicación de ninguna de las reglas de la Figura 3.8 modifica a C, luego C está en forma normal.*

Cropping

Un constraint en forma normal puede volverse bastante grande debido a la resolución de transitividad. Sin embargo, muchas de las conjunciones de constraints sólo sirven para definir y propagar resultados intermedios que pueden ser eliminados después de que la propagación termina. El siguiente ejemplo ilustra este punto.

Ejemplo 1 *Consideremos el término $\lambda x.a \cdot x$. El mismo da lugar al siguiente tipo restringido.*

$$\begin{array}{ll} & a \dot{\subseteq} \varphi_1 \\ \wedge & \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4 \\ \wedge & Str(\varphi_2) \to Str(\varphi_3) \to Str(\varphi_4) \dot{\leq} Str(\varphi_1) \to \alpha_2 \\ \wedge & \alpha_2 \dot{\leq} \alpha_1 \to \alpha_3 \\ \Rightarrow & \alpha_1 \to \alpha_3 \end{array}$$

La normalización y expansión de las sustituciones lleva a

$$\begin{array}{ll} & a \dot{\subseteq} \varphi_1 \wedge \varphi_1 \dot{\subseteq} \varphi_2 \wedge a \dot{\subseteq} \varphi_2 \\ \wedge & \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4 \wedge a \cdot \varphi_3 \dot{\subseteq} \varphi_4 \wedge \varphi_2 \cdot \varphi_3' \dot{\subseteq} \varphi_4 \wedge a \cdot \varphi_3' \dot{\subseteq} \varphi_4 \\ \wedge & \varphi_4 \dot{\subseteq} \varphi_4' \wedge \varphi_3' \dot{\subseteq} \varphi_3 \wedge \varphi_2 \cdot \varphi_3 \dot{\subseteq} \varphi_4' \wedge a \cdot \varphi_3 \dot{\subseteq} \varphi_4' \\ \wedge & \varphi_2 \cdot \varphi_3' \dot{\subseteq} \varphi_4' \wedge a \cdot \varphi_3' \dot{\subseteq} \varphi_4' \\ \Rightarrow & Str(\varphi_3') \to Str(\varphi_4') \end{array}$$

Para evitar la explosión de constraints, se propone el *cropping* para liberarse de la mayor cantidad de variables de lenguaje que sea posible. Más concretamente, sea V el conjunto de variables de lenguaje y C un constraint, donde V define el conjunto de variables de lenguaje que son mencionadas en el tipo resultante o en el ambiente de tipo; estamos interesados en un conjunto C' de C tal que cualquier solución de C' extiende a una solución de C.

Por lo tanto, estamos buscando los $C = C' \wedge C''$ tales que $S, \Theta \models C$ si y sólo si existen S', Θ' tales que $S', \Theta' \models C'$ y $S\,|_V = S'|_V$ y $\Theta\,|_V = \Theta'|_V$. Para hallar C', asumimos que C está en forma normal y fijamos

$$\begin{aligned} V' = V \cup \{\varphi' \in reach(C, V) \mid reach(C, \varphi') = \emptyset\} \\ \cup \{\varphi' \in reach(C, V) \mid \varphi' \in reach(C, \varphi')\} \end{aligned}$$

es decir, consideramos que V' es el conjunto formado por las variables de lenguaje que se pueden alcanzar en C desde alguna variable de V que no tienen sucesores o son recursivas.

Luego seleccionamos aquellos constraints que sólo mencionan variables en V' y mantenemos todos los constraints de subtipos.

$$\begin{aligned} C' = \bigwedge \left\{ r \dot{\subseteq} \varphi \in C \mid fv(r \dot{\subseteq} \varphi) \subseteq V' \right\} \\ \wedge \bigwedge \left\{ \tau \dot{\leq} \tau' \in C \right\} \end{aligned}$$

Lema 5 *Sean C y C' como en el párrafo anterior.*

1. *Si $S, \Theta \models C$, luego existen S', Θ' con $S', \Theta' \models C'$ y $S\,|_V = S'|_V$ y $\Theta\,|_V = \Theta'|_V$.*
2. *Si $S', \Theta' \models C'$, luego existen S y Θ con $S, \Theta \models C$ y $S'\,|_V = S|_V$ y $\Theta'\,|_V = \Theta|_V$.*

O sea, si existe modelo para alguno de los constraints, existe modelo para el otro constraint y ambos modelos son iguales en las variables de interés.

De ahora en adelante, se asume que la normalización también realiza cropping en cada regla de reconstrucción de tipos con respecto a las variables en el ambiente de tipo y el tipo resultante después de aplicar la sustitución computada.

Ejemplo 2 *Aplicando cropping al tipo restringido construído previamente, el constraint resultante es mucho más pequeño:*

$$a \cdot \varphi'_3 \dot{\subseteq} \varphi'_4 \Rightarrow Str(\varphi'_3) \rightarrow Str(\varphi'_4)$$

En este punto, cada constraint de inclusión de la forma $r \dot{\subseteq} \varphi$ puede ser considerado como la producción libre de contexto con las variables de lenguaje como no-terminales. De esta manera se llega a una situación similar

a la descrita por Christensen et al [2003], quien extrajo una gramática libre de contexto con lados derechos extendidos de los resultados del análisis de flujo.

Sin embargo, de ahora en adelante se toma una solución diferente. Mientras que Christensen aplica el algoritmo Mohri-Nederhof [2001] para transformar la gramática resultante en una gramática regular, en la solución dada aquí se intenta resolver los constraints de inclusión directamente usando la relación de reducción de una gramática de referencia libre de contexto.

3.2.3. Resolución con respecto a una gramática de referencia

Las reglas de normalización de la Figura 3.8 computan solamente la clausura transitiva de los constraints de inclusión, pero no intentan resolverlos de ninguna forma. Para obtener una noción útil de resolución para inclusión de constraints, recordemos que el problema de análisis de strings siempre se plantea en un contexto particular. Asumamos que ese contexto requiere que una cierta expresión de tipo string deba evaluarse por ejemplo a (un string conteniendo) una expresión XPath válida, una expresión SQL o un fragmento XML. En cualquiera de estos casos, el conjunto de strings legales es expresable mediante un lenguaje libre de contexto, que puede ser especificado por una gramática libre de contexto $G = (N, T, P, S)$, donde N es el conjunto de símbolos no-terminales, T es el conjunto de símbolos terminales, $P \subseteq N \times (N \cup T)^*$ el conjunto de producciones y $S \in N$ el símbolo de comienzo.

Por lo tanto, ya no estamos interesados en *todos* los modelos de un constraint, sino en aquellos modelos (S, Θ) donde la asignación Θ mapea variables de lenguaje a lenguajes *definidos en los términos de una gramática de referencia* G. Hay varias alternativas para definir un lenguaje en términos de G. Por ahora, nos concentraremos en los lenguajes $L_A(G) = \left\{\omega \in T^* \mid A \stackrel{*}{\Rightarrow} \omega\right\}$, esto es, el conjunto de strings derivables de $A \in N$ usando las producciones de G (donde $\Rightarrow \subseteq (N \cup T)^* \times (N \cup T)^*$ es la relación de derivación usual sobre strings de símbolos de la gramática G y $\stackrel{*}{\Rightarrow}$ su clausura transitiva reflexiva).

Debido a que una asignación de lenguaje Θ ahora está restringida a mapear las variables de lenguaje a uno de los $L_A(G)$, donde A es un no-terminal de G, cada Θ posible es definida completamente por un mapeo $\hat{\Theta} : \Phi \to N$ de variables de lenguaje a no-terminales. Resolver el constraint de inclusión $r \dot{\subseteq} \varphi$ en este seteo significa buscar una asignación a un no-terminal $\hat{\Theta} : \Phi \to N$ de variables de lenguaje a no-terminales tal que exista una derivación $\Theta(\varphi) \stackrel{*}{\Rightarrow} \Theta(r)$, donde $\Theta(\varphi) = L_{\hat{\Theta}(\varphi)}(G)$. De aquí en adelante, se sobrecarga Θ para expresar también la asignación de no-terminales $\hat{\Theta}$ y su asignación del lenguaje inducida.

Propocionar esta noción de modelo con respecto a la gramática de refe-

$$S, \Theta \models_G true \qquad \frac{S(\alpha) = \tau}{S, \Theta \models_G \alpha \doteq \tau} \qquad \frac{\Theta(\varphi) = \Theta(\varphi')}{S, \Theta \models_G \varphi \doteq \varphi'}$$

$$\frac{\Theta(\varphi') \stackrel{*}{\Rightarrow}_G \Theta(\varphi)}{S, \Theta \models_G Str(\varphi) \dot{\leq} Str(\varphi')}$$

$$\frac{S, \Theta \models_G \tau_1' \dot{\leq} \tau_1 \quad S, \Theta \models_G \tau_2 \dot{\leq} \tau_2'}{S, \Theta \models_G \tau_1 \to \tau_2 \dot{\leq} \tau_1' \to \tau_2'} \qquad S, \Theta \models_G \alpha \dot{\leq} \alpha$$

$$\frac{\Theta(\varphi) \stackrel{*}{\Rightarrow}_G \Theta(r)}{S, \Theta \models_G r \dot{\subseteq} \varphi} \qquad \frac{S, \Theta \models_G C \quad S, \Theta \models C'}{S, \Theta \models_G C \wedge C'}$$

$$\frac{S, \Theta \models_G S(C)}{S, \Theta \models_G C}$$

Figura 3.9: Satisfacción de constraints con respecto a una gramática de referencia G

rencia G requiere una definición refinada. La nueva definición reemplaza la relación de subconjuntos sobre lenguajes por la G-derivabilidad entre strings de símbolos de gramática.

Definición 2 *Sea S una sustitución y $\Theta : \Phi \to N$ un mapeo que asigna no-terminales a variables de lenguaje. El par (S, Θ) es un G-modelo del constraint C si $S, \Theta \models_G C$ es derivable usando las reglas de la Figura 3.9.*

Aserciones de Parsing

Puesto que solamente algunos lugares en un programa requieren que los strings tengan un formato específico, usualmente no hay necesidad de una solución total de los constraints en términos de la gramática de referencia. S. El programa hace la demanda para un cierto formato explícitamente usando un formato especial de constraints denominado *aserciones de parsing*. Una aserción de parsing es una expresión que dice que el valor string de la expresión debe ser derivable de un cierto símbolo no-terminal $A \in N$ de G.

$$e ::= ... \mid e \sqsubset A \qquad\qquad E ::= ... \mid E \sqsubset A$$

Su semántica está definida por la siguiente noción de reducción ($\longrightarrow$)

$$a_1 ... a_n \sqsubset A \quad \longrightarrow \quad a_1 ... a_n \qquad \textbf{si } A \models_G a_1 ... a_n$$

y su tipo está definido por

$$\cdot \sqsubset A : \forall \varphi . (\varphi \dot{\subseteq} A) \Rightarrow Str(\varphi) \to Str(\varphi)$$

donde $\varphi \dot{\subseteq} A$ es un *constraint de contención* que no es alcanzado por la propagación de constraints (recordar que los constraints de inclusión siempre tienen una variable de lenguaje en el lado derecho). La satisfacción de constraints se extiende a los constraints de contención por

$$\frac{A \stackrel{*}{\Rightarrow}_G \Theta(\varphi)}{S, \Theta \models_G \varphi \dot{\subseteq} A}$$

La corrección y completitud de la normalización de constraints no es afectada por la adición de este constraint porque no está involucrado en las reglas de propagación.

Como ejemplo del uso de una aserción de parsing, consideremos un programa en una versión extendida del cálculo que tiene acceso a un intérprete Lisp a través de una función de construcción eval. Eval toma un string conteniendo una expresión s y retorna el valor resultante también como un string. Si la gramática de referencia deriva expresiones s Lisp de un no-terminal A, luego la aserción de parsing en la expresión

$$eval(e \sqsubset A)$$

informa al análisis que todos los valores strings de e deben ser derivables de A, es decir, deben ser expresiones s.

Buscando soluciones

Encontrar asignaciones que resuelvan constrains de contención es simple, pero todavía resta encontrar asignaciones que resuelvan los constraints de inclusión. Una solución a fuerza bruta para resolver $C = r \dot{\subseteq} \varphi$ podría ser enumerar todas las posibles asignaciones de no-terminales a variables de lenguaje en C y luego verificar que $\Theta(\varphi) \stackrel{*}{\Rightarrow} \Theta(r)$. Resulta que la relación de derivación $\stackrel{*}{\Rightarrow}$ es una relación decidible sobre $(N \cup T)^*$ para cualquier lenguaje libre de contexto como argumentaremos en la siguiente sección. Sorprendentemente, una generalización del algoritmo de Earley para parsear lenguajes libres de contexto [Earley, 1970] es aplicable a un $\stackrel{*}{\Rightarrow}$ decidible sobre $(N \cup T)^*$ para una G arbitraria e incluso es fácil mejorar la enumeración a fuerza bruta de todas las asignaciones. La siguiente sección recupera la escencia de este algoritmo y lo generaliza para realizar la verificación de esa asignación.

3.2.4. Algoritmo de parsing de Earley

El algoritmo de parsing de Earley resuelve el problema formulado como "¿$a_1 ... a_n \in L(G)$?" para una gramática libre de contexto G. El algoritmo trabaja con una gramática libre de contexto arbitraria y corre en tiempo $O(n^3)$ en el peor de los casos.

init			$[\rightarrow \bullet S, 0] \in E_0$
scan	$[A \rightarrow \alpha \bullet a\beta, j] \in E_i$		
	y $a_{i+1} = a$	implica	$[A \rightarrow \alpha a \bullet \beta, j] \in E_{i+1}$
pred	$[A \rightarrow \alpha \bullet B\beta, j] \in E_i$		
	y $B \rightarrow \gamma \in P$	implica	$[B \rightarrow \bullet\gamma, i] \in E_i$
red	$[B \rightarrow \gamma\bullet, j] \in E_i$		
	y $[A \rightarrow \alpha \bullet B\beta, k] \in E_j$	implica	$[A \rightarrow \alpha B \bullet \beta, k] \in E_i$

Figura 3.10: Algoritmo de Earley

En esta sección presentamos el algoritmo original y su adaptación a formas sentenciales. Por convención, se usan las meta-variables $A, B, ...$ para los no-terminales N, las meta-variables $\alpha, \beta, ...$ para las palabras de los símbolos de gramática $(N \cup T)^*$, y las meta-variables X, X_i para los símbolos de la gramática $(N \cup T)$. Se denota $A \rightarrow \alpha$ a una producción de G.

Algoritmo original

El algoritmo de Earley [Earley, 1970] asigna un conjunto E_i de ítems de Earley a cada posición $i \in \{0, ..., n\}$ de una palabra de entrada $a_1, ...a_n$ donde 0 denota la posición justo delante del primer símbolo. Un ítem Earley tiene la forma $[A \rightarrow \alpha \bullet \beta, j]$ donde $A \rightarrow \alpha\beta$ es una producción y $0 \leq j \leq n$. El punto corresponde a la posición actual en la palabra de entrada, mientras que el índice j denota la posición en la entrada donde comienza la coincidencia con el lado derecho de la producción. El ítem plantea que la palabra entre la posición j y la posición actual es derivable de α. Para evitar introducir un nuevo símbolo de comienzo separado, los ítems degenerados de la forma $[\rightarrow \alpha \bullet \beta, j]$ también se usan. Los conjuntos E_i son los más chicos bajo la aplicación de las reglas de la Figura 3.10, para todo $0 \leq i \leq n$. La palabra de entrada es aceptada si $[\rightarrow S\bullet, 0] \in E_n$.

Adaptación a formas sentenciales

Cada string de símbolos terminales y no-terminales derivable de un símbolo de comienzo de una gramática libre de contexto es una *forma sentencial*. Debido a que el conjunto de todas las formas sentenciales es también un lenguaje libre de contexto (sólo extiende el conjunto de símbolos terminales por copias A' de todos los símbolos no-terminales A y agrega producciones $A \rightarrow A'$), el algoritmo de Earley también puede usarse para reconocer el lenguaje de formas sentenciales de G.

En lugar de construir la gramática modificada explícitamente y luego correr el algoritmo sobre la gramática modificada, el algoritmo puede modificarse para reconocer directamente el lenguaje de formas sentenciales. De he-

init			$[\rightarrow \bullet\gamma, 0] \in E_0$
scan	$[A \rightarrow \alpha \bullet X\beta, j] \in E_i$		
	y $X_{i+1} = X$	implica	$[A \rightarrow \alpha X \bullet \beta, j] \in E_{i+1}$
pred	$[A \rightarrow \alpha \bullet B\beta, j] \in E_i$		
	y $B \rightarrow \gamma \in P$	implica	$[B \rightarrow \bullet\gamma, i] \in E_i$
red	$[B \rightarrow \gamma\bullet, j] \in E_i$		
	y $[A \rightarrow \alpha \bullet B\beta, k] \in E_j$	implica	$[A \rightarrow \alpha B \bullet \beta, k] \in E_i$

Figura 3.11: Algoritmo generalizado de Earley

cho, el algoritmo modificado podrá decidir si $\gamma \stackrel{*}{\Rightarrow}_G \delta$ donde $\gamma, \delta \in (N \cup T)^*$.

La única diferencia con respecto a la formulación estándar del algoritmo de Earley es que los símblos no-terminales también pueden ser escaneados.

Para resolver las consultas de la forma "¿$\gamma \stackrel{*}{\Rightarrow}_G X_1...X_n$?", las reglas de Earley se generalizan como lo muestra la Figura 3.11. El algoritmo resultante es correcto en el siguiente sentido.

Lema 6 " $\gamma \stackrel{*}{\Rightarrow}_G X_1...X_n$" *si y sólo si* $[\rightarrow \gamma\bullet, 0] \in E_n$.

Lema 7 *El algoritmo de Earley generalizado termina después de* $O(n^3)$ *pasos.* La prueba es análoga a la prueba de Earley [Earley, 1970].

El algoritmo de Earley generalizado puede determinar si una asignación de no-terminales Θ resuelve un constraint $r \subseteq \varphi$ parseando el string $\Theta(r) \in (N \cup T)^*$ con la inicialización $[\rightarrow \bullet\Theta(\varphi), 0] \in E_0$.

Definición 3 *Escribimos* $\Theta \vdash_G r \dot{\subseteq} \varphi$ *si el parsing es exitoso, es decir, si* $fv(r \dot{\subseteq} \varphi) \subseteq dom(\Theta)$ *y* $\Theta(\varphi) \stackrel{*}{\Rightarrow}_G \Theta(r)$.

Ejemplo 3 *Considerar el constraint* $a.\varphi \dot{\subseteq} \varphi'$ *del Ejemplo 2 y la gramática G dada por las producciones* $S \rightarrow \epsilon \mid AS$ *y* $A \rightarrow a \mid AA$. *Hay tres parseos exitosos para* $\Theta \vdash_G a.\varphi \dot{\subseteq} \varphi'$:

$$\Theta 1 = [\varphi \mapsto A, \varphi' \mapsto A]$$
$$\Theta 2 = [\varphi \mapsto A, \varphi' \mapsto S]$$
$$\Theta 3 = [\varphi \mapsto S, \varphi' \mapsto S]$$

Debido a que el subtipado se define estructuralmente en términos de la relación de inclusión del lenguaje, el algoritmo produce una condición suficiente decidible para testear la relación de subtipado relativa a una asignación de no-terminales dada. Resta encontrar asignaciones de no-terminales adecuadas.

3.2.5. Constraints de asignación

Resolver un constraint de inclusión es esencialmente cuestión de buscar asignaciones de no-terminales adecuadas tal que el constraint pueda verificarse por parsing. Para cada constraint de inclusión puede haber múltiples asignaciones de no-terminales tales que el constraint sea parseable. Por lo tanto, se propone reemplazar un constraint de inclusión resuelto por un constraint de asignación que agrupe todas aquellas asignaciones que conduzcan a un parseo exitoso.

Además, el interés se encuentra en resolver sólo aquellos constraints que están sujetos a una aserción de parsing. Estos constraints pueden ser extraídos de un constraint en forma normal de la forma $\varphi_1 \dot{\subseteq} A_1 \wedge ... \wedge \varphi_n \dot{\subseteq} A_n \wedge C$ manteniendo aquellas inclusiones en C que sólo involucran variables de lenguaje $\varphi' \in reach(C, \varphi_1 ... \varphi_n)$ tal que $reach(C, \varphi') = \emptyset$ o $\varphi' \in reach(C, \varphi')$.

Luego se considera cada una de las inclusiones a la vez. Para cada una de ellas puede haber muchas asignaciones de no-terminales Θ que tomen el constraint *true*. Afortunadamente, el número de asignaciones de no-terminales es finito porque el constraint contiene un número finito de variables de lenguaje y porque hay una cantidad finita de símbolos no-terminales.

El algoritmo enumera todas las asignaciones de no-terminales, chequea si cada una de ellas conduce a un parseo exitoso usando el algoritmo de la Sección 3.2.4 y agrupa las exitosas en un constraint de asignación, o sea una disyunción de asignaciones $(\Theta_1 \vee ... \vee \Theta_k)$. Por construcción, todas las asignaciones en una disyunción tienen el mismo dominio. La definición de una solución se extiende a constraints de asignación como

$$\frac{(\exists i)\Theta_i \subseteq \Theta}{S, \Theta \models_G (\Theta_1 \vee ... \vee \Theta_k)}$$

Formalmente, la inclusión $r \dot{\subseteq} \varphi$ se reemplaza por $(\Theta_{i_1} \vee ... \vee \Theta_{i_k})$ si

- $\Phi_0 = fv(r \dot{\subseteq} \varphi)$ es el conjunto de variables de lenguaje que ocurren en el constraint de inclusión;
- $\Theta_1, ..., \Theta_n$ *son todas las posibles* asignaciones de no-terminales con $dom(\Theta_i) = \Phi_0$; y
- $I = \{i_1, ..., i_k\} \subseteq \{1, ...n\}$ tal que $i \in I \Leftrightarrow \Theta_i \vdash_G r \dot{\subseteq} \varphi$

Lema 8 *Bajo las condiciones listadas* $S, \Theta \models r \dot{\subseteq} \varphi$ *si y sólo si* $S, \Theta \models (\Theta_{i1} \vee ... \vee \Theta_{ik})$.

De manera análoga, el constraint de contención $\varphi \dot{\subseteq} A$ se reemplaza por $(\Theta_{i_1} \vee ... \vee \Theta_{i_k})$ si

- $\Phi_0 = fv(r \dot{\subseteq} A)$ es el conjunto de variables de lenguaje que ocurren en el constraint de contención;
- $\Theta_1, ..., \Theta_n$ *son todas las posibles* asignaciones de no-terminales con $dom(\Theta_i) = \Phi_0$; y
- $I = \{i_1, ..., i_k\} \subseteq \{1, ...n\}$ tal que $i \in I \Leftrightarrow A \stackrel{*}{\Rightarrow}_G \Theta_i(\varphi)$

Lema 9 *Bajo las condiciones listadas* $S, \Theta \models_G \varphi \dot{\subseteq} A$ *si y sólo si* $S, \Theta \models_G (\Theta_{i_1} \vee ... \vee \Theta_{i_k})$.

Es incompleto simplemente convertir $\varphi \dot{\subseteq} A$ en una asignación $[\varphi \mapsto A]$. Para ver esto, se pueden considerar las producciones $A \to B$, $B \to b$, $C \to B$ y el constraint $\varphi \dot{\subseteq} A \wedge \varphi \dot{\subseteq} \varphi'$. La asignación de A a φ evita la asignación de C a φ', por lo tanto se pierden soluciones.

Ejemplo 4 *Continuando con el Ejemplo 3, el constraint* $a.\varphi \dot{\subseteq} \varphi'$ *podría ser reemplazado por un constraint de asignación representando las tres soluciones posibles* Θ_1, Θ_2 *y* Θ_3. *Esto es*

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee [\varphi \mapsto S, \varphi' \mapsto S])$$

La aplicación exhaustiva del algoritmo de enumeración a todas las inclusiones y contenciones resulta en una conjunción de constraints de asignación:

$$(\Theta_1 \vee ... \vee \Theta_k) \wedge (\Theta'_1 \vee ... \vee \Theta'_l) \Longleftrightarrow \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Theta_i \bowtie \Theta'_j$$

El operador de join $\bowtie$ chequea si un par de asignaciones es compatible, es decir, si las asignaciones concuerdan en la intersección de sus dominios. Si el par de asignaciones es compatible, luego son mezcladas para formar una nueva asignación. Sino, el par es borrado usando la regla $x \vee fail = x$.

$$\Theta \bowtie \Theta' = \begin{cases} \Theta \cup \Theta' & \textbf{si } (\forall \varphi \in dom(\Theta) \cap dom(\Theta')) \quad \Theta(\varphi) = \Theta'(\varphi) \\ fail & \textbf{cc} \end{cases}$$

El constraint resultante es $fail$ o es un único constraint de asignación enumerando todas las posibles soluciones.

La reducción de conjunciones es también correcta y completa.

Lema 10 $S, \Theta \models_G (\Theta_1 \vee ... \vee \Theta_k) \wedge (\Theta'_1 \vee ... \vee \Theta'_l)$*si y sólo si* $S, \Theta \models_G \bigvee_{i=1}^{k} \bigvee_{j=1}^{l} \Theta_i \bowtie \Theta'_j$.

Ejemplo 5 *Comenzando desde el constraint de asignación del ejemplo anterior*

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee [\varphi \mapsto S, \varphi' \mapsto S])$$

se agrega un constraint de asignación más, sea $([\varphi \mapsto A])$. *El resultado de hacer join entre estos dos constraints de asignación es*

$$([\varphi \mapsto A, \varphi' \mapsto A] \bowtie [\varphi \mapsto A] \vee$$
$$[\varphi \mapsto A, \varphi' \mapsto S] \bowtie [\varphi \mapsto A] \vee$$
$$[\varphi \mapsto S, \varphi' \mapsto S] \bowtie [\varphi \mapsto A])$$

lo cual se simplifica a

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S] \vee fail)$$

y a

$$([\varphi \mapsto A, \varphi' \mapsto A] \vee [\varphi \mapsto A, \varphi' \mapsto S])$$

Debido a que las reglas de reducción de constraints dadas antes pueden ser aplicadas exhaustivamente para transformar todos los constraints de inclusión y contención en un sólo constraint de asignación, hay un algoritmo para resolver un constraint C en forma normal. Junto con el resultado anterior que la normalización de constraints transforma a un constraint en forma normal, tenemos lo siguiente.

Lema 11 *Hay un algoritmo que decide para un* C *y* S *dados si hay un* Θ *tal que* $S, \Theta \models_G C$.

Hay pocas formas obvias de mejorar la performance del algoritmo dado. Una de ellas es asociar con cada variable de lenguaje un conjunto de no-terminales que conduzcan a parseos exitosos y por lo tanto son asignados a la variable de lenguaje en un constraint de asignación. Otra forma más eficiente es además extender el algoritmo de parsing para inferir el constraint de asignación equivalente durante la construcción de los conjuntos de Earley.

3.2.6. Resumen

En resumen, el algoritmo de GBA toma como entrada un programa y una gramática de referencia e infiere el tipo restringido de dicho programa aplicando durante la inferencia las funciones *normalize* y *crop*, tal como se describe en la sección 3.2.2. A continuación, toma todos los constraints de inclusión ($r \dot{\subseteq} \varphi$) que figuran en el tipo restringido inferido y para cada uno de ellos calcula todas las asignaciones posibles de variables de lenguaje que aparecen en el constraint de inclusión a símbolos no-terminales de la gramática G ($\Theta : \Phi \to N$). Luego, toma cada asignación (Θ) computada para un constraint de inclusión particular y chequea si conduce a un parseo exitoso usando el algoritmo modificado de Earley, descripto en la Sección 3.2.4, agrupando las exitosas en un constraint de asignación, una disyunción de asignaciones ($\Theta_1 \vee ... \vee \Theta_k$). Luego de procesar todos los constraints de inclusión, tenemos una conjunción de constraints de asignación ($(\Theta_1 \vee ... \vee$

$\Theta_k) \wedge (\Theta'_1 \vee ... \vee \Theta'_l))$ a la cual se le aplica la función join ($\bowtie$), descripta en la Sección 3.2.5, para combinar todos las constraints de asignación en uno solo. Esta función chequea si un par de asignaciones es compatible, es decir, si las asignaciones concuerdan en la intersección de sus dominios. Si el par de asignaciones es compatible, luego son combinadas para formar una nueva asignación. Sino, el par es borrado.

Si tomamos como gramática de referencia a una gramática para el lenguaje SQL, por ejemplo, este método permite determinar que las ocurrencias de un string indexado con las variables que aparecen en el constraint de asignación sólo toman como valores sentencias SQL válidas que es justamente lo que se buscaba.

Capítulo 4

GBA para seguridad

En este capítulo primero explicaremos cómo pensamos utilizar la técnica de GBA para detectar vulnerabilidades de inyección, luego describiremos las modificaciones necesarias de la técnica que tuvimos que realizar para poder aplicarla a esta detección y finalmente contaremos las posibles modificaciones que podrían realizarse para mejorar la aplicación de la técnica al problema de seguridad.

En el desarrollo de este capítulo asuminos todo el vocabulario y nomenclatura del sistema GBA tal como fuera descripto en el capítulo 3.

4.1. Enfoque

El algoritmo de GBA toma como entrada un programa y una gramática de referencia, infiere el tipo del programa con las restricciones correspondientes y luego valida las mismas contra la gramática dada. Esta validación se realiza computando un constraint de asignación, es decir, una disyunción de asignaciones de variables de lenguaje a símbolos no-terminales de la gramática.

Para aplicar este resultado a la detección de vulnerabilidades de inyección necesitamos poder identificar qué variables de lenguaje son consideradas sensibles (utilizadas en contextos donde una vulnerabilidad podría ser explotada) y cuáles son los símbolos no-terminales inseguros (que modelan una vulnerabilidad por derivar en terminales que provienen del usuario y por lo tanto de un posible atacante). De esta forma, estamos interesados en aquellas asignaciones que involucran tanto variables de lenguaje sensibles como símbolos no-terminales inseguros, ya que si a una variable sensible se le puede asignar un no-terminal inseguro, estamos en presencia de una vulnerabilidad. En otras palabras, si alguna de las asignaciones computadas es una asignación de una variable a un símbolo no-terminal inseguro, podremos decir que existe una vulnerabilidad. Por ejemplo, en el código de la figura 4.1, donde la función `get` retorna el string ingresado por el usuario para

```
let
 uid    = get("UserName");
 psswd  = get("Password)"
 query = "SELECT * FROM users WHERE email=" . uid .
         "AND password =" . psswd
in
 mysql_query(query)
```

Figura 4.1: Ejemplo de SQL injection

la variable cuyo nombre recibe como parámetro y la función `mysql_query` envía la consulta recibida como parámetro en forma de string a la base de datos, especificando que la entrada de la función `mysql_query` es sensible podríamos decir que existe una vulnerabilidad si entre las asignaciones computadas existe una que asigna a la entrada de la función `mysql_query` un no-terminal inseguro (asumiendo que dicho no-terminal fue marcado inseguro por generar datos que pueden contener código malicioso insertado por un atacante). Explicaremos esto más detalladamente en la sección 4.2, luego de describir las modificaciones específicas necesarias.

4.2. Modificaciones Realizadas

En líneas generales, necesitamos enriquecer el lenguaje para poder escribir programas con vulnerabilidades, lo cual implica extender la teoría, particionar el alfabeto en dos conjuntos (caracteres seguros utilizados en el programa y caracteres inseguros provistos por el usuario), proveer una gramática G_V para cada tipo de vulnerabilidad V y enriquecer los constraints de modo de incorporar las funciones cuya entrada debe ser considerada sensible. De esta forma, la inferencia nos proveerá información sobre cómo se forman los strings usados para queries u otros tipos de ejecuciones no seguras y la resolución contra G_V establecerá si el programa dado como entrada tiene la vulnerabilidad V. A continuación se explican todos estos cambios en detalle.

El lenguaje considerado es básicamente el mismo definido en GBA con las diferencias que se describen a continuación.

En primer lugar, duplicamos el alfabeto generando cada caracter en una versión pura y una impura, para diferenciar los símbolos utilizados en el programa de los provistos por el usuario. Denominaremos $\sum$ a la versión de todos los caracteres puros, $\overline{\sum}$ a la copia impura y T a la unión de $\sum$ y $\overline{\sum}$.

En cuanto a las expresiones, mantenemos las usadas en el trabajo original y agregamos dos: *eq* y *prim*. La expresión *eq* sirve para comparar expresiones. Su sintaxis es: $e \;==\; e$. La expresión *prim* permite intro-

ducir funciones primitivas junto con su tipo. Las funciones primitivas son similares a las variables definidas con *let* pero sin código dado explícitamente (por ejemplo, operaciones de entrada/salida como *get*). Su sintaxis es $\mathsf{prim}\ x_1 :: t_1;\ ...\ ; x_n :: t_n\ \mathsf{in}\ e$. Una cambio accesorio fue la modificación de la sintaxis de la expresión *let* para que permita más de una variable en su definición (es accesorio pues en el lenguaje definido por Thiemman se puede hacer usando expresiones *let* anidadas). Su sintaxis es $\mathsf{let}\ x_1 = e_1;\ ...\ ; x_n = e_n\ \mathsf{in}\ e$.

También realizamos dos modificamos sobre la sintaxis de tipos: agregamos un nuevo constraint llamado *sink* y ampliamos la expresividad de los constraints de inclusión.

El constraint *sink* ($sink(\varphi)$) sirve para identificar aquellas variables de lenguaje que son sensibles, permitiendo de esa forma especificar que la entrada de una función es sensible, es decir que no debería ser manipulada por un atacante. Por ejemplo, en el código de la figura 4.1 queremos especificar que el string `query` pasado como entrada a la función `mysql_query` es sensible. Esto lo hacemos dando a la función `mysql_query` el siguiente tipo: $\forall\varphi, \varphi'.sink(\varphi) \Rightarrow Str(\varphi) \rightarrow Str(\varphi')$; observar el uso de φ, que es el índice del string input de la función.

En cuanto a la expresividad de los constraints de inclusión ($r \dot{\subseteq} \varphi$), ampliamos la misma para permitir que su lado izquierdo (r) pueda ser, además de lo definido por Thiemman (una variable del lenguaje (φ), el string vacío (ϵ), una constante a o una concatenación de los mismos ($r \cdot r$)), una constante *tainted* ($\overline{a}$), un comodín para caracteres del alfabeto puro (Σ) o un comodín para caracteres del alfabeto *tainted* ($\overline{\Sigma}$). Esto nos permite diferenciar entre caracteres puros y *tainted* en los constraints de inclusión y escribir nuevos constraints que expresen la capacidad de una variable de lenguaje (φ) de generar un caracter arbitrario (Σ) – y por lo tanto un string arbitrario escribiendo solamente dos constraints ($\epsilon \dot{\subseteq} \varphi$ y $\Sigma \cdot \varphi \dot{\subseteq} \varphi$).

Utilizar estos nuevos contraints de inclusión nos permite la identificación de *sources*. Un *source* es un punto del programa donde se generan datos que pueden producir problemas. Por ejemplo la función `get` que dado el nombre de una variable retorna el string ingresado por el usuario para esa variable, es considerada un *source* pues podría ser utilizada por un atacante para proveer entrada maliciosa. De esta manera, cuando queremos señalar que una función es un *source*, podemos hacerlo especificando que el tipo de salida de dicha función, por ejemplo $Str(\varphi)$, deriva en un subconjunto del alfabeto *tainted* ($r \dot{\subseteq} \varphi$, donde r es culquier string formado con terminales *tainted*). Para el ejemplo de la función `get`, identificarla como *source* significa darle el siguiente tipo: $\forall\varphi\varphi'.\epsilon \dot{\subseteq} \varphi',\ \overline{\Sigma} \cdot \varphi' \dot{\subseteq} \varphi' \Rightarrow Str(\varphi) \rightarrow Str(\varphi')$; nuevamente, observar el uso de la variable φ' que indexa al string resultante de ejecutar `get`.

Un cambio menor que no tiene demasiada importancia pero que vale la pena mencionar es que elegimos tener las operaciones básicas como funciones

Alfabeto	$\sum \bigcup \overline{\sum} = T$
Símbolos	$a, b \in \sum,\ \overline{a}, \overline{b} \in \overline{\sum}$
Palabras	$w \in (\sum \bigcup \overline{\sum})^*$
Expresiones	$e ::= x \mid e(e) \mid \mathsf{rec}\ f(x) \rightarrow e \mid \mathsf{let}\ x_1 = e_1; ...; x_n = e_n\ \mathsf{in}\ e$
	$\mid w \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid e{+}{+}e$
	$\mid e == e \mid \mathsf{prim}\ x_1 :: t_1;\ ...\ ; x_n = t_n\ \mathsf{in}\ e$

Figura 4.2: Sintaxis

Tipos	$\tau ::= \alpha \mid Str(\varphi) \mid \tau \rightarrow \tau$
Variables del lenguaje	$\varphi \in \Phi$
Tipos restringidos	$\rho ::= C \Rightarrow \tau$
Constraints	$C ::= true \mid \tau \dot{\leq} \tau' \mid r \dot{\subseteq} \varphi \mid C \wedge C \mid Sink(\varphi)$
Indices del tipo string	$r ::= \varphi \mid \epsilon \mid a \mid \overline{a} \mid \Sigma \mid \overline{\Sigma} \mid r \cdot r$
Esquemas de tipos	$\sigma ::= \forall \widetilde{\varphi} \widetilde{\alpha}.\rho$
Ambientes de tipos	$\Gamma ::= \emptyset \mid \Gamma(x : \sigma)$

Figura 4.3: Lenguaje de tipos

primitivas del lenguaje en lugar de constantes como lo hace Thiemman ($\cdot$, if, a) por una cuestión de simplicidad.

En las Figuras 4.2 y 4.3 presentamos la sintaxis del lenguaje y el lenguaje de tipos adaptados con las modificaciones antes mencionadas. Las reglas de tipos y las correspondientes reglas de reconstrucción de tipos basadas en constraints no han sido modificadas. No obstante en la Figura 4.4 se especifican las reglas de tipos para las nuevas expresiones y en la Figura 4.5 las correspondientes reglas de reconstrucción de tipos basadas en constraints. Observar que la igualdad retorna un string que modela un booleano, siendo el string vacío (ϵ) el valor falso y cualquier string w no vacío el valor verdadero (en concordancia con la semantica del if then else). Como trabajo futuro queda el desarrollo formal de la semántica y la prueba de corrección del sistema de tipos que no se encaró en esta etapa porque no era una prioridad para la empresa.

El último cambio consistió en modificar la gramática para que además de especificar el conjunto de símbolos terminales y no-terminales, las producciones y el símbolo de comienzo ($G = (N, T, P, S)$), especifique el conjunto de símbolos no-terminales inseguros ($G = (N, T, P, S, NI)$). De esta forma si alguna de las asignaciones computadas por el algoritmo incluye una variable de lenguaje especificada como *sink* y uno de estos símbolos inseguros, estaremos en presencia de una vulnerabilidad. Daremos un ejemplo de esta situación luego de finalizar la explicación de las modificaciones.

Por otro lado, al igual que hicimos con los constraints de inclusión, ampliamos la expresividad de la gramática permitiendo que en el lado derecho de las producciones se pueda distinguir entre símbolos terminales pu-

$$\frac{C||\!\vdash w \dot{\subseteq} \varphi}{C, \Gamma \vdash w \ : Str(\varphi)} \ (\mathsf{string})$$

$$\frac{C, \Gamma \vdash e : Str(\varphi) \quad C, \Gamma \vdash e_1 : \tau \quad C, \Gamma \vdash e_2 : \tau}{C, \Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ : \tau} \ (\mathsf{if})$$

$$\frac{C||\!\vdash \varphi_1 ++ \varphi_2 \dot{\subseteq} \varphi \quad C, \Gamma \vdash e_1 : Str(\varphi_1) \quad C, \Gamma \vdash e : Str(\varphi_2)}{C, \Gamma \vdash e_1 ++ e_2 : Str(\varphi)} \ (\mathsf{concat})$$

$$\frac{C||\!\vdash \epsilon \dot{\subseteq} \varphi \ \wedge \ w \dot{\subseteq} \varphi \quad C, \Gamma \vdash e_1 : Str(\varphi_1) \quad C, \Gamma \vdash e : Str(\varphi_2) \quad w \ no \ vacío}{C, \Gamma \vdash e_1 == e_2 : Str(\varphi)} \ (\mathsf{eq})$$

$$\frac{C, \Gamma(x_1 : \sigma_1 ... x_n : \sigma_n) \vdash e : \tau}{C, \Gamma \vdash \mathsf{prim}\ x_1 :: \sigma_1;\ ... ; x_n : \sigma_n\ \mathsf{in}\ e : \tau} \ (\mathsf{prim})$$

Figura 4.4: Reglas de tipos

$$\frac{(S'', C'') = normalize(w \dot{\subseteq} \varphi)}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' w \ : Str(\varphi)} \ (\mathsf{string}')$$

$$\frac{\begin{array}{c} C, \Gamma \vdash' e : \tau \quad C_1, \Gamma \vdash' e_1 : \tau_1 \quad C_2, \Gamma \vdash' e_2 : \tau_2 \quad \alpha, \varphi \ \ fresh \\ (S'', C'') = normalize(\tau < Str(\varphi) \ \wedge \ \tau_1 < \alpha \ \wedge \ \tau_2 < \alpha \ \wedge C \wedge C_1 \wedge C_2) \end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \ : S''(\alpha)} \ (\mathsf{if}')$$

$$\frac{\begin{array}{c} C_1, \Gamma \vdash' e_1 : \tau_1 \quad C_2, \Gamma \vdash' e_2 : \tau_2 \quad \varphi, \varphi_1, \varphi_2 \ \ fresh \\ (S'', C'') = normalize(\tau_1 < Str(\varphi_1) \ \wedge \ \tau_2 < Str(\varphi_2) \\ \wedge \varphi_1 . \varphi_2 \dot{\subseteq} \varphi \ \wedge C_1 \wedge C_2) \end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' e_1 ++ e_2 : Str(\varphi)} \ (\mathsf{concat}')$$

$$\frac{\begin{array}{c} C_1, \Gamma \vdash' e_1 : \tau_1 \quad C_2, \Gamma \vdash' e_2 : \tau_2 \quad \varphi, \varphi_1, \varphi_2 \ \ fresh \\ (S'', C'') = normalize(\tau_1 < Str(\varphi_1) \ \wedge \ \tau_2 < Str(\varphi_2) \\ \wedge \epsilon \dot{\subseteq} \varphi \ \wedge \ w \dot{\subseteq} \varphi \ \wedge C_1 \wedge C_2) \quad w \ no \ vacío \end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash' e_1 == e_2 : Str(\varphi)} \ (\mathsf{eq}')$$

$$\frac{\begin{array}{c} C, \Gamma(x_1 : \sigma_1 ... x_n : \sigma_n) \vdash' e : \tau \\ (S'', C'') = normalize(C) \end{array}}{S''|_{fv(\Gamma)}, C'', \Gamma \vdash \mathsf{prim}\ x_1 :: \sigma_1;\ ... ; x_n :: \sigma_n\ \mathsf{in}\ e : \tau} \ (\mathsf{prim}')$$

Figura 4.5: Reglas de reconstrucción de tipos basadas en constraints

```
prim
    get          :: ∀φφ′.ϵ ⊆̇ φ′, ⨆ · φ′ ⊆̇ φ′ ⇒ Str(φ) → Str(φ′)
    mysql_query  :: ∀φ, φ′.sink(φ) ⇒ Str(φ) → Str(φ′)
in let
    uid          = get  "UserName";
    psswd        = get  "Password";
    query        = "SELECT * FROM users WHERE email=" . uid .
                   "AND password =" . psswd
in
    mysql_query  query
```

Figura 4.6: Ejemplo completo de SQL Injection

ros y *tainted* y se puedan escribir los no-terminales reservados SIGMA y D_SIGMA, los cuales expresan implícitamente que el no-terminal del lado izquierdo de la producción puede reducirce a cualquier caracter válido (ya sea puro o *tainted* respectivamente) que pueda aparecer en un string. Cabe mencionar que esta modificación junto con la de los constraints de inclusión requiere una modificación del algoritmo generalizado de Earley especificado en la Figura 3.11. La definición de las funciones `pred` y `red` no sufre modificaciones ya que sólo involucra símbolos terminales. Sin embargo la función `scan` contempla cualquier símbolo que pueda aparecer en un string (X) y ahora éstos pueden ser no sólo símbolos terminales y no-terminales sino también los no-terminales reservados SIGMA y D_SIGMA que denotan cualquier caracter puro o *tainted* respectivamente. En consecuencia ampliamos la definición del algoritmo generalizado de Earley para contemplar estos casos en la función `scan` de la siguiente manera:

scan $[A \to \alpha \bullet \Sigma\beta, j] \in E_i$ y $(X_{i+1} = \Sigma$ o $X_{i+1} \in \sum)$
implica $[A \to \alpha\Sigma \bullet \beta, j] \in E_{i+1}$
scan $[A \to \alpha \bullet \overline{\Sigma}\beta, j] \in E_i$ y $(X_{i+1} = \overline{\Sigma}$ o $X_{i+1} \in \overline{\sum})$
implica $[A \to \alpha\overline{\Sigma} \bullet \beta, j] \in E_{i+1}$

En este punto estamos en condiciones de mostrar cómo pensamos utilizar el algoritmo de GBA modificado para la detección de vulnerabilidades de inyección. Consideremos el código dado en la Figura 4.6. Como podemos ver, el mismo utiliza dos primitivas: *get* y *mysql_query*. El tipo provisto para *get* indica que es un *source* y el tipo provisto para *mysql_query* indica que es un *sink*. Bajo esta especificación, podemos observar una vulnerabilidad de inyección (SQL injection), ya que en la última línea del código se invoca a la primitiva *mysql_query* con el parámetro *query*, cuyo valor es un string formado con los valores de las variables *uid* y *psswd*, los cuales son obtenidos a través de la primitiva *get* que es un source.

Nuestro algoritmo de inferencia para este código arroja el siguiente tipo:

$\epsilon \dot{\subseteq} \varphi_2$,
$\overline{\Sigma} \cdot \varphi_2 \dot{\subseteq} \varphi_2$,
$\epsilon \dot{\subseteq} \varphi_3$,
$\overline{\Sigma} \cdot \varphi_3 \dot{\subseteq} \varphi_3$,
`"SELECT * FROM users WHERE email = "++`
`  `φ_2`++ " AND password = "++` $\varphi_3 \dot{\subseteq} \varphi_1$,
$sink(\varphi_1)$
$\Rightarrow Str(\varphi_5)$

que expresa que φ_1 es un *sink* que deriva en una concatenación de strings involucrando las variables φ_2 y φ_3, las cuales derivan en un string *tainted*.

Finalemente, necesitamos proveer una gramática G_V donde V representa la vulnerabilidad SQL injection. A continuación damos una versión simplificada de tal gramática (no intentamos capturar la estructura de SQL, como podría y debería hacerse):

G_V = (

```
    S           → DANGEROUS | HARMLESS,
    DANGEROUS   → D_SIGMA,
    DANGEROUS   → "",
    DANGEROUS   → HARMLESS DANGEROUS,
    DANGEROUS   → DANGEROUS DANGEROUS,
    HARMLESS    → "SELECT",
    HARMLESS    → "*",
    HARMLESS    → "FROM",
    HARMLESS    → "users",
    HARMLESS    → "WHERE",
    HARMLESS    → "email",
    HARMLESS    → "=",
    HARMLESS    → "AND",
    HARMLESS    → "password",
    HARMLESS    → " ",
    HARMLESS    → "",
    HARMLESS    → HARMLESS HARMLESS
    DangerousNTs:
    DANGEROUS, S
```

)

Notar que en esta gramática no estamos especificando explícitamente los símbolos terminales y no-terminales, sino sólo las producciones y los no-terminales inseguros. Esto se debe a que asumimos que los símbolos terminales son todos los caracteres válidos que pueden aparecer en un string y que los símbolos no-terminales están implícitos en las producciones – son

todos aquellos que no están encerrados entre comillas dobles. Tampoco especificamos el símbolo de comienzo pues se asume que es el primero para el que se da producciones (en este caso `S`).

Como podemos ver, esta gramática indica que el no-terminal `DANGEROUS` es inseguro, pues se reduce a un string conteniendo caracteres del alfabeto *tainted* y que un string seguro concatenado con uno inseguro se convierte en inseguro. Nuestro algoritmo de resolución de constraints para el tipo inferido mostrado anteriormente usando esta gramática arroja el siguiente constraint de asignación:

$\Theta_1 \vee \Theta_2$, donde $\Theta_1 := [\varphi_1 \mapsto \texttt{S}]$ $\Theta_2 := [\varphi_1 \mapsto \texttt{DANGEROUS}]$

Debido a que φ_1 es una variable de lenguaje especificada como sensible y existe una asignación de φ_1 a un no-terminal inseguro (`DANGEROUS`) nuestro algoritmo detecta la vulnerabilidad como se deseaba.

Cabe mencionar que para que esta técnica sea útil necesitamos tener un conjunto de gramáticas que capturen diversas vulnerabilidades que se quieran detectar. Por ejemplo, la gramática presentada anteriormente es una simplificación de una gramática que captura una vulnerabilidad SQL injection. Además se pueden estudiar alternativas para realizar preguntas sobre la salida del algoritmo dado que la misma puede interpretarse de diversas maneras de acuerdo a las preguntas que se hagan. En nuestro caso, preguntamos si en la salida del algoritmo existe una asignación de una variable de lenguaje sensible a un no-terminal inseguro, pero pueden realizarse otras preguntas que arrojen a resultados aún más interesantes. El usuario que utilice una herramienta basada en este enfoque debe capturar las vulnerabilidades que quiere detectar proveyendo distintas gramáticas que las expresen. Esto muestra la flexibilidad del enfoque.

4.3. Modificaciones a futuro

Existen diversos aspectos a tratar fuera del alcance del presente trabajo que contribuirían a perfeccionar la aplicación de la técnica GBA para detectar vulnerabilidades.

La primera modificación sería estudiar la incorporación de las operaciones de descomposición de strings en su primer caracter y el resto, conocidas como Head y Tail. Esto implica definir el efecto de estas operaciones en forma de constraints y conservar esta definición a lo largo de las diversas fases de transformación a las que la técnica somete a los strings. A partir de dicha incorporación se podrían analizar genéricamente las funciones que descomponen a los strings, particularmente y para nuestro interés, aquellas funciones de sanitización (funciones que tranforman un string inseguro en seguro) que eliminan caracteres del string de entrada. A modo de ejemplo podemos observar el siguiente código que incluye la función de sanitización *sanit*:

```
prim
    get           :: ∀φφ'.ϵ ⊆̇ φ', ∑̄ · φ' ⊆̇ φ' ⇒ Str(φ) → Str(φ')
    mysql_query   :: ∀φ, φ'.sink(φ) ⇒ Str(φ) → Str(φ')
    sanit         :: ????
in let
    uid           = get  "UserName";
    psswd         = get  "Password";
    uid_sanit     = sanit  uid;
    psswd_sanit   = sanit  psswd;
    query         = "SELECT * FROM users WHERE email=" . uid_sanit .
                    "AND password =" . psswd_sanit
in
    mysql_query  query
```

donde *sanit* es una función que recibe un string y mediante operaciones de descomposición elimina del mismo caracteres peligrosos y retorna el resultado. Sin operaciones de descomposición *Head* y *Tail* no podemos definir el código de *sanit* y por lo tanto debemos especificar su tipo mediante la primitiva *prim* como figura en el ejemplo de arriba. El uso de *prim* es útil para expresar el tipo de una función cuyo código no es necesario analizar; sin embargo consideramos inapropiado usarlo para funciones de sanitización debido a que una caracterización inválida de su tipo puede conducir a resultados incorrectos. En el ejemplo anterior, si el tipo de *sanit* no se especifica correctamente, el análisis podría conducir a un falso positivo, o peor aún a un falso negativo. Un falso positivo es la detección errónea de una vulnerabilidad: se indica la existencia de una vulnerabilidad cuando en realidad no existe. Un falso negativo se produce cuando no se detecta la presencia de una vulnerabilidad existente.

Además, la técnica necesita, para su correcto funcionamiento, gramáticas con formulaciones precisas (ambiguas y basadas en caracteres), mientras que el uso actual utiliza gramáticas con otro tipo de formulaciones (no ambiguas y basadas en tokens). Consideramos apropiado estudiar la manera de ambiguar gramáticas y destokenizarlas, así como adecuar la técnica para que permita el uso de gramáticas no ambiguas y basadas en tokens.

Existe un proyecto propuesto por LIFIA para una cooperación con Alemania con el grupo del Dr. Thiemann que se propone abordar todos estos temas.

Otra tarea interesante sería modificar el proceso para detectar e informar de dónde viene la vulnerabilidad. Es decir, en lugar de detectar e informar que existe una vulnerabilidad como lo hacemos ahora, se podría dar información más precisa detectando en qué parte del código se encuentra la vulnerabilidad y cuáles son las causas que la producen. Por ejemplo, para el código de la figura 4.6 la información de salida del algoritmo podría ser que existe una vulnerabilidad en la línea 9 (última línea) dado que la funciíon

mysql_query ha sido marcada como *sink* y su argumento es un dato inseguro. Además podría brindarse información de contexto que permita determinar en qué punto del código el parámetro de la función marcada como *sink* se convirtió en inseguro, es decir, de que *source* proviene.

Una modificación aún más ambiciosa podría ser extender el proceso para la construcción de exploits de manera automática o semiautomática. Un exploit es un programa informático malicioso, o parte del programa, que trata de forzar alguna deficiencia o vulnerabilidad de otro programa (llamadas bugs) y es usado normalmente para explotar una vulnerabilidad en un sistema. En nuestro caso, construir un exploit significa dar el string necesario para que se produzca la vulnerabilidad. Por ejemplo, en el código de la figura 4.6 un posible exploit sería que el valor de la variable *uid* fuera `"alice@host' OR '0'='1"` ya que la aplicación ejecuta una consulta cuyo resultado es independiente del código de pin provisto tal como se explicó en el ejemplo de la sección 2.2.1. Esta extensión requeriría estudiar la forma de dar las gramáticas con mayor nivel de detalle en cuanto a la manera de expresar vulnerebilidades (el análisis de funciones de sanitización también se vería favorecido por esto).

La última modificación que mencionaremos requiere mucho más estudio y dedicación que las anteriores y consiste en transladar los resultados a lenguajes con otras características (imperativos, orientados a objetos) para de esa forma poder utilizar la herramienta para detectar vulnerabilidades en programas escritos en lenguajes reales como Java, PHP, etc.

Como podemos observar, nuestro trabajo de grado es solamente un primer paso a la utilización de la técnica de GBA en la detección de vulnerabilidades de inyección. Si bien el trabajo realizado ha sido muy satisfactorio tanto para nosotras como para la gente de la empresa Core y los resultados han sido altamente exitosos por haber quedado demostrada la utilidad de la técnica para el fin propuesto, quedan muchas modificaciones por hacer que permitirán el crecimiento de la herramienta con el objetivo de convertirla en una herramienta que automatice el trabajo de auditoría realizado por muchas empresas de seguridad.

Capítulo 5

Descripción de la Implementación

En este capítulo describiremos la implementación del prototipo que desarrollamos en el lenguaje de programación funcional Haskell. La selección del lenguaje de programación la realizamos con el equipo de CORE basándonos en el alto poder expresivo que tienen los lenguajes funcionales.

En primer lugar explicaremos la arquitectura del sistema mediante una breve descripción de cada módulo implementado. Luego presentaremos en detalle una estructura de datos que diseñamos para optimizar la normalización de constraints; denominamos a dicha estructura GIxTrees. Por último daremos un ejemplo de uso del prototipo implementado.

5.1. Arquitectura del Sistema

El sistema fue estructurado en diversos módulos. Cada módulo implementa una funcionalidad particular del sistema. A continuación ofrecemos una breve descripción de cada uno de ellos.

Definición de las estructuras de datos del lenguaje
En este módulo se encuentran las estructuras de datos usadas para representar expresiones, tipos y constraints del lenguaje. Los mismos son representados mediante tipos algebraicos. También se incluye el cálculo de variables libres de tipo y de lenguaje (mediante la clase FVs).

Módulo para parsing
El problema de parsing consiste básicamente en obtener una estructura de datos a partir de una descripción elemental de la misma, usualmente en formato de texto. Para implementar el parsing de expresiones y de gramáticas usamos una técnica muy conocida en programación funcional conocida como Combinadores de Parsing. Los Combinadores de Parsing resuelven el pro-

blema de parsing mediante Combinadores, funciones que toman soluciones a subproblemas y las combinan en una solución al problema capturando muy bien la idea de programación modular.

Nos basamos en el trabajo de Graham Hutton y Eric Meijer [Hutton and Meijer, 1996] y en una implementación de John Hughes, para lo cual seguimos la línea de lo realizado por Martínez López en su tesis [Martínez López, 2005].

Módulo para pretty printing
El módulo de pretty printing también fue implementado utilizando Combinadores. Nos basamos en el trabajo de John Hughes [Hughes, 1995]. En el mismo estudia una biblioteca de combinadores para Pretty-printing como un ejemplo para guiar el diseño y la implementación de una biblioteca de combinadores.

Básicamente las funciones de pretty printing recorren recursivamente las estructuras de datos armando la salida correspondiente.

Mónada de Inferencia
En este módulo se especifica la mónada usada para la inferencia de tipos, denominada TIM. La programación monádica [Wadler, 1995] es un estilo de programación funcional que proporciona una manera simple y atractiva de abordar la interacción en un entorno funcional. Las principales características suministradas por nuestra mónada son: manejo de variables libres (de tipo y de lenguaje), manejo de sustituciones, administración de la información útil para realizar la normalización y tracing para debbuging.

Inferencia
Este módulo implementa las reglas de inferencia de tipos con constraints para todas las expresiones del lenguaje, que fueron presentadas en los Capítulos 3 y 4.

La inferencia es implementada por una función que hace uso de la mónada TIM. Las diferentes reglas se implementan según el caso, dependiendo de la expresión discriminada mediante pattern matching. La implementación de la inferencia en cada regla sigue el esquema general de inferir el tipo restringido de las subexpresiones, agregar nuevos constraints a los constraints resultantes dependiendo de la regla, luego normalizar los constraints y una vez aplicada la sustitución realizar crop para obtener el tipo restringido final.

En cuanto a la función de normalización, la misma procesa primero los constraints de subtipos y luego los constraints de inclusión, valiéndose de funciones intermedias. Para optimizar el recorrido de los constraints, utilizamos una estructura de datos donde los mismos se dividen en 3 clases según sean constraints de subtipos, de inclusión u otros. De esta manera la función que analiza los constraints de subtipos sólo trabaja con el subconjunto de constraints de subtipos y lo mismo ocurre con la función que analiza los

constraints de inclusión. Cada una de estas funciones mantiene dos listas de constraints: los constraints a analizar y los constraints resultantes de la aplicación de las reglas. Para cada constraint de la primera lista se determina la regla a aplicar, si existiese, y se agregan los nuevos constraints en la segunda lista. Cuando la regla establece que debe realizarse cierta sustitución en los constraints, primero se extiende el ambiente con dicha sustitución y luego se invoca a una función intermedia que se encarga de aplicar las sustituciones indicadas por el ambiente en ambas listas y continuar el procesamiento.

Contamos con 3 alternativas para normalizar los constraints, las cuales nos resultaron útiles para el estudio de casos intermedios. Cada una de ellas genera distintos constraints en el tipo de salida para la expresión del lenguaje analizada y básicamente son: normalizar sólo los constraints de subtipos, normalizar los constraints de subtipos y de inclusión (sin incluir la implementación de una regla específica que es tratada con GIxTrees y describiremos en la siguiente sección) y normalizar todos los constraints.

Solving

El módulo de solving incluye las estructuras de datos para representar las gramáticas e implementa el solving de constraints de derivación. Incluye la implementación del algoritmo de Earley tal como lo presentamos en los Capítulos 3 y 4. También realiza la normalización de la gramática llevando todos los terminales a forma de caracter lo cual es necesario para el correcto funcionamiento del algoritmo de Earley.

Interface

El módulo de interface permite la interacción con el prototipo mediante un conjunto de comandos. Los comandos que pueden ejecutarse mediante la interface están clasificados en comandos generales, comandos para paths, comandos para módulos y comandos de aplicación. Los mismos se detallan en el Capítulo 6.

5.2. Estructura de datos usada para normalización de constraints

La normalización de constraints está definida por la aplicación exhaustiva de las reglas de normalización presentadas en la Figura 8 del Capítulo 3. Con el objetivo de optimizar la función de normalización diseñamos e implementamos una estructura de datos que denominamos GIxTrees para tratar de manera exclusiva la regla más costosa:

$$\begin{array}{lll} C@(\ C' \wedge & \Rightarrow & r_1 r r_2 \dot{\subseteq} \varphi' \ \ \textbf{si}\ \varphi \notin reach(C, \varphi) \\ \quad r \dot{\subseteq} \varphi \wedge & & \\ \quad r_1 \varphi r_2 \dot{\subseteq} \varphi') & & \end{array}$$

Esta regla establece que ante la presencia de los constraints de inclusión $r \dot{\subseteq} \varphi$ y $r_1 \varphi r_2 \dot{\subseteq} \varphi'$, debe agregarse el constraint $r_1 r r_2 \dot{\subseteq} \varphi'$ siempre y cuando φ no pertenezca al conjunto de variables alcanzables en uno o más pasos desde ella misma en el grafo de dependencias inducido por el constraint C; esta condición es necesaria para asegurar terminación. El grafo de dependencias tiene un conjunto de nodos que son las variables de lenguaje, y una arista dirigida $\varphi \to \varphi'$ para cada inclusión $\tau_1 \varphi' \tau_2 \dot{\subseteq} \varphi \in C$.

La estructura de datos que diseñamos es una estructura de datos recursiva con estructura no lineal que permite capturar todos los posibles caminos a partir de una variable de lenguaje en el grafo inducido por un constraint C1. De esta manera conseguimos una representación jerárquica de la información de constraints de nuestro interés a la cual podemos manipular más eficientemente.

Para implementar la estructura de datos en Haskell introducimos un nuevo tipo algebraico. Un tipo algebraico se define dando la forma de cada elemento con constructores (que son funciones especiales). Los árboles se pueden representar con tipos algebraicos recursivos y los GIxTrees que definimos tienen la particularidad de que utilizan una lista de nodos (los cuales pueden contener los casos recursivos). Dicha representación permite usar las funciones de alto orden existentes para la manipulación de las listas y de este modo facilitar la programación de operaciones sobre la estructura.

A continuación presentamos la definición en Haskell de nuestra estructura de datos:

```
data GIxNode a = GISym Symbol | GITree (GIxTree a)
data GIxTree a = GIN a [ [GIxNode a] ]
```

donde un `GixTree a` es un nodo que contiene un valor de tipo `a` y una lista de listas de `GIxNode a`. Un `GIxNode a` es o bien un nodo hoja `GISym Symbol` que contiene un valor de tipo `Symbol` o bien un nodo interno que contiene un `GIxTree a`.

A modo de ejemplo consideremos la siguiente lista de constraints:

$\epsilon \dot{\subseteq} \varphi_1$,
φ_2`++`$\varphi_3 \dot{\subseteq} \varphi_1$,
φ_4`++`$\varphi_5 \dot{\subseteq} \varphi_2$,
`"a"` $\dot{\subseteq} \varphi_4$,
`"b"` $\dot{\subseteq} \varphi_5$,
`"c"` $\dot{\subseteq} \varphi_3$,
$\varphi_8 \dot{\subseteq} \varphi_6$

El GIxTree formado a partir de la variable φ_1 puede visualizarse en la Figura 5.1. Observar que el GIxTree está formado por el valor φ_1 en la raíz y una lista de listas de `GixNode a` cuyo primer elemento es una lista con

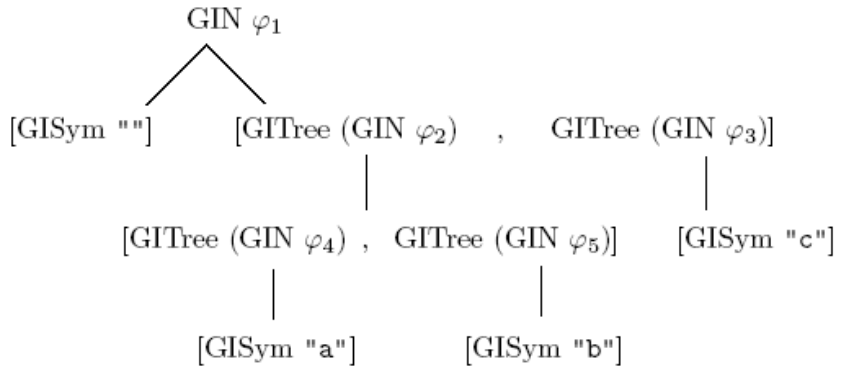

Figura 5.1: GIxTree

un único elemento `GISym ""` y cuyo segundo elemento es una lista con dos elementos de tipo `GITree a`, cada uno de los cuales representa el GIxTree correspondiente a las variables φ_1 y φ_2 respectivamente.

Normalización usando GIxTrees

En primer lugar, normalizamos los constraints de subtipos y los constraints de inclusión sin considerar la regla implementada por los GIxTrees. Luego construímos y tratamos un GIxTree por cada una de las variables de lenguaje mencionadas en los constraints de inclusión. El tratamiento de cada uno de los GIxTrees se realiza en tres etapas bien definidas: generación, pruning e interpretación. Cada una es implementada por una función diferente.

La etapa de generación consiste en crear el GIxTree a partir de una lista de constraints de inclusión y de una variable.

La etapa de pruning es necesaria para asegurar terminación. Por ejemplo, si en la lista de constraints mencionada arriba se agrega el constraint $\varphi_1 \dot{\subseteq} \varphi_3$, la generación del GIxTree es claramente infinita. La etapa de pruning elimina dichas expansiones infinitas de ciclos en el grafo inducido representado por el árbol. La función que realiza el pruning toma como entrada el árbol resultante de la etapa de generación; sin embargo ambas etapas trabajan conjuntamente gracias a que la evaluación *lazy* permite que un argumento no sea necesariamente evaluado por completo; sólo se evalúan aquellas partes que contribuyen efectivamente al cómputo.

La etapa de interpretación se encarga de generar una lista de constraints de inclusión a partir del GIxTree construído. Cada uno de los constraints de inclusión tiene como variable de lenguaje a la raíz del árbol y como índices a los nodos hojas. Para armar dicha salida contamos con funciones de interpretación que recorren recursivamente la estructura del GIxTree. Puesto

que el formato final no se corresponde con los datos necesarios para armar las partes intermedias de los llamados recursivos, definimos una función recursiva que recorre el árbol para recolectar la información en una estructura intermedia que luego es utilizada para armar la salida final. De manera más específica, la interpretación de un GIxTree raíz da lugar a un conjunto de constraints de inclusión mientras que un GIxTree interno da lugar a una lista de listas de índices, cada una indicando el lado izquierdo de una inclusión. Además, para implementar adecuadamente esta última etapa tuvimos que poner especial atención en el tratamiento de variables de lenguaje recursivas (es decir, en aquellas variables alcanzables en uno o más pasos desde ellas mismas en el grafo de dependencias inducido por el constraint).

La lista de constraints resultante del proceso de interpretación a partir del GIxTree del ejemplo es:

$\epsilon \dot{\subseteq} \varphi_1$,
`"a"++"b"++"c"` $\dot{\subseteq} \varphi_1$

El resultado final de la normalización de los constraints de inclusión usando GIxTrees es la concatenación de cada una de las listas de constraints que resultan de la interpretación de los GIxTrees generados por cada una de las variables de lenguaje mencionadas en los constraints de inclusión inicial.

5.3. Ejemplo de Uso

A continuación presentaremos un ejemplo de uso del prototipo en el cual podrá notarse cómo participa cada uno de los módulos antes descriptos en la generación de la solución. Además mostraremos a partir del ejemplo cuáles son los constraints resultantes del algoritmo de normalización de acuerdo a que se consideren sólo los constraints de subtipos, los constraints de subtipos más los de inclusión sin incluir la regla tratada con GIxTrees, o todos los constraints.

Paso 1: Carga del código a analizar.

El código a analizar debe colocarse en un archivo y se invoca su carga desde la interface (ver Capítulo 6). A continuación ofrecemos un código para el ejemplo presentado en el Capítulo 4:

```
prim
   get         :: S_2 > d_sigma++S_2,
                  S_2 > epsilon => String(S_1) -> String(S_2);
   mysql_query :: sink(S_0) => String(S_0) -> String(S_1)
in let
    uid    = get @ "UserName";
    psswd  = get @ "Password"
```

```
in let
    query  = "SELECT * FROM users" ++
             " WHERE email = "    ++ uid ++
             " AND password = " ++ psswd
in
  mysql_query @ query
```

Este texto es la entrada del módulo de parsing de expresiones; si la sintaxis es correcta se obtiene como resultado una expresión cuya estructura se encuentra definida en el módulo de definiciones de estructuras de datos del lenguaje. Si la sintaxis no es correcta se muestra un mensaje indicando la línea en la cual se encuentra el error sintáctico. La expresión parseada puede visualizarse en la consola (tarea realizada por el módulo de pretty printing) cuyo resultado en este caso es el siguiente:

```
prim get ::
       \/S_1 S_2.
         ( S_2 > D_SIGMA++S_2 ),
         ( S_2 > epsilon )
          => String(S_1) -> String(S_2)
   ; mysql_query ::
       \/S_1 S_2. ( sink(S_2) ) => String(S_2) -> String(S_1)
 in
let uid = get @ "UserName"
  ; psswd = get @ "Password"
 in
let query =
      "SELECT * FROM users" ++
     " WHERE email = " ++ uid ++
     " AND password = " ++ psswd
 in mysql_query @ query
```

Paso 2: Inferencia.

Es posible inferir el tipo de la expresión anterior. El comportamiento de la normalización de contraints puede variarse alternando el valor de una variable en el módulo de inferencia (no se provee como parámetro pues sólo la utilizamos para debugging). Mostramos las distintas salidas de la inferencia dependiendo de la normalización realizada. En la Figura 5.2 mostramos el tipo resultante sin normalizar; se puede observar el gran número de constraints y el uso de variables intermedias, tanto de tipos (`ft_i`) como de lenguaje (`fd_i`). En la Figura 5.3 mostramos el tipo obtenido a partir de la normalización de los constraints de subtipos únicamente; dicha normalización tiene como objetivo eliminar la mayor cantidad de variables de tipo (en este caso puede observarse que tales variables fueron eliminadas por completo). En la Figura 5.4 mostramos el tipo obtenido mediante la normalización

```
( fs_29 > fs_30++fs_31 ),
( fs_32 > fs_33++fs_34 ),
( fs_35 > fs_36++fs_37 ),
( fs_38 > fs_39++fs_40 ),
( fs_41 > "SELECT * FROM users" ),
( fs_42 > " WHERE email = " ),
( fs_43 > D_SIGMA++fs_43 ),
( fs_43 > epsilon ),
( fs_44 > "UserName" ),
( fs_45 > " AND password = " ),
( fs_46 > D_SIGMA++fs_46 ),
( fs_46 > epsilon ),
( fs_47 > "Password" ),
( String(fs_27) -> String(fs_28) < String(fs_29) -> ft_6 ),
( String(fs_32) < String(fs_30) ),
( ft_4 < String(fs_31) ),
( String(fs_35) < String(fs_33) ),
( String(fs_45) < String(fs_34) ),
( String(fs_38) < String(fs_36) ),
( ft_5 < String(fs_37) ),
( String(fs_41) < String(fs_39) ),
( String(fs_42) < String(fs_40) ),
( String(fs_48) -> String(fs_43) < String(fs_44) -> ft_5 ),
( String(fs_49) -> String(fs_46) < String(fs_47) -> ft_4 ),
( sink(fs_27) )
 => ft_6
```

Figura 5.2: Sin Normalización de constraints

de los constraints de subtipos y los constraints de inclusión sin considerar la regla implementada con GIxTrees; el resultado es similar al anterior pero se resolvieron transitividades y otros detalles. Por último, en la Figura 5.5 presentamos el tipo resultante de normalizar todos los constraints; se puede observar la eliminación de todos los constraints intermedios.

Paso 3: Carga de la gramática.

La gramática a analizar puede cargarse desde un archivo de texto. La gramática es la misma que la del ejemplo presentado en el Capítulo 4. El texto que especifica la gramática es la entrada del módulo de parsing de gramáticas; si la sintaxis es correcta se obtiene como resultado una gramática cuya estructura se encuentra definida en el módulo de solving y si no es correcta se muestra un mensaje indicando la línea en la cual se encuentra el error sintáctico. La gramática parseada puede visualizarse en la consola.

```
( fs_56 > fs_32 ),
( fs_31 > fs_37 ),
( fs_33 > fs_34 ),
( fs_35 > fs_36 ),
( fs_37 > fs_35++fs_33 ),
( fs_38 > fs_39 ),
( fs_40 > fs_41 ),
( fs_36 > fs_40++fs_38 ),
( fs_42 > fs_43 ),
( fs_44 > fs_45 ),
( fs_41 > fs_44++fs_42 ),
( fs_46 > fs_47 ),
( fs_48 > fs_49 ),
( fs_45 > fs_48++fs_46 ),
( fs_49 > "SELECT * FROM users" ),
( fs_47 > " WHERE email = " ),
( fs_43 > fs_50 ),
( fs_51 > fs_52 ),
( fs_50 > D_SIGMA++fs_50 ),
( fs_50 > epsilon ),
( fs_52 > "UserName" ),
( fs_39 > " AND password = " ),
( fs_34 > fs_53 ),
( fs_54 > fs_55 ),
( fs_53 > D_SIGMA++fs_53 ),
( fs_53 > epsilon ),
( fs_55 > "Password" ),
( sink(fs_31) )
 => String(fs_56)
```

Figura 5.3: Normalización de constraints de subtipos únicamente

```
( fs_56 > fs_32 ),
( fs_31 > fs_53 ),
( fs_55 > fs_36 ),
( fs_54 > fs_41 ),
( fs_53 > fs_54++fs_55 ),
( fs_38 > " AND password = " ),
( fs_52 > fs_45 ),
( fs_51 > fs_50 ),
( fs_40 > fs_51++fs_52 ),
( fs_47 > " WHERE email = " ),
( fs_49 > "SELECT * FROM users" ),
( fs_50 > fs_48++fs_46 ),
( fs_48 > fs_49 ),
( fs_46 > fs_47 ),
( fs_45 > fs_43 ),
( fs_44 > fs_42 ),
( fs_43 > epsilon ),
( fs_43 > D_SIGMA++fs_43 ),
( fs_42 > "UserName" ),
( fs_41 > fs_39++fs_37 ),
( fs_39 > fs_40 ),
( fs_37 > fs_38 ),
( fs_36 > fs_34 ),
( fs_35 > fs_33 ),
( fs_34 > epsilon ),
( fs_34 > D_SIGMA++fs_34 ),
( fs_33 > "Password" ),
( sink(fs_31) )
 => String(fs_56)
```

Figura 5.4: Normalización sin la regla tratada con GIxTrees

```
( fs_32 > fs_28 ),
( fs_27 > "SELECT * FROM users"++" WHERE email = "++fs_30++
          " AND password = "++fs_29),
( fs_30 > epsilon ),
( fs_30 > D_SIGMA++fs_30 ),
( fs_29 > epsilon ),
( fs_29 > D_SIGMA++fs_29 ),
( sink(fs_27) )
 => String(fs_32)
```

Figura 5.5: Normalización completa

Paso 4: Solving.

Una vez inferido el tipo de la expresión y cargada la gramática es posible realizar el solving. La salida del algoritmo es la lista de posibles asignaciones de no terminales de la gramática a variables de lenguaje. En este ejemplo es:

```
[[fs_27 := S],[fs_27 := DANGEROUS]]
```

Además se muestra el mensaje

```
Existe una vulnerabilidad!!!
```

Para resumir, en este Capítulo describimos la implementación del prototipo de GBA modificado. Comenzamos brindando una descripción general de cada uno de los componentes del sistema y luego nos detuvimos en explicar una estructura de datos diseñada para lograr mayor eficiencia en la normalización de contraints. Además presentamos como ejemplo un programa representativo al problema de las vulnerabilidades para ilustrar el uso del prototipo y la participación de cada uno de los módulos implementados; y por último, mostramos resultados obtenidos de la ejecución del algoritmo a partir de dicho programa. De esta manera, explicamos cómo logramos integrar las ideas teóricas mencionadas a lo largo del trabajo, en un entorno operativo.

Capítulo 6

Prototipo

En este capítulo describiremos los pasos necesarios para correr el prototipo y la forma de uso del mismo. En cuanto a la forma de uso, describiremos cada una de las funcionalidades provistas por el prototipo, así como la forma de escribir los parámetros necesarios para realizar el solving, o sea, los programas y las gramáticas.

6.1. Operación del prototipo

Como mencionamos anteriormente el prototipo ha sido desarrollado en Haskell, con lo cual el único requisito para correrlo es tener instalado el intérprete para Haskell, Hugs98, que se puede bajar de la página: `http://cvs.haskell.org/Hugs/pages/downloading.htm`.

Una vez instalado el intérprete, podemos correr el prototipo ejecutando el Hugs sobre el archivo Main, que se encuentra en la carpeta que contiene el código, llamada GBA. Luego que carga, se evalua la expresión ``‘‘main’’`` para entrar al ambiente de GBA.

El prototipo funciona como un shell de comandos; los comandos son identifiicados con el símbolo ":" (dos puntos). A continuación se describen los comandos permitidos – los parámetros encerrados entre `<>` son obligatorios y los encerrados entre `[]`, opcionales.

- Comandos generales:

  h, ? Ambos comandos se utilizan para visualizar la ayuda en la pantalla, la cual explica brevemente los comandos disponibles.

  q Sale del ambiente de GBA.

- Comandos para paths:

  a `<directorio>` Por defecto, la búsqueda de los archivos que se cargan (archivos de programas y de gramáticas) se realiza sobre el

directorio raíz (nosotras en esta versión lo llamamos GBA). Este comando agrega el `directorio` al path por defecto, de manera que la búsqueda de los archivos que se cargan se realice también sobre todos los subdirectorios del directorio raíz que hayan sido agregados utilizando este comando. De esta forma no necesitamos escribir todo el path cada vez que cargamos un archivo, sino que lo agregamos una vez y luego escribimos sólo su nombre. Por ejemplo, si queremos que la búsqueda se realice dentro del directorio GBA\Ejemplos, ingresamos el comando `:a Ejemplos`.

o <directorio> Remueve el `directorio` del path por defecto, si existe.

- Comandos para módulos:

  le `<archivo>` Carga el `archivo` de programa. La extensión por defecto es `.str`. Por ejemplo, si queremos cargar el archivo de programa del ejemplo en la Sección 4.2, ingresamos el comando `:le Ejemplos\ejemploCap4` (o utilizando el comando para agregar paths `:a Ejemplos`, seguido de `:le ejemploCap4`). La ejecución de este comando consiste en realizar el parsing del archivo, mantener en memoria al árbol sintáctico que representa al programa y marcar dicho archivo como el archivo de programa actual. Para indicar esto el prompt cambia a `ejemploCap4`, `No file loaded>`, lo cual indica que el archivo `ejemploCap4` es el archivo de programas actual y que aún no se ha cargado ningún archivo de gramática. El comando `:se` se usa para cambiar entre los archivos cargados.

  lg `<archivo>` Carga el `archivo` de gramática. La extensión por defecto es `.gr`. Por ejemplo, si queremos cargar el archivo de gramática del ejemplo en la Sección 4.2, ingresamos el comando `:lg Ejemplos\gejemploCap4` (o utilizando el comando para agregar paths `:a Ejemplos`, seguido de `:lg gejemploCap4`). La ejecución de este comando consiste en realizar el parsing del archivo de gramática, mantener en memoria al árbol sintáctico que representa a la gramática y marcar dicho archivo como el archivo de gramática actual. Para indicar esto el prompt cambia a `No file loaded`, `gejemploCap4>`, lo cual indica que el archivo `gejemploCap4` es el archivo de gramática actual y que aún no se ha cargado ningún archivo de programa. El comando `:sg` se utiliza para cambiar entre los archivos cargados.

  re `[archivo]` Vuelve a cargar el `archivo` de programa; en el caso de no ingresar el parámetro, trabaja sobre el archivo actual. Luego de la recarga `archivo` pasa a ser el archivo actual. Si `archivo` no se encuentra cargado, informa tal situación.

rg `[archivo]` Vuelve a cargar el `archivo` de gramática; en el caso de no ingresar el parámetro, trabaja sobre el archivo actual. Luego de la recarga `archivo` pasa a ser el archivo actual. Si `archivo` no se encuentra cargado, informa tal situación.

de `[archivo]` Borra el `archivo` de programa; en el caso de no ingresar el parámetro, borra el archivo actual. Si el archivo de programa borrado coincide con el archivo actual, deja como actual el último archivo cargado; en caso de que el archivo borrado sea el último cargado el prompt lo indica con `No file loaded`.

dg `[archivo]` Borra el `archivo` de gramática; en el caso de no ingresar el parámetro, borra el archivo actual. Si el archivo de gramática borrado coincide con el archivo actual, deja como actual el último archivo de gramática cargado; en caso de que el archivo borrado sea el último cargado el prompt lo indica con `No file loaded`.

ce Borra todos los archivos de programas que se encuentren cargados.

cg Borra todos los archivos de gramática que se encuentren cargados.

c Borra todos los archivos que se encuentren cargados, tanto de programa como de gramática.

se `<archivo>` Cambia el archivo de programa actual por el `archivo` de programa dado, si existe.

sg `<archivo>` Cambia el archivo de gramática actual por el `archivo` de gramática dado, si existe.

f Muestra los nombres de todos los archivos cargados, tanto de programa como de gramática.

- Comandos de aplicación: Estos comandos realizan acciones sobre alguno de los archivos cargados o sobre el archivo actual si no se indican parámetros.

  e `[archivo]` Muestra el programa del `archivo` de programa dado.

  g `[archivo]` Muestra la gramática del `archivo` de gramática dado.

  t `[archivo]` Infiere el tipo del `archivo` de programa dado.

  s `[archivo de programa [archivo de gramática]]` Realiza el solving con el tipo inferido para el `archivo` de programa y la gramática del `archivo` de gramática.

6.2. Gramáticas concretas

A modo de guía para escribir los archivos de programa y de gramática necesarios para realizar el solving utilizando el prototipo, presentamos la

Expresiones

```
<exp>       ::= <munchers> | <bexp>
<bexp>      ::= <sexp> | <sexp> <rop> <sexp>          #(infijo no asociativo)
<sexp>      ::= <fterm> | <sexp> <sop> <fexp>         #(infijo asociativo a izquierda)
<fexp>      ::= <aexp> | <fexp> @ <aexp>              #(aplicación de funciones)
<aexp>      ::= <varid>                               #(variable)
                | ( <exp> )                           #(expresión entre paréntesis)
                | <liter>
<munchers>  ::= rec <varid>(<varid>) → <exp>          #(abstracción lambda)
                | let <decls> in <exp>                #(expresión let)
                | prim <primdecls> in <exp>           #(expresión prim)
                | if <exp> then <exp> else <exp>      #(condicional)
<liter>     ::= <string>
<rop>       ::= ==
<sop>       ::= ++
<decls>     ::= <decl> | <decl>; <decls>
<decl>      ::= <varid> = <exp>
<primdecls> ::= <primdecl> | <primdecl>; <primdecls>
<primdecl>  ::= <varid> :: <sigmaexp>
```

Tipos

```
<sigmaexp>      ::= <typeexp> | <constraints> ⇒ <typeexp>
<typeexp>       ::= <typefact> → <typeexp> | <typefact>
<typefact>      ::= <typevar>
                    | String(<langvar>)
                    | (<typeexp>)
<typevar>       ::= T_<num>
<langvar>       ::= S_<num>
<constraints>   ::= <constraint> | <constraint>, <constraints>
<constraint>    ::= <tvar> < <tvar>                   #(constraint de subtipo)
                    | <lvar> > <indexes>              #(constraint de inclusión)
                    | sink(<lvar>)                    #(constraint sink)
                    | true                            #(constraint true)
<indexes>       ::= epsilon | <nindexes>
<nindexes>      ::= <index> | <index> ++ <nindexes>
<index>         ::= <lvar> | <symbol> | <dSymbol> | sigma | d_sigma
<symbol>        ::= <string>
<dSymbol>       ::= D_<string>
<num>           ::= <dig> | <dig> <num>
```

Analizador léxico

```
<varid>       ::= (small {small | large | digit | '}) ⟨< reservedid >⟩
<reservedid>  ::= let | in | rec
                  | if | then | else | String
                  | epsilon | sigma | d_sigma
<string>      ::= ″[^″]″          #indica que es una cadena de caracteres
                                  distintos de ″ encerrados entre ″
```

Figura 6.1: Gramática concreta para el lenguaje

```
<CFG>              ::= <MProductions>
                       DangerousNTs: <Nonterminals>
<Nonterminals>     ::= <Nonterminal>
                       | <Nonterminal>, <Nonterminals>
<Nonterminal>      ::= <varid>
<Terminal>         ::= <string>
<DTerminal>        ::= D_<string>
<MProductions>     ::= <MProduction>
                       | <MProduction>, <MProductions>
<MProduction>      ::= <Nonterminal> → <MProductionRHS>
<MProductionRHS>   ::= <prodRHS>
                       | <prodRHS> <bar> <MProductionRHS>
<prodRHS>          ::= <GrammarSymbol>
                       | <GrammarSymbol> <prodRHS>
<GrammarSymbol>    ::= <Nonterminal>
                       | <Terminal>
                       | <DTerminal>
                       | SIGMA
                       | D_SIGMA
```

Analizador léxico

```
<varid>      ::= ({< small > | < large > | < digit > | ""})⟨< reservedid >⟩
<string>     ::= "[^"]"
<bar>        ::= |
reservedid   ::= DangerousNTs | SIGMA | D_SIGMA
```

Figura 6.2: Gramática concreta para la gramática

gramática concreta para el lenguaje (Figura 6.1) y la gramática concreta para las gramáticas (Figura 6.2).

En este punto el lector está en condiciones de utilizar el prototipo para probar los ejemplos existentes ubicados en el directorio GBA\Ejemplos, así como también de generar nuevos archivos de programas y de gramáticas para probar (utilizando las gramáticas concretas dadas anteriormente).

Capítulo 7

Trabajos Relacionados

En este capítulo presentaremos dos técnicas de análisis estático que, al igual que GBA, permiten la detección de vulnerabilidades en el código de una aplicación web. El propósito de este capítulo no es dar una descripción detallada de dichas técnicas sino presentarlas en forma breve con el fin de mostrar las ventajas y desventajas de las mismas con respecto a nuestra técnica.

7.1. IFA

IFA es un sistema que realiza un análisis de flujo de información para Java basado en ejecución simbólica y es el otro análisis incluído en el proyecto "Plataforma híbrida de seguridad para protección de aplicaciones web", antes mencionado. El efecto de ejecutar una expresión o comando, desde el punto de vista del 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 variable (local, estática o de instancia) se le asigna un *security label*. Un *security label* es un *security level* o un *symbolic reference*:

Sec. Level	l	::=	U	*Untainted*
		|	T	*Tainted*
Sec. Label	σ	::=	l	*Sec. level*
		|	(A, C, l)	*Symbolic reference*

Los security levels son marcas asociadas a los datos del programa que permiten diferenciar entre datos seguros e inseguros. IFA asigna el security level U a los datos seguros y T a los inseguros. Las symbolic references modelan la creación dinámica de objetos. Éstas incluyen un conjunto de symbolic reference names A, un class name C y un security level l. La C es el

nombre de la clase del objeto que reside en la heap y A tiene las referencias a este objeto.

El comportamiento de seguridad de los métodos requiere que alguna información adicional esté disponible antes y después de la ejecución simbólica para especificar adecuadamente cómo la ejecución de un constructor modifica el estado de ejecución. La información requerida antes de la ejecución simbólica refleja el pre-estado del análisis y la requerida después el post-estado del análisis. Notar que ambos estados incluyen un estado de ejecución.

De esta manera, para cada sentencia y expresión del lenguaje se define un juicio que expresa como afecta su ejecución a los estados de ejecución y análisis. Por lo tanto, existen dos juicios principales: uno para expresiones y otro para comandos. El juicio de análisis para las expresiones tiene la forma:

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

Este juicio se lee "dado el pre-estado del análisis que consiste en la clase host E, el estado de ejecución $\mathcal{M}$ y el security level del contador de programa l, la ejecución de la expresión e produce el post-estado del análisis. El post-estado consiste de un estado de ejecución $\mathcal{M}_1$ junto con el security label del valor de la expresión".

El juicio de análisis para los comandos es similar al de las expresiones con algunos datos adicionales que no vale la pena mencionar dado que como se mencionó anteriormente no estamos interesadas en dar demasiado detalle.

Para tratar los métodos recursivos definidos por el usuario y los métodos de las bibliotecas del sistema se utiliza un conjunto de tablas de seguridad de métodos. Estas tablas especifican la forma en que cada método se comporta desde el punto de vista de seguridad. Dada la información de entrada de un método, las tablas indican como afecta el código del método al estado del análisis. IFA puede verse como un sistema cuyo objetivo es computar dichas tablas. La relación entre estas tablas y los esquemas del análisis es escencialmente que los esquemas de análisis deben computar las entradas de estas tablas. Desde un punto de vista operacional, IFA computa el conjunto de tablas de seguridad de métodos mediante iteraciones progresivas hasta alcanzar un punto fijo, excepto para las entradas correspondientes a los métodos de las bibliotecas del sistema cuyos datos se cargan desde un archivo prelude antes de que comience el análisis.

Para terminar sólo nos queda mencionar que IFA provee dos tipos de análisis llamados *branch-based* y *join-based*. Para explicar la diferencia entre ellos cosideremos el siguiente código:

```
x = 0;
if (e)
  x = secret;
else
```

```
x = public;
```

En un análisis de flujo de información estándard el security label asignado a `x` en el punto en que convergen las ramas del "then" y el "else" (a veces denominado *junction point* de la sentencia if) es T. Este enfoque conservativo se basa en la suposición obvia de que al momento de compilación no se conoce que rama será ejecutada realmente. Una situación similar se presenta para las referencias simbólicas en la siguiente sentencia:

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

La ejecución simbólica asignará a `x` en la línea 4 el security label $(\{a\}, D, \mathsf{U})$ y a `x` en la línea 6 el security label $(\{b\}, E, \mathsf{U})$ donde a, b son nuevos symbolic reference names. La pregunta que surge es qué label se le asigna a `x` en el junction point. Hay dos soluciones posibles que se corresponden con los dos tipos de análisis provistos por IFA mencionados anteriormente:

- En la solución join-based se asume la siguiente hipótesis: el comportamiento de seguridad de los métodos sobreescritos por una subclase es idéntico al de la superclase. Esta hipótesis es similar a la condición estándar impuesta sobre la mayoría de los lenguajes orientados a objetos en los que el tipo de un método redefinido no puede cambiar el del método original (por otro lado, no es posible el chequeo estático de polimorfismo de subtipos [1]). Bajo esta supocición, a `x` se le asigna el label $(\{a, b\}, C, \mathsf{U})$. Notar que esta referencia ahora es del tipo declarado para `x`, llamdo C. Además, el level U resulta de tomar el supremo de los levels de cada referencia (al igual que en el primer ejemplo anterior).

- La solución branch-based consiste en dividir todos los análisis siguientes en dos casos. A diferencia del análisis join-based, este análisis considera todos los posibles estados que se pueden dar en la ejecución. Los beneficios son que la hipótesis anterior no se requiere permitiendo de esta forma un análisis mas flexible. Además, permite conocer el camino que se tiene que seguir para explotar una vulnerabilidad. Como contrapartida hay una penalidad de tiempo obvia. Además, en algunas situaciones la solución práctica parece no ser una opción. Un ejemplo es cuando una estructura de bifurcación se anida dentro de una sentencia de iteración como un while o un for.

[1] Java 1.5 permite cambios en los tipos de los métodos redefinidos siempre y cuando los tipos del argumento sean más generales y el tipo de retorno más específico.

La forma en que IFA detecta vulnerabilidades de inyección es similar a GBA; proveyendo una manera de identificar métodos sources y sinks. En este caso un método source es aquel que devuelve algo Tainted y un método sink indica que sus parámetros son sensibles y por lo tanto no deberían ser Tainted. De esta manera si a un método sink de le pasa como parámetro un valor Tainted, IFA informa dicha situación como una vulnerabilidad.

Todo esto fue prototipado en el marco del proyecto como un plugin de Eclipse que permite tomar un código escrito en Java y analizarlo obteniendo como resultado un informe de todas las vulnerabilidades encontradas y una forma de identificarlas en dicho código.

Al nivel de desarrollo actual, la principal diferencia de IFA con respecto de GBA es que permite el análisis de programas escritos en un lenguaje de programación real, como Java, pudiendo, por lo tanto, ser utilizado para analizar cualquier aplicación web real que este escrita en dicho lenguaje y además no sólo informa la existencia de vulnerabilidades sino que brinda cierta información de contexto que permite identificar el punto en el que cada vulnerabilidad se produce. (Estas dos características podrían ser alcanzadas por GBA, pero como se dijo en la sección 4.3, requiere más desarrollo).

Sin embargo, GBA tiene el potencial de ser extremadamente más flexible, ya que los constraints generados expresan las restricciones que los strings deben satisfacer independientemente del tipo de vulnerabilidad a detectar, siendo de esta forma fácilmente parametrizable para detectar distintos tipos de vulnerabilidades de inyección. Además, IFA no permite un tratamiento de strings tan preciso como GBA pues sólo aproxima la información de strings "manchando" cualquier string que use datos provenientes de un source, sin discriminar cómo se los usa, o si son filtrados y de qué manera. Estas características hicieron razonable que el proyecto considerase ambos enfoques.

7.2. LAPSE

Otra de las técnicas existentes de análisis estático para la detección de vulnerabilidades es la propuesta en el trabajo realizado por Livshits y Lam, [2005]. Esta técnica de análisis estático se basa en análisis de punteros para encontrar vulnerabilidades en aplicaciones Java causadas por entrada no chequeada. Los resultados del análisis estático son presentados en una interface denominada LAPSE la cual está integrada con Eclipse, un popular ambiente de desarrollo Java. Los usuarios de la herramienta pueden describir patrones de vulnerabilidades de interés en un lenguaje de consulta fácil de programar con sintaxis *Java-like*. La herramienta transforma consultas especificadas por el usuario a bytecodes Java y encuentra estáticamente todas las coincidencias. Una ventaja con respecto a nuestra técnica es que el uso de análisis a nivel de bytecode obvia la necesidad de tener acceso al código fuente. Sin embargo, una vulnerabilidad puede estar presente de diferentes

maneras en el código fuente de una aplicación y para que el análisis a nivel de bytecode detecte una vulnerabilidad deben considerarse diferentes especificaciones para la misma. Nuestra técnica, es mucho más genérica en este sentido ya que detecta la presencia de una vulnerabilidad a partir del análisis del uso de los strings a lo largo del programa para luego determinar si los mismos son admisibles por una gramática que capture la vulnerabilidad.

De manera similar a nuestra técnica, esta técnica cuenta con un conjunto de descriptores *source* y descriptores *sink*. Además, para manejar la semántica de strings de Java se introdujo un conjunto de descriptores de *derivación* que permite especificar cómo los datos se propagan entre objetos en el programa. Para encontrar violaciones de seguridad estáticamente, es necesario conocer a qué objetos se refieren estos descriptores, un problema general conocido como análisis de punteros. Un número ilimitado de objetos puede ser alocado por el programa en tiempo de ejecución, por lo que para computar una respuesta finita, el análisis de punteros aproxima estáticamente los objetos dinámicos del programa con un conjunto finito de objetos estáticos nombrados. El análisis estático está basado en un análisis de punteros Java sensitivo de contexto por Whaley y Lam [2004]. Su algoritmo usa diagramas de decisión binaria (BDDs) para representar y manipular eficientemente resultados *points-to* para muchos contextos en el programa. Ellos han desarrollado una herramienta llamada bddbddb (BDD-Based Deductive DataBase) que automáticamente traduce análisis de programas expresados en términos de Datalog (lenguaje usado en bases de datos deductivas) en implementaciones basadas en BDD altamente eficientes. Es relativamente fácil implementar el análisis de propagación de objetos tainted usando bddbddb. Los constraints de una especificación pueden ser traducidos en queries Datalog apropiadamente.

En cuanto al manejo de los strings de Java, métodos como `String.toLowerCase()` alocan objetos String que luego son retornados y considerados tainted si tal método es invocado con un string tainted. Del mismo modo que IFA, este manejo de strings aproxima la información de strings y por lo tanto no es tan preciso como GBA.

Capítulo 8

Conclusiones

Este trabajo surgió hace algunos años, por el interés de un grupo de expertos en seguridad de la empresa Core de estudiar varias técnicas de análisis de programas con el fin de utilizarlas en el desarrollo de una herramienta capaz de resolver una parte considerable del problema de seguridad que afecta a las aplicaciones web: las vulnerabilidades de inyección. La motivación para esto fué la alta frecuencia de aparición de este tipo de vulnerabilidades en el código fuente de las aplicaciones web, y la escasez de herramientas para detectarlas. Hasta ese momento el trabajo de auditoría de código fuente era completamente manual y requería un gran consumo de tiempo y concentración. Si bien el objetivo de ese proyecto no era automatizar completamente dicha tarea, contar con una herramienta que permitiera detectar un conjunto considerable de vulnerabilidades en forma semi-automática agilizaría y mejoraría el trabajo de auditoría permitiendo seguir trabajando en este área de manera de cubrir la mayor parte del problema.

En la primer parte de este trabajo nos especializamos en estudiar el problema general de seguridad y privacidad para luego concentrarnos en el estudio de las vulnerabilidades de inyección dado que era el problema a atacar. Esto para nosotras fué completamente nuevo dado que nunca habíamos estudiado nada sobre seguridad y nos resultó altamente interesante. El análisis del problema lo realizamos con la ayuda de la gente de Core y nos permitió tener una visión más clara de los objetivos para de esa manera poder empezar a buscar soluciones. En esta primer etapa, también analizamos las herramientas existentes y su funcionamiento.

Una vez entendido el problema y los objetivos, comenzamos a considerar varias técnicas, tanto de análisis estático como dinámico. Encontramos que la principal diferencia en la aplicación de dichas técnicas en seguridad consiste en que los enfoques dinámicos proveen protección para el software en funcionamiento en un entorno operativo real, mientras que los enfoques estáticos permiten detectar vulnerabilidades potenciales antes de que el software esté funcionando. Decidimos enfocarnos en técnicas basadas en análisis

estático, cuya principal ventaja con respecto al análisis dinámico es el reuso; verificar la ausencia de un tipo de error en el código estáticamente permite reducir el riesgo de varios errores en tiempo de ejecución, evitando de esta manera analizar múltiples escenarios de ejecución de una aplicación. De todos modos, en el transcurso del proyecto tuvimos que enfrentarnos con ciertas limitaciones propias del análisis estático ya que en muchas situaciones es difícil ofrecer satisfactoriamente una aproximación del estado del programa en tiempo de ejecución.

Entre las técnicas candidatas que fueron analizadas en el proyecto se seleccionaron dos, la que describimos en el presente trabajo y una de análisis de flujo que también fue desarrollada y prototipada como parte del proyecto marco, denominada IFA y presentada en la sección 7.1. Las ventajas de nuestra técnica con respecto a la otra son la extremada precisión y flexibilidad que provee, ya que los constraints generados expresan las restricciones que los strings deben satisfacer independientemente del tipo de vulnerabilidad a detectar. Sin embargo, GBA es una técnica mucho más compleja y requiere mucho más tiempo de estudio para poder convertirla en una técnica que analice programas escritos en un lenguaje de producción real, como ser Java, tal como lo hace IFA en la actualidad.

Luego de haber estudiado en forma analítica la técnica de GBA y haber entendido completamente su funcionamiento, nos concentramos en pensar las modificaciones necesarias para poder aplicarla a la detección de vulnerabilidades de inyección. En esta línea surgieron muchas ideas, pero nos concentramos en las modificaciones mínimas indispensables que permitieran resolver el problema, dejando el resto como trabajo a futuro por una cuestión de tiempo. Consideramos que era mejor tener una herramienta pequeña funcionando y testeada que tener una gran cantidad de ideas sin concretar.

Finalmente, decidimos volcar todas estas ideas en un prototipo desarrollado en Haskell y de esta forma integrarlas en un entorno operativo. La selección del lenguaje de programación la realizamos con el equipo de Core basándonos en el alto poder expresivo que tienen los lenguajes funcionales y la inmediatez para codificar dichas ideas. Como es usual, la etapa de codificación involucró una etapa de diseño de las estructuras de datos necesarias para llevar a cabo la implementación eficiente del prototipo, una etapa de implementación y una última etapa de pruebas. Debido a la ineficiencia del algoritmo de normalización de constraints como estaba especificado, tuvimos que diseñar entre otras cosas una estructura de datos que nos permitiera mejorarla.

Consideramos de gran importancia el desarrollo del prototipo ya que en la etapa de diseño e implementación del mismo fueron surgiendo nuevas inquietudes y cuestionamientos acerca de la técnica, que no se nos presentaron al momento de su estudio y que nos permitieron comprenderla en profundidad y determinar posibles mejoras. Los resultados obtenidos con nuestro prototipo fueron altamente exitosos. Escribimos varios programas con cier-

tas vulnerabilidades y el prototipo efectivamente las detectaba tal como se esperaba.

No sólo los resultados obtenidos en el estudio de IFA y GBA condujeron al éxito del proyecto. Queremos destacar que tanto la empresa Core como el grupo de estudiantes que participamos en el mismo coincidimos en que fue un gran desafío la experiencia de realizar investigación conjunta entre la universidad y la empresa. Esta interacción es poco frecuente y sin embargo se pueden obtener resultados muy valiosos para el crecimiento de la industria.

Creemos que este trabajo es un gran aporte como un primer paso a la solución del problema de seguridad de las aplicaciones web y que merece seguir siendo estudiado.

Bibliografía

[Abadi and Cardelli, 1996] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[Agat, 1997] Johan Agat. Types for register allocation. In *IFL'97*, volume 1467 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Anupam and Mayer, 1998] Vinod Anupam and Alain Mayer. Security of web browser scripting languages: Vulnerabilities, attacks, and remedies. In *Proceedings of the 7th USENIX Security Symposium*, pages 187–199. USENIX Association, January 1998.

[Atkinson and Morrison, 1988] Malcolm P. Atkinson and Ronald Morrison. Types, bindings and parameters in a persistent environment. In Malcolm P. Atkinson, Peter Buneman, and Ronald Morrison, editors, *Data Types and Persistence*. Springer-Verlag, 1988.

[Atkinson *et al.*, 1983] Malcolm P. Atkinson, Philip J. Bailey, Kenneth J. Chisholm, W. Paul Cockshott, and Ronald Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, 1983.

[Breazu-Tannen *et al.*, 1991] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[Burstall *et al.*, 1980] Rod M. Burstall, David B. MacQueen, and Donald T. Sannella. Hope: An experimental applicative language. In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM Press, August 1980.

[Christensen *et al.*, 2003] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In Radhia Cousot, editor, *Proceedings of International Static Analysis Symposium, SAS'03*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, June 2003.

[Coquand and Huet, 1988] Thierry Coquand and Gèrard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.

[Cranor *et al.*, 2002] Lorrie Faith Cranor, Manjula Arjula, and Praveen Guduru. Use of a P3P user agent by early adopters. In *WPES '02: Proceedings of the 2002 ACM Workshop on Privacy in the Electronic Society*, pages 1–10. ACM Press, 2002.

[Damas and Milner, 1982] Luis Damas and Robin Milner. Principal type-schemes for functional languages. In *Proceedings of the 9th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.

[Danvy, 1998] Olivier Danvy. Type-directed partial evaluation. In Torben Æ. Mogensen John Hatcliff and Peter Thiemann, editors, *Partial Evaluation - Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 367–411. Springer-Verlag, June 1998.

[Dearle, 1988] Alan Dearle. *On the Construction of Persistent Programming Environments*. PhD thesis, University of St. Andrews, 1988.

[Earley, 1970] Jey Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[Freund and Mitchell, 1998] Stephen Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. In *ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 310–328. ACM Press, 1998.

[Frühwirth, 1995] Thom Frühwirth. Constraint handling rules. In Andreas Podelski, editor, *Constraint Programming: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Futoransky and Waissbein, 2005] Ariel Futoransky and Ariel Waissbein. Enforcing privacy in web applications. In *Third Annual Conference on Privacy, Security and Trust*, October 2005.

[Gaster and Jones, 1996] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical report, Department of Computer Science, University of Nottingham, November 1996. Technical report NOTTCS-TR-96-3.

[Girard, 1989] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press, 1989.

[Hannan and Hicks, 1988] John J. Hannan and Patrick Hicks. Higher-order arity raising. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 27–38. ACM Press, September 1988.

[Henglein and Rehof, 1997] Fritz Henglein and Jakob Rehof. The complexity of subtype entailment for simple types. In *Proceedings, Twelfth Annual IEEE Symposium on Logic in Computer Science*, pages 352–361, July 1997.

[Herzberg and others, 2005] Amir Herzberg et al. Trustbar, May 2005. URL: `http://trustbar.mozdev.org`.

[Hindley, 1969] Robin Hindley. The principal type-scheme of an object in combinatory. *Transactions of the American Mathemetical Society*, 146:26–60, December 1969.

[Hoang *et al.*, 1993] My Hoang, John Mitchell, and Ramesh Viswanathan. Standard MLNJ weak polymorphism and imperative constructs. In Moshe Vardi, editor, *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 15–25. IEEE Computer Society Press, June 1993. Journal version appeared as [Mitchell and Viswanathan, 1996].

[Hopcroft, 1969] John E. Hopcroft. On the equivalence and containment problems for context-free languages. *Math. Systems Theory*, 3(2):119–124, 1969.

[Huang *et al.*, 2004] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yeng Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[Hudak *et al.*, 1992] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[Hughes *et al.*, 1996] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1996.

[Hughes, 1995] John Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96, May 1995.

[Hughes, 1996a] John Hughes. An introduction to program specialisation by type inference. In *Functional Programming*. Published electronically, July 1996.

[Hughes, 1996b] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Selected papers of the International Seminar "Partial Evaluation"*, volume 1110 of *Lecture Notes in Computer Science*, pages 183–215. Springer-Verlag, February 1996.

[Hutton and Meijer, 1996] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-1, Department of Computer Science, University of Nottingham, 1996.

[Igarashi and Kobayashi, 2001] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. *ACM SIGPLAN Notices*, 36(3):128–141, 2001.

[Jim, 1996] Trevor Jim. What are principal typings and what are they good for? In *Proceedings of POPL96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53. ACM Press, January 1996.

[Jones, 1994a] Mark P. Jones. *Qualified Types: Theory and Practice*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Jones, 1994b] Mark P. Jones. Simplifying and improving qualified types, June 1994. Research Report YALEU/DCS/RR-1040, Yale University.

[Jones, 1996] Mark P. Jones. Overloading and higher order polymorphism. In Erik Meijer and John Jeuring, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1996.

[Jouvelot and Gifford, 1991] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1991.

[Knabe, 1995] Frederick Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, USA, December 1995.

[Livshits and Lam, 2005] V. Livshits and M. Lam. Finding security vulnerabilities in java applications with static analysis, 2005.

[Maor and Shulman, 2004] Ofer Maor and Amichai Shulman. SQL injection signatures evasion. Technical report, Imperva Application Defense Center, 2004.

[Martínez López and Hughes, 2002] Pablo E. Martínez López and John Hughes. Principal type specialisation. In Wei-Ngan Chin, editor, *Proceedings of the 2002 ACM SIGPLAN Asian Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105. ACM Press, September 2002.

[Martínez López, 2005] Pablo E. Martínez López. *The Notion of Principality in Type Specialisation.* PhD thesis, University of Buenos Aires, November 2005.

[Milner *et al.*, 1990] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* August 1990.

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

[Mitchell and Plotkin, 1988] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[Mitchell and Viswanathan, 1996] John Mitchell and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. *Information and Computation*, 127(2):102–116, June 1996.

[Mitchell, 1991] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, July 1991.

[Mohri and Nederhof, 2001] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 156–163. Kluwer Academic Publishers, Dordrecht, 2001.

[Morrison *et al.*, 1993] Ronald Morrison, Richard C.H. Connor, Quintin I. Cutts, Graham Kirby, and Dave Stemple. Mechanisms for controlling evolution in persistent object systems. *Journal of Microprocessors and Microprogramming*, 17(3):173–181, 1993.

[Mossin, 1996] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, December 1996.

[Nordlander, 1999] Johan Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Chalmers University of Technology, 1999.

[Nordlander, 2000] Johan Nordlander. Polymorphic subtyping in O'Haskell. In *Proceedings the APPSEM Workshop on Subtyping and Dependent Types in Programming*, 2000.

[Nordström *et al.*, 1990] Bengt Nordström, Ken Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[O'Callahan, 1998] Robert O'Callahan. *Scalable Program Analysis and Understanding Based on Type Inference.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh PA, January 1998.

[Odersky *et al.*, 1995] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proceedings of ACM Conference on Functional Programming and Computer Architecture.* ACM Press, 1995.

[Odersky *et al.*, 1999] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[OWASP Organization, 2003] OWASP Organization. The ten most critical web application security vulnerabilities. Technical report, OWASP, January 2003. URL: `http://www.owasp.org`.

[Palsberg *et al.*, 1997] Jens Palsberg, Mitchell Wand, and Patrick OKeefe. Type inference with non-structural subtyping. *Formal Aspects of Computer Science*, 9:49–67, 1997.

[Pietraszek and Berghe, 2005] Tadeuz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, pages 124–145, 2005.

[Pottier, 2000] Francois Pottier. A 3-part type inference engine. In Gert Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP '00)*, volume 1792 of *Lecture Notes in Computer Science*, pages 320–335. Springer-Verlag, March/April 2000.

[Rèmy, 1989] Didier Rèmy. Type checking records and variants in a natural extension of ML. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–88. ACM Press, January 1989.

[Reynolds, 1974] John C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[Reynolds, 1983] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A.Mason, editor, *Information Processing83. Proceedings of the IFIP 9th World Computer Congress*, pages 513–523, 1983.

[Rezgui *et al.*, 2002] Abdelmounaam Rezgui, Mourad Ouzzani, Athman Bouguettaya, and Brahim Medjahed. Preserving privacy in web services. In *The 4th international workshop on Web information and data management*, pages 56–62. ACM Press, 2002.

[Romanenko, 1990] Sergei Romanenko. Arity raising and its use in program specialisation. In Neil D. Jones, editor, *Proceedings of 3rd European Symposium on Programming (ESOP'90)*, volume 432 of *Lecture Notes in Computer Science*, pages 341–360. Springer-Verlag, May 1990.

[Sandhu, 2003] Ravi S. Sandhu. Good-enough security: Toward a pragmatic business-driven discipline. *IEEE Internet Computing*, 7(1), 2003.

[Shields and Meijer, 2001] Mark Shields and Erik Meijer. Type-indexed rows. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*, pages 261–275. Springer-Verlag, January 2001.

[Symantec, 2005] Symantec. Symantec internet security threat report: Vol. VII. Technical report, Symantec Inc., March 2005.

[Syme, 1999] Don Syme. Proving Java type soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer-Verlag, 1999.

[Tabuchi *et al.*, 2003] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. In Gilles Barthe and Peter Thiemann, editors, *Proceedings of Workshop on Types in Programming (TIP02)*, volume 75 of *ENTCS*, pages 1–19, 2003.

[Talpin and Jouvelot, 1992] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1992.

[Thiemann, 1999] Peter Thiemann. Higher-order code splicing. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 243–257. Springer-Verlag, March 1999.

[Thiemann, 2005] Peter Thiemann. Grammar-based analysis of string expressions. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI'05)*, pages 59–70. ACM Press, 2005.

[Tiuryn and Wand, 1993] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *CAAP '93: 18th Colloquium on Trees in Algebra and Programming*, July 1993.

[Turner, 1986] David Turner. An overview of Miranda. *SIGPLAN Not.*, 21(12):158–166, 1986.

[Volpano and Smith, 1997] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.

[Volpano *et al.*, 1996] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[von Oheimb and Nipkow, 1999] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 119–156. Springer-Verlag, 1999.

[Wadler, 1995] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer-Verlag, 1995.

[Wand, 1987] Mitchell Wand. Complete type inference for simple objects. In David Gries, editor, *Proceedings 2nd IEEE Symposium on Logic in Computer Science*, pages 37–44. IEEE Computer Society Press, June 1987.

[Wand, 1991] Mitchell Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93:1–115, 1991. Preliminary version appeared in Proc. 4th IEEE Symposium on Logic in Computer Science (1989), 92-97.

[Wand, 1994] Mitchell Wand. Type inference for objects with instance variables and inheritance. In Carl Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 97–120. MIT Press, 1994. Originally appeared as Northeastern University College of Computer Science Technical Report NU-CCS-89-2, February, 1989.

[Wand, 1997] Mitchell Wand. Types in compilation: Scenes from an invited lecture. In *Workshop on Types in Compilation (invited talk), held in conjunction with ICFP97*, 1997.

[Whaley and Lam, 2004] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams, 2004.

[Wright and Felleisen, 1994] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Wright, 1992] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *Proceedings of ESOP'92, 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, February 1992.

[Wright, 1995] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[Xie and Aiken, 2006] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languajes. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, 2006.

[Zeller Jr., 2005] Tom Zeller Jr. Black market in credit cards thrives on web. *New York Times (late edition)*, June 21 2005.