Análisis Estático de Flujo de la Información para Bytecode

Renato Meneghini

rmeneghini@gmail.com
Departamento de Computación
Facultad de Ciencias Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Abstract. Este trabajo presenta una herramienta que permite garantizar la confidencialidad de la información manipulada por un programa Bytecode. El método analiza el flujo de información en el programa Bytecode utilizando análisis de dependencias. El análisis del flujo de información se divide en dos fases. La primera es la determinación de relaciones de información de dependencias entre los datos manipulados y, la segunda fase, es la verificación de la seguridad basada en las clases de seguridad definidas por el usuario. Si la verificación no falla entonces el código es seguro (no contiene flujos de información no permitidos). En el caso que falle entonces se puede determinar cuál es el flujo de información que viola la política de seguridad definida por el usuario.

Palabras Clave: Confidencialidad, Seguridad, Infomatition-Flow, Análisis de Dependencias.

1 Introducción

La industria del software y en particular los desarrolladores requieren herramientas cada vez más poderosas, que provean mayor asistencia, en forma automática, para detectar una clase cada vez más amplia de posibles errores. Esta detección debe realizarse, tanto como sea posible, en etapas tempranas del proceso de desarrollo.

Un problema con un alto impacto en el desarrollo del software, reside en el tiempo (en muchos casos considerable) que pierden los programadores en la depuración de sus programas. Se calcula que entre el veinticinco y el cincuenta por ciento del costo y tiempo destinado al desarrollo de un sistema se emplea en las actividades de prueba y depuración. Dada la intensa migración de código que impone el uso masivo de redes de computadoras actual, uso que se incrementa con el avance hacia la sociedad informatizada, uno de los problemas que se ha tornado capital es el de garantizar la confidencialidad de la información manipulada por los programas. Pero es muy difícil y costoso realizar el testing y la depuración tendiente a asegurarlo. Por ello, es necesario contar con herramientas automáticas que verifiquen estáticamente esta propiedad. La construcción de un prototipo podría clarificar la viabilidad, eficiencia y eficacia del uso de análisis estático para la verificación de confidencialidad de la información. Este problema no es específico de un lenguaje, sin embargo este trabajo se orientará en la búsqueda de garantizar de la confidencialidad para programas escritos en Bytecode Java.

Bytecode, es el lenguaje de la máquina virtual de Java (JVML). Los programas escritos en este lenguaje se pueden cargar en la red a un host remoto, como los applets y los agentes móviles, también puede interactuar con los recursos e instalaciones del host. Si los programas acceden a datos confidenciales del usuario y se comunican a través de la red, la información privada podría estar siendo liberada. Los hosts tienen la opción de proteger la información confidencial mediante el uso de mecanismos de control de acceso. Sin embargo, esto afecta la función de los programas, ya que aquellos de utilidad general necesitan datos de acceso al host para llevar a cabo sus tareas. Además restringir el acceso no garantiza que un usuario con permisos sobre el sistema no libere más información de la permitida.

Para abordar el problema de garantizar la confidencialidad de los datos, se propone la aplicación de técnicas de análisis de "flujo de información" [1]. Mediante el análisis del flujo de la información a través del programa, los datos pueden ser protegidos de fugas no permitidas hacia canales públicos. Sin embargo, la mayoría de los análisis convencionales se centra principalmente en los programas escritos en lenguajes de programación de alto nivel y generalmente lo realizan utilizando sistemas de tipos. Esto resulta insuficiente para hacer frente al flujo de información en Bytecode. Además, el método de verificación con sistemas de tipos, si bien es estático, en muchos casos es demasiado conservativo.

Este trabajo presenta una herramienta que permite garantizar la confidencialidad de la información manipulada por un programa Bytecode. El método analiza el flujo de información en el programa Bytecode utilizando análisis de dependencias. El análisis del flujo de información se divide en dos fases. La primera es la determinación de relaciones de información de dependencias entre los datos manipulados y, la segunda fase, es la verificación de la seguridad basada en las clases de seguridad definidas por el usuario. Si la verificación no falla entonces el código es seguro (no contiene flujos de información no permitidos). En el caso que falle entonces se puede determinar cuál es el flujo de información que viola la política de seguridad definida por el usuario.

El eje principal para el desarrollo de este trabajo se enfoca en presentar el diseño e implementación de una herramienta que verifica la confidencialidad de los datos en aplicaciones Java Bytecode. Esta herramienta utiliza análisis de dependencia de datos para detectar posibles violaciones a la política de confidencialidad. Cabe destacar, que el aporte de este trabajo no solo es la herramienta, sino que tambien se extiende el análisis para contemplar manejo de excepciones y la creación dinámica de objetos. Dichos aspectos permiten que la técnica contemple un gran subconjunto de programas Java Bytecode. Si bien la herramienta contempla un subconjunto extenso (y representativo) de Java Bytecode, no incluye threads.

Para poder desarrollar este trabajo, y por ende, implementar la herramienta se estudiaron contenidos y técnicas específicas del área de flujo de la información (Information-Flow) y del área de análisis estático de programas.

2 Flujo de la Información

El control del flujo de la información (IFC por Information-Flow *Control*) es una técnica importante para descubrir filtraciones de información de un software que comprometa la seguridad. IFC puede ser utilizado para garantizar dos propiedades fundamentales:

- **Confidencialidad**, los datos secretos (confidenciales) no pueden seducirse de los datos públicos.
- **Integridad**, los cómputos críticos no se pueden manipular del exterior.

IFC analiza el programa asignando y propagando niveles de la seguridad a las variables y a las expresiones, garantizando que cualquier escape potencial de seguridad será encontrado.

Las técnicas para realizar IFC se pueden dividir en dos grandes categorías: técnicas dinámicas y técnicas estáticas. Las técnicas dinámicas detectan las violaciones reales en el momento en que se producen (en tiempo de ejecución). Mientras que las técnicas estáticas determinan las violaciones de seguridad antes de la ejecución (en tiempo de compilación).

Las técnicas estáticas rechazan algunos programas seguros. Es decir, aquellos programas que no se pueden determinar en tiempo de compilación si son o no seguros son rechazados. Pero, si bien son más efectivas, las técnicas dinámicas tienen un impacto sobre la performance del código. Estas técnicas deben implementar algún mecanismo que permita mantener información, mientras se ejecuta el programa, sobre los niveles de seguridad de los valores y las instrucciones; además, deben implementar algún mecanismo que permita volver a un estado consistente después de detener la ejecución del programa si se viola la política de seguridad. Por ejemplo, por medio de una excepción.

Las técnicas estáticas se han convertido en el mecanismo de aplicación primario para las políticas del flujo de información. Entre estas técnicas se encuentran aquellas que utilizan sistemas de tipos.

El lenguaje basado en IFC utiliza el código fuente del programa solamente para descubrir los escapes de la seguridad. Esto tiene la ventaja enorme que puede explotar una larga historia de la investigación sobre análisis del programa, y descubrirá cualquier escape de la seguridad causado por el software.

Uno de los desafíos en el área de seguridad en informática es definir políticas de confidencialidad y desarrollar mecanismos para garantizar que el software las satisface.

El concepto de flujo de información *segura* es formalizado en términos de "*no interferencia*", esta es una política de alto nivel de seguridad que garantiza la absorción de información ilícita a través del programa en ejecución. Es decir, no se revela ninguna información total o parcialmente a los usuarios desautorizados.

La información fluye de x a y cuando y puede observar cambios en su entorno que dependen del valor de x. La forma más directa en la cual la información puede fluir dentro de un programa es a

través de una asignación, esto es lo que se denomina flujo explícito. Por ejemplo, en la sentencia x = *y*, hay un flujo de información de y a x ya que uno puede conocer el valor de y mirando el valor de x.

Otro flujo, explícito, con pérdida de información, p encontrarse en $z = x + y$. Aquí la información fluye tanto de *x* como de y hacia z. Alguna información se pierde debido a la operación aritmética. Todas las expresiones de asignación $x = expresión$, incluyendo el pasaje de parámetros y valores de retorno, producen un flujo de información de todos los operandos en *expresión* hacia *x*.

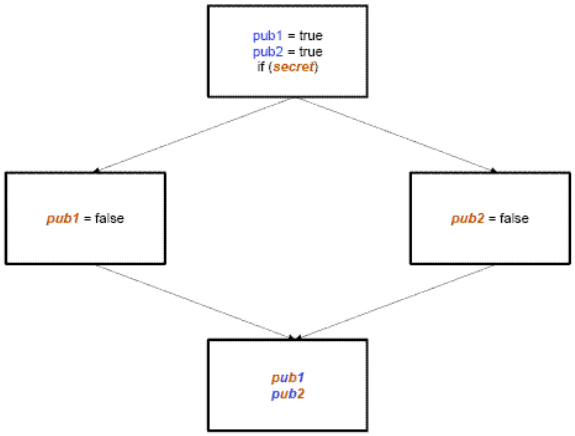


Fig. 1. Flujo de información implícito de secret a pub1 y pub2

La información también puede ser propagada a través del control de flujo de un programa. Por ejemplo, en la Fig. 1, no hay una asignación directa de *secret* a cualquiera de las otras dos variables, pero aún al final de la ejecución del fragmento de código, se puede inferir cual era el valor de *secret*. La información puede fluir implícitamente en dos caminos. Siguiendo la rama, se puede inferir cual era la variable de control. En la Fig. 1, luego de observar que el valor de *pub1* ha cambiado, se puede inferir que *secret* es verdadero.

Al observar que el valor de *pub2* no ha cambiado cuando se vuelve a unir el control de flujo, se puede inferir que *secret* es verdadero. En este ejemplo, observando solo una de las variables *pub1* o *pub2* debería ser suficiente para inferir el estado de *secret*, aún cuando cada una de las dos alternativas de control de flujo modifique sólo una de las variables.

Este es exactamente el problema que hace difícil controlar el flujo de información a través de medios estrictamente dinámicos, por ejemplo, siguiendo solo el camino a través del programa como se ha explicado. Para inhibir los flujos que resulten de asignaciones en ramas alternativas, se necesita considerar todas las ramas alternativas simultáneamente.

2.1 Flujo de Información para Bytecode

La diferencia con el flujo de información para lenguajes de alto nivel esta dado en que los lenguajes de bajo nivel presentan las siguientes características:

• Los tipos de los valores almacenados en una misma variable (o registro) pueden cambiar durante la ejecución del programa.
• Hay instrucciones de bifurcación no estructuradas (como por ejemplo la instrucción goto).

A continuación se presentan algunos ejemplos de cómo se puede dar el flujo de información en bytecode.

Ejemplo 1: (Flujos directos) Este fragmento de programa almacena en la variable "x" de bajo nivel de seguridad el valor de la variable "y"de alto nivel de seguridad, por lo tanto se produce un filtro de información.

```
1 load yH
2 store xL
3 return
```

Ejemplo 2: (Flujo indirecto mediante asignaciones) El siguiente ejemplo demuestra como la información puede ser filtrada a través de asignaciones dentro del alcance de una instrucción de salto. El valor final de X_L (0 o 1) depende del valor inicial de Y_H

```
1 load yH
2 if 6
3 push 0
4 store xL
5 goto 8
6 push 1
7 store xL
8 return
```

Ejemplo 3: (Flujo indirecto mediante operaciones sobre el stack) El valor final de *x*L (3 or 4) depende del valor inicial de *y*H. El problema es causado por una instrucción que manipula el stack en el alcance de una instrucción *if*.

```
1 push 3
2 push 4
3 load yH
4 if 6
5 store xL
6 store xL
7 return
```

En este caso no hay reuso de variables, ya que ambas variables (x e y) mantienen el mismo nivel a lo largo de toda la ejecución (High y Low, respectivamente). Este ejemplo es un caso típico donde se

produce flujo de información no deseado, un atacante puede inferir valores de variables High a partir de variables Low, ya que se puede inferir si se cumplió la condición del if (es decir si yH, es cero o no) según el valor final de una variable Low (x). Si el valor final de x es 4 entonces no se ejecuto la primer rama del if (por lo tanto y vale 0) y si x vale 3 entonces y es distinto de cero.

En los anteriores ejemplos no se reusa ninguna de las variables locales, no ocurre lo mismo con los ejemplos que presentamos a continuación.

Ejemplo 4: (Flujo de información mediante Stack) El valor final de *x*L (3 or 4) depende del valor inicial de *y*H. El problema es causado por una instrucción aritmética que manipula el stack en el alcance de una instrucción *if*.

```
1 push 3
2 load yH
3 if 6
4 push 1
5 prim +
6 store xL
7 return
```

En este ejemplo hay reuso de la variable x porque si entra al if entonces se ejecuta el prim y en el tope del stack queda un valor High que es luego asignado a x. En caso de no entrar al if entonces x termina con un valor 3 y con nivel de seguridad Low.

3 La Herramienta

En esta sección se explica el funcionamiento de la herramienta, el cual, se puede dividir en las siguientes cuatro etapas que juntas conforman el análisis de flujo de la información :

1. Construcción del Grafo de Control de Flujo (CFG).
2. Construcción de los Pares Definición-Uso (DUPs).
3. Análisis de Dependencias.
4. Definición de las clases de Seguridad, Computación y Verificación.

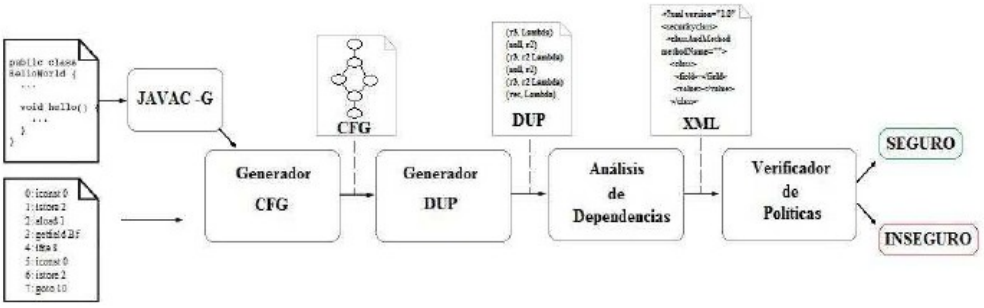


Fig. 2. Secuencias de acciones de la herramienta.

En síntesis, como se puede ver en la Fig. 2, el prototipo desarrollado toma el código y genera un CFG de cada método. Luego genera las dependencias entre los datos. En la etapa siguiente propaga las dependencias generando un XML con esta información. Por último el verificador toma las dependencias y los niveles de seguridad asignados para determinar si el programa cumple o no con la política de seguridad dada. Si se viola alguna dependencia, rechaza el programa, de lo contrario es aceptado y catalogado como Seguro.

3.1 Bytecode

El conjunto de Bytecode considerado incluye instrucciones de creación y manipulación de objetos (por ejemplo, new, putfield y getfield), manipulación del stack (por ejemplo, load, store y dup), creación de arreglos (newarray), operaciones unarias y binarias (suma, producto, etc), saltos incondicionales y condicionales (goto, if, etc).

Solo se consideran programas mono-threaded y se asume que los mismos son aceptados por el Bytecode Verifier. Es decir, entre otras cosas, no tiene errores de tipado. Además, sin perdida de expresividad, se asume que todos los métodos retornan un valor y tiene una única instrucción de retorno.

3.2 Construcción del CFG

Como primer paso, se realiza la construcción del Grafo de Control de Flujo(CFG) para cada uno de los métodos de las clases analizadas. Para ello se analiza el Bytecode (obtenido por medio de la librería BCEL, la cual recibe el .class compilado con el parámetro –g para obtener toda la información de debug) línea a línea y se “transcribe” a un vector de objetos equivalentes, para que sea más fácil su manipulación.

El CFG se define como:

$$CFG : (V,A) = \{S1, S2, S3, \ldots Sm\}$$
$$Si = \{I0, I1, \ldots In\}, Ik = (ik, cdk)$$

Cada Si representa un posible camino en el flujo de ejecución del programa. S esta formado por un conjunto de pares de la siguiente forma (instrucción, control de dependencia); el control de dependencia es el número de línea de la cual depende la primer componente del par para ser ejecutada, por lo general es una instrucción de branch, salvo que dependan del flujo principal, en dicho caso se indica con -1.

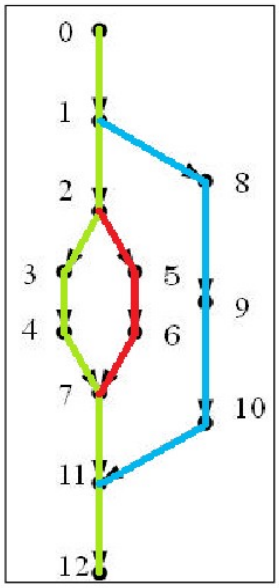
Las instrucciones de branch (o de salto) en el CFG se clasifican en 4 conjuntos: condicionales, sin condición, compuestas y por excepción.

- Branch sin condición: goto, goto_w, jsr, jsr_w, ret
- Branch condicionales: ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.
- Branch compuestas: tableswitch, lookupswitch.
- Branch por excepción: getfield, putfield, add, sub, mul, div, invoke

Los branch generados por las instrucciones que pueden dispara una excepción dependen si tienen o no un handler (manejador de excepciones) definido o no. Por lo cual, al generar el grafo es necesario contar con la información de la tabla de manejadores de excepciones de cada método. En esta tabla cada entrada contiene la siguiente información:

- StartPC: número de línea del Bytecode del método donde arranca el fragmento de código en el cual puede ocurrir una excepción.
- EndPC: número de línea del Bytecode donde termina la zona donde puede ocurrir excepciones. Esta línea no está incluida en la zona crítica.
- HandlerPC: indica el número de línea donde arranca el fragmento de código encargado tratar la excepción.
- CatchType: contiene el tipo de excepción que maneja.

Al detectar una instrucción que puede disparar una excepción se debe generar en el grafo un branch. Si la instrucción tiene un manejador definido entonces el branch es hacia la primer instrucción de dicho manejador. En caso de no tener un manejador definido el branch es hacia la instrucción de retorno del método.



CFG={S_1,S_2,S_3}
S_1={(0,-1);(1,-1);(2,1);(3,2);(4,2);(7,1);(11,-1);(12,-1)}
S_2={(2,1);(5,2);(6,2);(7,1)}
S_3={(1,-1);(8,1);(9,1),(10,1),(11,-1)}

Si se toma la cadena S_1 y la interpretamos se leería:
(1,-1) la instrucción de la línea 1 depende del flujo principal.
(8,1) la instrucción de la línea 8 depende de la instrucción de branch de la línea 1.
(9,1) la instrucción de la línea 9 depende de la instrucción de branch de la línea 1.
(10,1) la instrucción de la línea 10 depende de la instrucción de branch de la línea 1.
(11,-1) la instrucción de la línea 11 depende del flujo principal.

Como se puede ver en la Figura 9 existen 2 instrucciones de branch, las que se encuentran en la línea 1 y la de la línea 2.

Fig.3. Ejemplo de CFG

3.3 Construcción de los DUP's

A partir del Grafo de Control de Flujo se crean los DUPs (Definition-Use Pair). Los DUPs consisten en un par (v,U) donde v representa una variable (que puede ser un registro, atributo, método, etc.) y U es el conjunto de usos de dicha variable. A continuación se muestran algunas reglas para generar los DUPs. Donde, D es un conjunto de DUPS, U es un conjunto de usos temporales y S representa una piula de niveles de seguridad.

B[i] = Tpush n |Tconst null < U,D, S > < U,D, λ · S >	B[i] = ldc x < U,D, S > < U,D, x · S >
B[i] = prim op < U,D, v1 · v2 · S > < U,D, (v1 ∪ v2) · S >	B[i] = newarray < U,D, v · S > < v ∪ U,D, ∅ · S >
B[i] = pop < U,D, v · S > < U,D, S >	B[i] = anewarray x < U,D, v · S > < v ∪ U,D, ∅ · S >
B[i] = load x < U,D, v1 · v2 · S > < U,D, (v1 ∪ v2) · S >	B[i] = getfield C.f < U,D, v · S > < U,D,C.f · S >
B[i] = Tstore x < U,D, v · S > < ∅, (x, v ∪ U) ∪ D, S >	B[i] = putfield C.f < U,D, v1 · S > < ∅, (d = C.f,U ∪ v1) ∪ D, S >
B[i] = Tastore x < U,D, v1 · v2 · v3 · S > < ∅, (v1, v2 ∪ v3 ∪ U) ∪ D, S >	B[i] = invoke C.mt < U,D, S > < U,D, S >< U_0,D_0, S_0 >
B[i] = ifcond j|tableswitch|lookupswitch < U,D, v · S > < ∅, (null, v ∪ U) ∪ D, S >	B[i] = return < U,D, S >< U_0,D_0, S_0 > < ∅, (ret,U ∪ U_0) ∪ D ∪ D_0, S ><∅,∅,∅ >
B[i] = iinc x < U,D, S > < U, (x, ∅) ∪ D, S >	

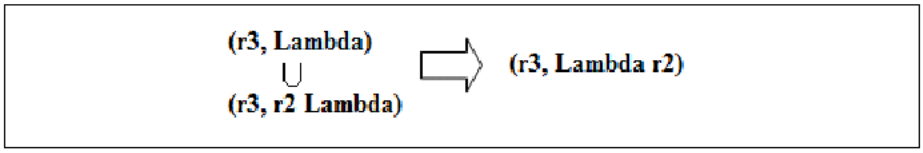
3.4 Análisis de Dependencias

En la tercera etapa se propaga la información de los usos. Incluida la propagación de información de las instrucciones que generan branchs. Esto consiste en la propagación de los usos, para ello se aplicaron las siguientes reglas sobre los DUPs obtenidos en la etapa anterior.

Unión de Branchs:

Para todo $D_i=(d_i,U_i)$, $D_j=(d_j,U_j)$

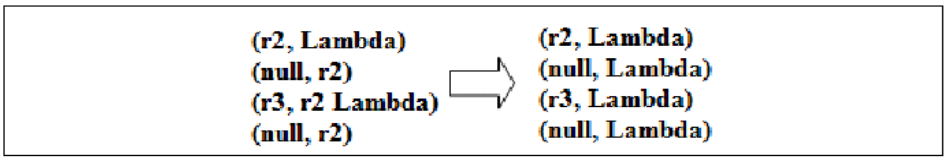
$$\{D_i\} \cup_m \{D_j\} = \begin{cases} \{d_i, U_i \cup U_j\} \; si \; d_i = d_j \\ \{D_i \cup D_j\} \; otros \; casos \end{cases}$$



Transferencia de información:

$D_i=(d_i,U_i)$, $D_j=(d_j,U_j)$ $i < j$

Si $d_i \in U_j$ y $i < j$ entonces $D_i \cup D_j = (d_j, U_i \cup U_j)$



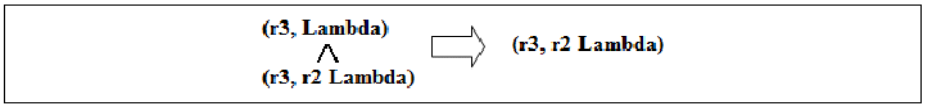
Operaciones con constantes:

$U_i \cup \lambda = U_i$

Alcance

Para todo $D_i=(d_i,U_i)$, $D_j=(d_j,U_j)$ $i < j$

$$\{D_i\} \curlywedge_m \{D_j\} = \begin{cases} \{D_j\} \; si \; d_i = d_j \\ \{\} \; otros \; casos \end{cases}$$



Aplicando estas reglas se obtienen el conjunto mínimo de dependencias necesarias para representar todas las dependencias entre los objetos manipulados por un programa.

3.5 Clases de Seguridad, computación y Verificación

Esta cuarta y última etapa consiste primeramente en definir los niveles de seguridad asociados a cada dato. Es decir, asignar a todos los elementos referenciados en los DUP un nivel de seguridad. Estos niveles de seguridad deben ser asignados por el usuario y definen su política de seguridad. Cuando se menciona los elementos referenciados en los DUPs se quiere significar tanto a la parte izquierda como la derecha del DUP. Por ejemplo, para el DUP (r3, {r2,o.getParam}) se deben definir 3 niveles de seguridad, para r3, r2 y otro para o.getParam.

Con los niveles de seguridad definidos por el usuario, la herramienta, verifica si se cumple con la política de seguridad.

Por ejemplo, dado un DUP (v, u0,u1,...,un) y los niveles de seguridad definidos por el usuario para cada uno de estos elementos S(v), S(u0), …, S(un) entonces se computa el supremo S'(v) = S(u0) ∨ S(u1) ∨....∨ S(un). Luego se verifica que el nivel de S'(v) sea menor o igual al definido para v por el usuario (S'(v) ≤ S(v)). Si S'(v) ≤ S(v) entonces el DUP (v,{u0,u1,...,un}) es un uso seguro, de lo contrario se viola la política de seguridad del usuario.

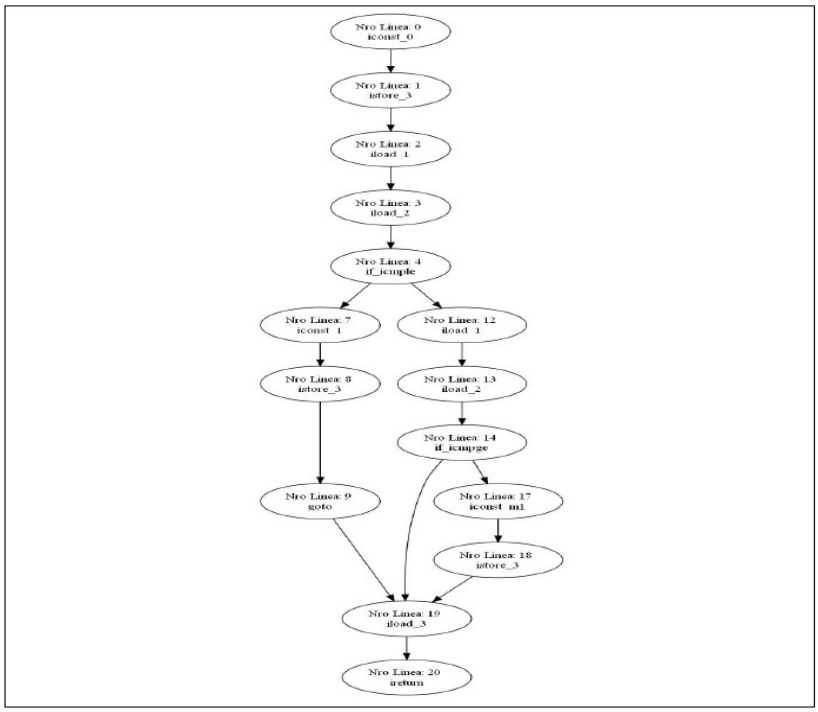
4 Ejemplo de Ejecución del Prototipo

A continuación se mostrará los datos obtenidos de la ejecución de la herramienta para un ejemplo puntual. Primero se muestra el código Java y el bytecode asociado.

```
public class Numero {

/* El método comparar retorna -1 si el primer número es menor al
segundo, 1 si el primero es mayor o 0 en caso de igualdad */

      public int comparar(int primNum, int segNum){

            int resultado = 0;

                  if(primNum > segNum){

                        resultado = 1;

                  }else if(primNum < segNum){

                              resultado = -1;

                  }

            return resultado;

            }
```

```
0:    iconst_0
1:    istore_3
2:    iload_1
3:    iload_2
4:    if_icmple        #12
7:    iconst_1
8:    istore_3
9:    goto        #19
12:   iload_1
13:   iload_2
14:   if_icmpge        #19
17:   iconst_m1
18:   istore_3
19:   iload_3
20:   ireturn
```

El CFG generado es el siguiente:



Los DUPs generados son los siguientes

```
(null, r2)
(r3, r2 Lambda)
(ret, Lambda r2)
```

Donde “r2”, “r3” son variables locales que representan al segundo parámetro y a la única variable local del método analizado, respectivamente. “ret” representa al valor de retorno del método. Lambda es una constante que representa al nivel mínimo.

Si se asignan los siguientes niveles de seguridad: r2=3 y r3=2, con el siguiente orden 2 < 3 (el nivel de seguridad 3 es mayor que el nivel 2). Los DUPS a verificar son los siguientes:

```
(null, 3)
(r3, 3 2)
(ret, 2 3)
```

Luego del análisis se informará que se violó la política de seguridad en los siguientes DUPS:

DUP	Justificación de la violación
(null, r2)	El valor ingresado por el usuario para null es 2, ya que se toma e menor de los valores ingresados por el usuario y el valor computado es 3 porque como se puede observar en el DUP, el uso de la variable null es r2 el cual computa el valor 3 según lo ingresado por el usuario.
(r3, r2 Lambda)	El valor ingresado por el usuario para r3 es 2, y el computado por Bytomic es 3, ya que se obtiene el máximo de los usos de la variable, por lo tanto el máximo entre el valor de r2(3) y Lambda(2) es 3. Lambda al igual que null toma como valor clase de seguridad, el menor de los ingresados por el usuario.

5 Trabajos Relacionados

Dentro de la cantidad de trabajos que abordan el tema interés de este trabajo se describirán a continuación aquellos que han sido utilizados como guía para este desarrollo.

Denning propuso por primera vez un método estático de certificación para verificar el flujo de información seguro de un programa [3]. A cada objeto del programa se le asigna una cierta clase de seguridad. Las clases de seguridad se suponen que forman una estructura reticular, ordenadas por ≤. Su método es un enfoque basado en la aplicación del flujo de información segura. En este trabajo no se

provee una prueba formal de la corrección de la propuesta. Volpano desarrolló un sistema de tipos dirigido por la sintaxis para anotar las variables del programa, los comandos y parámetros de procedimiento con las clases de seguridad . También demostró que su sistema de tipos garantiza no interferencia. Banerjee y Naumann extendieron el sistema de tipos de Volpano con objetos [5]. Su extensión abarca el flujo de datos a través de los atributos de los objetos y el control de flujo a través de llamadas a métodos de forma dinámica. Demuestran formalmente que su sistema es no interferente considerando una cantidad importante de características de un lenguaje: punteros, estados mutables, atributos privados, visibilidad de clases, enlace dinámico y herencia, conversiones de tipo, y clases y métodos recursivos.

Los enfoques basados en sistemas de tipos para asegurar el flujo de información son fáciles de aplicar, pero a menudo son demasiado imprecisos. Consideremos el ejemplo:

B := A; B := 0;

donde B y A corresponden a variables con niveles de seguridad bajo y alto respectivamente. Un sistema de tipos rechazaría este programa basado en la primera asignación, sin embargo, el programa cumple claramente con la propiedad de no interferencia. La mayoría de los enfoques basados en tipos, rechazan cualquier programa con subprogramas inseguros porque evalúan línea por línea del programa y son insensibles al contexto (el contexto del programa se ignora).

En [6,7] proponen un análisis de flujo de la información basado en el análisis de dependencias para programas bytecode. Pero, el subconjunto de bytecode considerado es muy limitado y se detectaron algunas inconsistencias en su análisis. En este trabajo se extiende el análisis considerando excepciones y creación dinámica de objetos. A diferencia de los sistemas de tipos, que analizan y verifican el programa instrucción por instrucción, este enfoque certifica el programa después de analizar todo el programa, por lo tanto puede proporcionar mayor precisión.

Genaim y Spoto presentan un análisis de flujo de la información (sensitivo al contexto) para (mono-threaded) Java Bytecode. En este trabajo, transforman el Bytecode en un grafo de control de flujo de bloques básicos (el cual hace explicito las características complejas del Bytecode). Para representar los flujos de información utilizan funciones booleanas y diagramas de decisión binarios [8].

6 Conclusiones y Trabajos Futuros

Con el desarrollo de esta herramienta se logro el objetivo fijado que era aportar a la enorme tarea de garantizar la seguridad en programas Java Bytecode. El énfasis consistió en garantizar la confidencialidad de los datos, abordando dicho análisis a través de los métodos de control de flujo de la información y análisis de dependencias. Un aporte, importante de mencionar, es la inclusión de manejo de excepciones y permitir creación dinámica de objetos.

Asimismo, se quiere remarcar que este trabajo sólo se abocó a realizar una herramienta con ciertas características, por lo que está abierta la posibilidad de nuevos trabajos que refuercen este enfoque. Este es un “átomo” en el universo de las herramientas de desarrollo que asisten a los desarrolladores en su afán de garantizar la seguridad.

6.1 Trabajos futuros

El prototipo aquí presentado es solo un paso más de un largo camino en busca de un entorno de generación de código móvil seguro que pueda ser usado industrialmente, por lo tanto se debe seguir su desarrollo, extenderlo y optimizarlo para lograr este cometido. Como futuras extensiones de la herramienta se proponen los siguientes puntos:

• Mejorar el manejo de excepciones. Para aumentar la certeza del análisis y aumentar la eficiencia del mismo sería importante realizar pre-análisis para determinar cuáles instrucciones pueden disparar excepciones y qué tipo de excepciones (por ejemplo, análisis de nulless). La información de dichos análisis permitirán reducir la cantidad de bifurcaciones en el flujo de control de los grafos generados. Se debe también agregar el análisis de la instrucción throw, la cual no fue contemplada en este trabajo.

• Con el fin de validar experimentalmente la herramienta, es importante realizar estudios de casos de código real. Los posibles ejemplos incluyen los casos de estudios desarrollados en Jif [1].

• Es interesante extender la herramienta para analizar aplicaciones del sistema operativo Android, el cual tiene su propia SDK.

Referencias

[1] A. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. http://www.cs.cornell.edu/jif/, 2001. Software release.

[2] Gaowei Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa. “Java Bytecode Dependence Analysis for Secure Information Flow”. International Journal of Network Security, Vol.4, No.1, PP.59–68, Jan. 2007

[3] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow”, Communications of the ACM, vol. 20, no. 7, pp. 504-513, 1977.

[4] D. Volpano, G. Smith, and C. Irvine, “A sound type system for secure flow analysis,” Jornal of Computer Security, vol. 4, no. 3, pp. 167-187, 1996.

[5] A. Banerjee and D. Naumann, “Secure information flow and pointer confinement in a java-like language,” in Proceedings of IEEE Computer security Foundations Workshop, pp. 253-267, June 2002.

[6] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa, “Mobile code security by java bytecode dependence analysis,” in Proceedings of the International Symposium on Communications and Information Technologies 2004 (ISCIT 2004), pp. 923-926, Sapporo, Japan, Oct. 26- 29, 2004.

[7] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa, “Java Mobile Code Security by Bytecode Analysis,” ECTI Transactions on Computer and Information Technology, vol. 1, no. 1, pp. 30-39, 2005.

[8] Samir Genaim and Fausto Spoto. Information Flow Analysis for Java Bytecode. Verification, Model Checking, and Abstract Interpretation Lecture Notes in Computer Science, Volume 3385/2005, 346-362. 2005.