Garantizando Confidencialidad de la Información en Programas Bytecode con Análisis de Dependencias

Renato Meneghini y Francisco Bavera

Departamento de Computación
Facultad de Ciencias Exactas Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
rmeneghini@gmail.com, pancho@dc.exa.unrc.edu.ar

Abstract. Este trabajo presenta un análisis 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, Infomation-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 también 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.

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: (1) **confidencialidad**, los datos secretos (confidenciales) no pueden deducirse de los datos públicos y (2) **integridad**, los cómputos críticos no pueden ser manipulados por usuarios no permitidos.

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. Por esto, las técnicas estáticas se han convertido en el mecanismo de aplicación primario para las políticas del flujo de información. 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.

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: (1) los tipos de los valores almacenados en una misma variable (o registro) pueden cambiar durante la ejecución del programa; (2) hay instrucciones de salto 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
```

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

Ejemplo 3: (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

El proceso implementado por la herramienta 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.

Como se puede ver en la Fig. 1, 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 (esta información se almacena en un archivo XML). 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 considerado seguro.

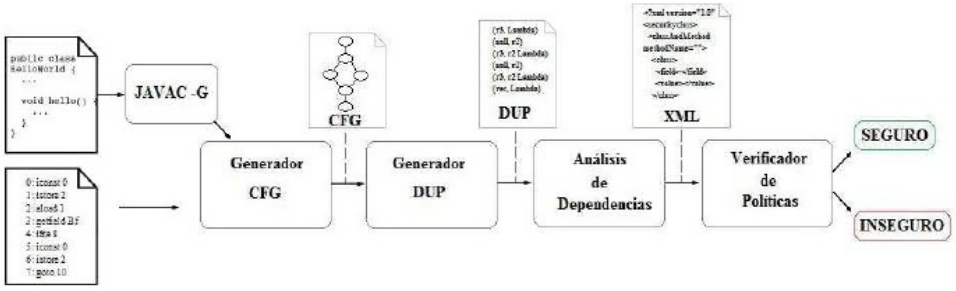


Fig. 1. Módulos y etapas del proceso de verificación.

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), manejo de excepciones (throw). 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. El CFG se define como:

CFG : (V,A) = {S1, S2, S3, . . . Sm} y Si = {I0, I1, . . . 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* (o de salto), salvo que dependan del flujo principal, en dicho caso se indica con -1.

Las instrucciones de *branch* en el CFG se clasifican en 4 conjuntos: condicionales, sin condición, compuestas y por excepción. Por ejemplo, goto, ifeq, ifnull, tableswitch, getfield, putfield, entre otros.

Los branch generados por las instrucciones que pueden dispar una excepción dependen si tienen o no un handler (manejador de excepciones) definido o no. Dicha información se extrae de la tabla de manejadores de excepciones de cada método. Al detectar una instrucción que puede disparar una excepción se genera 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.

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 pila 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 |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 >	

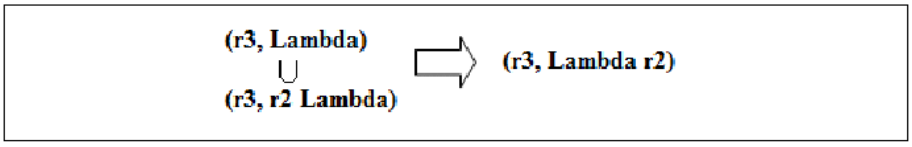
3.4 Análisis de Dependencias

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

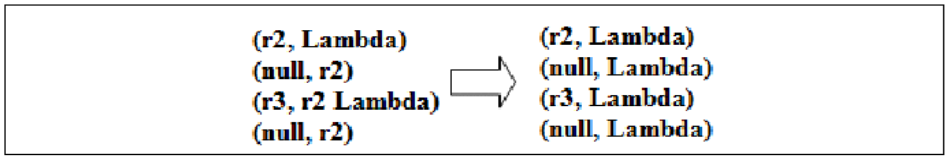
$$\{D_i\} \cup_m \{D_j\} = \begin{cases} \{d_i, U_i \cup U_j\} \; si \; d_i = d_j \\ \{D_i \cup D_j\} \; otros \; casos \end{cases}$$



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 $D_i \cup D_j = (d_j, U_i \cup U_j)$



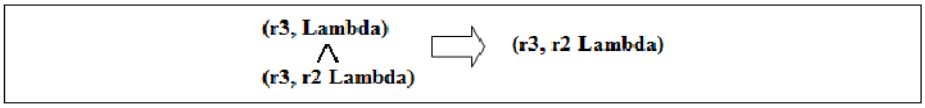
Operaciones con constantes:

$U_i \cup \lambda = U_i$

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 en, primero, 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 a 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 $S'(v) = S(u0) \vee S(u1) \vee .... \vee S(un)$. Luego se verifica que el nivel de *S'(v)* sea menor o igual al definido para *v* por el usuario $(S'(v) \leq S(v))$. Si $S'(v) \leq S(v)$ 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 mostran 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) y *(ret, Lambda r2)*.

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

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) y (ret, 2 3)*.

Luego del análisis se informará que se violó la política de seguridad, ya que, el nivel de *r3* es *2* y la herramienta computa que se le asigna un valor de nivel superior (en este caso *3*).

5 Trabajos Relacionados

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

Denning y Denning [3] propusieron por primera vez un método estático de certificación para verificar el flujo de información seguro de un programa. Banerjee y Naumann [5] extendieron el sistema de tipos de Volpano et. al [4] con objetos. 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. Los enfoques basados en sistemas de tipos para asegurar el flujo de información son fáciles de aplicar, pero a menudo son demasiado imprecisos. La mayoría de los enfoques basados en tipos, rechazan cualquier programa con subprogramas inseguros porque evalúan línea por línea del programa y no son sensibles al contexto.

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 [8] presentan un análisis de flujo de la información (sensible 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. Nuestro análisis no necesita construir los bloques básicos.

6 Conclusiones

Se presento una herramienta que permite analizar programas Bytecode y determinar si garantizan la confidencialidad de los datos manipulados. El análisis que realiza la herramienta es sensible al contexto y se basa en métodos (tradicionales en la compilación y análisis de código) de control de flujo y análisis de dependencias. Se extendió el análisis para permitir manejo de excepciones y creación dinámica de objetos. Con estas extensiones, se considera un subconjunto muy importante del lenguaje Bytecode, consideramos que es posible utilizar el método sobre aplicaciones reales.

Con el fin de estudiar el alcance y la efectividad de la herramienta es necesario realizar experimentaciones con aplicaciones reales. Estamos trabajando en la extensión del análisis para permitir declasificación de datos y también para incluir threads al lenguaje. Además, queremos establecer y demostrar si el análisis propuesto garantiza alguna propiedad, como por ejemplo, no interferencia.

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.