Seguridad en Java: sobre el diseño formal de máquinas virtuales

David Cerezo Sánchez

http://david.cerezo.name

Requisitos

- Lenguaje de programación Java
- Lógica matemática
- Máquinas virtuales
- Teoría de lenguajes de programación

El lenguaje Java

- Java = lenguaje orientado a objetos + plataforma independiente SO + JVM
- Inicialmente creado para sistemas con poca memoria. Luego se extendió para terminar centrándose en programas que se ejecutasen en entornos de red heterogéneos.
- Su diseño se inspira en C++: se le puede considerar una versión limpia de C++.
- JVM (Java Virtual Machine): ordenador abstracto definido no formalmente que ejecuta 200 instrucciones y dispone de stack y registros.
- De renovado interés por su uso en teléfonos móviles y *smartcards* (ejecutan un subconjunto de Java).

El modelo de seguridad de Java

- Seguridad con respecto al manejo de la memoria:
  - ★ recolección de basura: el programador no controla la liberación de memoria
  - ★ *strong type-safety*: sin aritmética de punteros y estricta comprobación del tipo
  - ★ en tiempo de ejecución se comprueban los accesos a arrays y *casts* entre tipos.

- Verificación del bytecode: complejo algoritmo que busca garantizar que el código Java compilado mantenga las propiedad de *type-safety*, entre otras. De esta manera, se elimina gran parte del proceso de verificación en tiempo de ejecución, reduciéndose a comprobaciones de violación de límites de memoria en variables y de políticas de seguridad.

- *Sandbox* para applets: entorno seguro de ejecución para programas en los que no se puede confiar, prohibiéndose cualquier acceso al sistema subyacente. Existe una política por defecto del sandbox, modificable, que evita:
  - ★ cargar librerías.
  - ★ ejecución de otros programas en el sistema cliente.
  - ★ conexiones de red a ordenadores remotos que no sean el host donde reside el applet.
  - ★ creación de sockets en estado de escucha.
  - ★ leer y escribir archivos en el ordenador cliente.

El modelo de seguridad provisto por un *sandbox*, basado en el aislamiento de las aplicaciones entre sí, es parecido al proporcionado por el hardware para el uso de los sistemas operativos, basado en el control de acceso a páginas de memoria y a requerir que las operaciones pasen por el sistema operativo. De la misma manera, la máquina virtual controla el acceso a memoria y a recursos del sistema.

Type-safety

Forzando esta propiedad del código, se busca evitar los siguientes problemas:

- falsificación de punteros (*pointer forging):*
  - ★ utilizando aritmética de punteros
  - ★ con cast de enteros: (byte []) 0x1000
- accesos fuera de los límites definidos en una estructura de datos
  - ★ casos especiales de esto son la gran parte de los buffers overflows
- cast no válidos:
  - ★ por ejemplo: de *class A* {*int a;*} a *class B* {*byte[] b;*}

- manejo de la memoria por el programador:
  - liberando un objeto y manteniendo su referencia hasta que se reasigne espacio y se pueda acceder a otras zonas de memoria con esa referencia

Java Class Loader

- Class Loader: primera línea defensiva, se compone de objetos y clases que cargan clases Java en la JVM.
- En una misma JVM pueden existir múltiples Class Loader, cada uno de ellos definiendo su propio espacio de clases.
- Cada clase cargada almacena una referencia al Class Loader que lo cargó.
- Diferentes tipos de Class Loader, cada uno con políticas diferentes:
  - ★ System Class Loader: carga clases del sistema.
  - ★ Applet Class Loader: carga applets y sus clases asociadas.
  - ★ Remote Method Invocation(RMI) Class Loader: carga clases provenientes de RMI.

Java Security Manager

- Guarda la políticas de seguridad de las aplicaciones Java
- Antes de la ejecución de cada operación potencialmente peligrosa, se comprueba que no viole la política establecida para la aplicación
- Se debe proporcionar explícitamente acceso a aquellas funcionalidades peligrosas

Java Bytecode Verifier

- Algoritmo que comprueba la consistencia e integridad de las clases Java durante la carga y linkado, buscando que se cumpla la propiedad de type-safety.

- Comprueba que:
  - ★ el formato de archivo es correcto
  - ★ la definición de la clase es correcta
  - ★ el código no viola privilegios de acceso
  - ★ simula el flujo de ejecución de cada instrucción, buscando que se man-tenga el tipo: los parámetros de las instrucciones deben ser correctos
  - ★ la altura del stack y los tipos de stack y registros se debe mantener: no ocurren stack overflows ni underflows

- Formalmente, se lo podría describir como una relación de transición en una máquina virtual a nivel de los tipos de variables, operándose sobre

tipos y no valores concretos:

$$instrucción : (\tau_{registros}, \tau_{stack}) \longrightarrow (\tau'_{registros}, \tau'_{stack})$$

en la que se configuran ecuaciones de flujo de ejecución y datos, que se resolverán utilizando iteración de punto fijo standard utilizando el algoritmo de Kindall para listas de trabajo. Algunas reglas del intérprete abstracto son las siguientes:

$$\begin{aligned}
\text{iconst}\, n &: (S,R) \rightarrow (\text{int}.S,R) \ \text{si} \ |S| < M_{stack} \\
\text{ineg} &: (\text{int}.S,R) \rightarrow (\text{int}.S,R) \\
\text{iadd} &: (\text{int}.\text{int}.S,R) \rightarrow (\text{int}.S,R) \\
\text{iload}\, n &: (S,R) \rightarrow (\text{int}.S,R) \\
& \quad \text{si } 0 \leq n < M_{reg} \text{ y } R(n) = \text{int y } |S| < M_{stack} \\
\text{istore}\, n &: (\text{int}.S,R) \rightarrow (S, R\{n \leftarrow \text{int}\}) \ \text{si } 0 \leq n < M_{reg} \\
\text{aconstnull} &: (S,R) \rightarrow (\text{null}.S,R) \ \text{si} \ |S| < M_{stack}
\end{aligned}$$

$$\begin{aligned}
\text{aload}\, n &: (S,R) \rightarrow (R(n).S,R) \\
&\quad \text{si } 0 \leq n < M_{reg} \text{ y } R(n) <: \text{Object y } |S| < M_{stack} \\
\text{astore}\, n &: (\tau.S,R) \rightarrow (S,R\{n \leftarrow \tau\}) \\
&\quad \text{si } 0 \leq n < M_{reg} \text{ y } \tau <: \text{Object} \\
\text{getfield}\, C.f.\tau &: (\tau'.S,R) \rightarrow (\tau.S,R) \text{ si } \tau' <: C \\
\text{putfield}\, C.f.\tau &: (\tau_1.\tau_2.S,R) \rightarrow (S,R) \text{ si } \tau_1 <: \tau \text{ y } \tau_2 <: C \\
\text{invokestatic}\, C.m.\sigma &: (\tau'_n \ldots \tau'_1.S,R) \rightarrow (\tau.S,R) \\
&\quad \text{si } \sigma = \tau(\tau_1,\ldots,\tau_n), \tau'_i <: \tau_i \\
&\quad \text{para } i = 1 \ldots n \text{ y } |\tau.S| \leq M_{stack} \\
\text{invokevirtual}\, C.m.\sigma &: (\tau'_n \ldots \tau'_1.\tau'.S,R) \rightarrow (\tau.S,R) \\
&\quad \text{si } \sigma = \tau(\tau_1,\ldots,\tau_n), \tau' <: C, \tau'_i <: \tau_i \\
&\quad \text{para } i = 1 \ldots n, |\tau.S| \leq M_{stack}
\end{aligned}$$

La ausencia de transiciones en las ecuaciones anteriores indicarían un

error de tipo.

- El algoritmo opera método por método, bajo la conjetura de que el resto que de los métodos está bien tipado: por coinducción, si todos los métodos están bien tipados, también lo estará el programa.

- Propiedades formales que debe cumplir el intérprete abstracto:

1. **Determinancia**: las transiciones de intérprete abstracto definen un función de orden parcial, esto es, si la transición i puede ser descrita como

$$i : (S,R) \longrightarrow (S_1,R_1) \text{ y } i : (S,R) \longrightarrow (S_2,R_2)$$

entonces

$$S_1 = S_2 \text{ y } R_1 = R_2$$

2. **Monotonicidad**: dada la transición

$$i : (S,R) \longrightarrow (S',R')$$

entonces por cualquier tipo del stack $S_1 <: S_2$ y tipo de los registro $R_1 <: R_2$, existe un tipo del stack S_1' y tipo del registro R_1' tal que

$$i : (S_1, R_1) \longrightarrow (S_1', S_1') \text{ con } S_1' <: S' \text{ y } R_1' <: R$$

3. **Correcta correspondencia** con la semántica dinámica de la JVM: toda transición

$$i : (S, R) \longrightarrow (S', R')$$

del interpretador abstracto, debe ser ejecutada sin errores de tipo en una DJVM. Probada y formalizada por [40, 74, 138, 140].

- En los cambios del flujo de control (branch, exceptions,...), el estado de los tipos del stack y de los registros antes de una instrucción es la menor cota superior de todos los estados después después de todos los predecesores de la instrucción. Una descripción del algoritmo sería la siguiente:

$$\begin{aligned} i : in(i) &\longrightarrow out(i) \\ in(i) &= lub\{out(j) \text{ tal que } j \text{ es un predecesor de } i\} \end{aligned}$$

$$in(i_{start}) \quad = \quad ((P_0, \ldots, P_{n-1}, \top, \ldots, \top), \varepsilon)$$

es decir, se toma un programa en el punto i de la lista y su estado tras $out(i)$ vendrá determinado por su estado antes de $in(i)$ utilizando el intérprete abstracto. Por cada sucesor j de i, se reemplaza $in(j)$ por $lub(in(j), out(i))$, introduciéndose en la lista los sucesores j para los que $in(j)$ cambió en la lista. Cuando la lista termina y está vacía, la verificación concluye con éxito; la verificación fallará si hay estados sin transiciones o lub está no definido.

- Formalizaciones del algoritmo de verificación de bytecode:
  - ★ En Isabelle/HOL [172, 103, 101]
  - ★ Usando sistemas de tipos [159, 70, 68]
  - ★ Zhenyu Qian [140]
  - ★ Alessandro Coglio, Allen Goldberg, Zhenyu Qian [40]
  - ★ Robert Stärk, Joachim Schmid, Egon Börger [157]

Verificación práctica de bytecode

- La tercera fase de la verificación de bytecode Java realiza las siguientes comprobaciones estáticas en cada una de las instrucciones:
  - ★ El objetivo de cada instrucción de salto, condicional, de cada elemento de la tabla de salto de la instrucción *tableswitch* o de las parejas de *lookupswitch* debe apuntar al opcode de una instrucción del código de un método. Además, no podrán apuntar al opcode de una instrucción que sea modificada por una instrucción *wide* a menos que apunte a la propia instrucción *wide*.
  - ★ El número de entradas en una tabla de saltos de una instrucción *tableswitch* debe ser consistente con sus operandos alto y bajo de tabla; el valor del operando bajo deberá ser menor o igual que el valor del operando alto.
  - ★ El número de parejas de cada instrucción *lookupswitch* debe ser consistente con su operando *npair*. Las parejas deben estar ordenadas numéricamente de menor a mayor.

- ⋆ Los operandos de cada instrucción *ldc* y *ldc_w* deben ser constantes válidas de tipo *int, float* o *String*; además, los operandos de cada instrucción *ldc2_w* debe ser una constante válida de tipo *long* o *double*.
- ⋆ El operando de cada instrucción *getfield, putfield, getstatic,* y *putstatic* deben ser referencias válidas a un campo; además, los operandos de cada instrucción *invokevirtual, invokespecial,* y *invokestatic* deben ser referencias válidas a un método.
- ⋆ El operando de cada instrucción *instanceof*, *checkcast*, *new, anewarray* y *multinewarray* debe ser una referencia válida a una clase.
- ⋆ El operando *atype* de cada instrucción *newarray* debe ser de tipo *boolean*, *char*, *float*, *double*, *byte*, *short*, *int* o *long*.
- ⋆ El índice ímplícito de cada instrucción *iload*, *fload*, *aload*, *istore*, *fstore*, *astore*, *wide*, *iinc*, *ret* debe referenciar una variable local válida, entre 0 y *max_locals-1*.
- ⋆ El índice ímplícito de cada instrucción *iload_<n>*, *fload_<n>*, *aload_<n>*, *istore_<n>*, *fstore_<n>*, *astore_<n>*, *wide_<n>*, *iinc_<n>*, *ret_<n>* debe referenciar una variable local válida, entre 0 y *max_locals-1*.

- ⋆ El operando índice de cada instrucción *lload*, *dload*, *lstore* y *dstore* debe referenciar una variable local válida, entre 0 y *max_locals-2*.
- ⋆ El índice implícito de cada instrucción *lload_<n>*, *dload_<n>*, *lstore_<n>* y *dstore_<n>* debe referenciar una variable local válida, entre 0 y *max_locals-2*.
- ⋆ Ninguna instrucción de invocación de métodos podrá llamar a métodos que comiencen con "<", excepto *invokespecial* que podrá llamar al método de inicialización de instancias "<init>".
- ⋆ El operando de cada instrucción *invokeinterface* debe ser una referencia válida a un interfaz y el valor del operando *nargs* debe coincidir con el número de argumentos del interfaz del método dado. El cuarto byte de cada instrucción *invokeinterface* debe tener el valor de 0.
- ⋆ La instrucción *new* no se utilizará para crear un array, interfaz o instancia de una clase abstracta.
- ⋆ Ninguna instrucción *anewarray* creará arrays de más de 255 dimensiones.
- ⋆ La instrucción *multianewarray* sólo se utilizará para crear un array de tipo con al menos tantas dimensiones como el valor de su operando

dimensional, siempre diferente de 0. No se tendrá porqué crear todas las dimensiones de tipo de array, pero creará más dimensiones de las definidas en su tipo de array.

- La tercera fase de la verificación de bytecode Java realiza las siguientes comprobaciones estructurales:
  - ⋆ Por cada instrucción, el stack de operandos y las variables locales deben tener siempre el tipo correcto y en el número adecuado.
  - ⋆ El stack de operandos debe contener siempre el mismo número de elementos.
  - ⋆ La palabras de cada palabra doble no podrán ser accedidas individualmente, ni intercambiadas, ni separadas.
  - ⋆ Ninguna variable local puede ser accedida antes de ser asignada un valor.
  - ⋆ El stack de operandos no podrá crecer más del número de elementos permitido, ni se le podrán extraer más elementos de los que contiene.
  - ⋆ La instrucción *invokespecial* sólo se utilizará para llamar al método de inicialización “<*init*>”, a un método en la clase actual, a un método

privado o a un método en una superclase de la clase actual. Si se utiliza para llamar a "$<$*init*$>$", tendrá que haber una clase no inicializada en el stack de operandos.

⋆ Los métodos de instancia sólo podrán ser utilizados por instancias inicializadas de las clases que lo contengan.

⋆ No se podrá acceder a ninguna variable de instancia antes de que la clase de instancia que lo contiene sea inicializada.

⋆ No habrá instancias de clases no inicializadas en el stack de operandos o en una variable local cuando se tome cualquier salto hacia atrás. Tampoco habrá instancias de clases no inicializadas en una variable local en el código protegido por un cláusulas finales o de excepción. Sin embargo, una instancia de clase no inicializada podría estar el stack de operandos en el código protegido por una cláusula final o de excepción. Cuando se lance una excepción, los contenidos del stack de operandos serán descartados.

⋆ Cada método de inicialización de instancias, excepto el método de inicialización de instancias de la clase *java.lang.Object*, debe llamar o al método de inicialización de instancias de la clase actual o a un método

de inicialización de instancias de la superclase inmediata antes de que cualquiera de sus variables de instancia sean accedidos.

- ⋆ Los argumentos de cada método invocado deben ser compatibles con el descriptor de métodos correspondiente.
- ⋆ Un método abstracto no puede ser invocado.
- ⋆ El retorno de un método se hará en función del tipo de retorno declarado: usando la instrucción *ireturn* para devolver *byte, char, short* o *int*; la instrucción *freturn*, *lreturn* o *dreturn* para devolver *float*, *long* o *double*; la instrucción *areturn* para devolver una referencia, con el valor devuelto compatible en cuanto a asignación con el descriptor de retorno del método; el resto, métodos de inicialización, inicializadores estáticos y métodos que no devuelven valor alguno usaan la instrucción *return*.
- ⋆ Si se accede al campo protegido de una superclase con las instrucciones *getfield* o *putfield*, entonces el tipo de la clase instanciada debe ser el mismo o de una subclase de la clase actual. Si se usa la instrucción *invokevirtual* para acceder a un método protegido de una superclase, entonces el tipo de la instancia de clase a la que se accede debe ser igual o de una subclase de la clase actual.

- ★ Las instrucciones *getfield* o *putfield* sólo se usarán para acceder a instancias de clase que son del tipo de clase o subclase del tipo de clase declarado en del descriptor de campo correspondiente.
- ★ El tipo de cada valor almacenado por una instrucción *putfield* o *putstatic* debe ser compatible con el descriptor del campo de cada instancia de clase o clase en la que se guarda. Si el tipo del descriptor es *byte*, *char*, *short* o *int,* entonces el valor debe ser *int*; si el tipo del descriptor es *float*, *long* o *double*, el valor será *float*, *long* o *double*; por último, si el tipo del descriptor es de referencia, entonces el valor debe ser de un tipo cuya asignación sea compatible con el descriptor del tipo.
- ★ La instrucción *aastore* sólo podrá ser usada para almacenar valores referenciales en un array. El tipo de los valores almacenados debe ser compatible en cuanto a asignación con el tipo que compone el array.
- ★ La instrucción *athrow* sólo podrá retornar valores que sean instancias de la clase o la subclase de *java.lang.Throwable*.
- ★ La ejecución no continuará después de la última instrucción del código.
- ★ No se cargarán direcciones de retorno desde una variable local en la pila de operandos. Sin, embargo, lo contrario se permite siempre y cuando

los valores de las direcciones de retorno puedan ser almacenados en variables locales del stack de operandos.

- ⋆ El retorno tras cada instrucción que siga las instrucciones *jsr* o *jsr_w* sólo podrá ser realizado con el uso de una única instrucción *ret*.
- ⋆ Las instrucciones *jsr* o *jsr_w* no podrán ser utilizadas para llamar recursivamente a una subrutina que ya esté presente en la cadena de llamadas de las subrutinas.
- ⋆ El retorno a cada instancia del tipo *returnAddress* sólo podrá ser realizado una vez. Si la instrucción *ret* vuelve al punto en la cadena de llamadas de la subrutina anterior a la instrucción *ret* correspondiente a una instancia del tipo *returnAddress*, entonces esa instancia no podrá ser utilizada como dirección de retorno.

Problemas verificación de bytecode

- Sin embargo, hay partes del proceso de verificación que escapan a este método de análisis, como la subrutinas, interfaces y la inicialización de objetos:
  - **Interfaces**: también son tipos y cada clase puede implementar varios interfaces, por lo que puede darse el caso de que existan pares de sub-tipos que no tengan una menor cota superior. Soluciones:
    - SUN: todos los interfaces son de tipo *Object*. No se puede garantizar estáticamente que la referencia a un objeto implemente un interfaz concreto.
    - Conjuntiva: se manipulan conjuntos de tipos, en vez de tipos únicos.
    - Complección Dedekind-MacNeille de un *poset*: la jerarquía de clases e interfaces se completa introduciendo todos los tipos intermedios que sean necesarios, fruto de la unión de varias interfaces.
  - **Inicialización de Objetos:** dividido en dos operaciones diferentes, *new C*, para crear el objeto sin inicializarlo, más una llamada al método

<*init*> de la clase para inicializar el objeto; después, se almacena una referencia al objeto en la variable especificado con la instrucción *astore*. Esto rompe la propiedad de monotonicidad, por lo que hay que tomar diversas restricciones para asegurar un correcto análisis del código.

⋆ **Subrutinas**: complica el análisis porque los puntos de entrada de las subrutinas aunan demasiados tipos y porque no se pueden determinar los sucesores de una instrucción *ret*, ya que la dirección de retorno es un valor *first-class*. Soluciones:

- SUN: busca implementar que una llamada a una subrutina no debe cambiar el tipo de los registros que no se utilizan en el cuerpo de la subrutina: para ello, el estado tras la instrucción *i* siguiendo a una instrucción *jsr* es idéntica al estado tras la instrucción *ret*, excepto para los tipos de los registros que no se utilizan en la subrutina. Formalizado en [140, 157, 158, 70, 170].
- [159]: antes de la verificación de tipos, se infiere la pertenencia de las instrucciones a las subrutinas. No considera las excepciones ni la inicialización de objetos.
- [78]: el análisis previo otorga un tipo polimórfico sobre los tipos de las

variables locales que no utiliza. No considera las excepciones ni la inicialización de objetos.

- [128]: se ofrecen reglas de chequeo de tipos utilizando *continuation types*, tales que el tipo asignado a la dirección de retorno contiene el stack completo y el tipo de registro siguientes a la instrucción *jsr*.
- Algoritmos de verificación polivariantes [14, 28, 37, 85, 104, 132, 167]: en el que cada punto del programa tiene asociados múltiples estados posibles, permitiendo así que las instrucciones dentro de una subrutina puedan ser analizadas múltiples veces. Ejemplos:
  - Algoritmo Brisset-Coglio[28, 37]: explora todos los estados posibles aplicando varias veces veces una relación de transición extendida comenzando con el estado inicial del método. Aunque de complejidad exponencial, es el más correcto de todos los algoritmos propuestos.
  - Algoritmo Coglio-Frey-Henrio-Serpette [37, 85]: versión ampliada de BC que reduce el número de estados y, por tanto, la complejidad de la comprobación.

Pero... ¿es Java *type-safe*?

- En la definición del lenguaje, no se ofrece una demostración de que Java sea *type-safe*: sin esta demostración, no se puede afirmar que Java sea seguro.

- Ha habido un intenso trabajo para demostrar que Java es *type-safe*: [55, 56, 58, 59, 90, 127, 162, 174, 175, 40, 74, 98, 102, 180]

  ★ Principalmente se ha demostrado que subconjuntos de Java son *type-safe*, y luego se ha ido extendiendo la demostración.

Java seguro != JVM segura

Técnicas de ataque y contraejemplos:

- *DNS spoofing* en Navigator $<$2.01 y JDK 1 (Dean, Felten, Wallach @ Princeton, Feb. 1996): el SecurityManager no comprobaba si la dirección IP de la que se ha obtenido el applet era la primera de las direcciones en la lista de IPs de las respuestas DNS o si se podía resolver al mismo nombre DNS que en el valor del host de la propiedad *CODEBASE*. Como resultado, los applets podían establecer conexiones a direcciones de red arbitrarias.

- Fallo de implementación en el ClassLoader, Navigator $<$2.02 y JDK $<$1.01 (Hopwood @ Oxford, Mar. 1996): los nombres completos de clases podían empezar con barra invertida, por lo que se podía cargar código fuera de los directorios indicados por la variable de entorno *CLASSPATH*, por ejemplo, la caché del navegador. Por lo tanto, se podía cargar remoto como si fuese código en el que se confiase.

- Fallo en el verificador de bytecode, Navigator <2.02 y JDK <1.02 (Dean, Felten, Wallach @ Princeton, Mar. 1996): se podían crear objetos parcialmente inicializados de clases base que invocaban a las comprobaciones del SecurityManager. Como consecuencia, se podía heredar de clases de seguridad críticas, como el SecurityManager o el ClassLoader.

- Applets hostiles, Navigator <=3.0 (LaDue, Abr. 1996): ataques de denegación de servicio con applets, que consumían los recursos del sistema.

- Fallo de implementación en el ClassLoader, Navigator <=3.0 (Cargill, May. 1996): ya que todos los interfaces de las interfaces eran públicos y se podía implementar una interfaz heredando un método privado de la clase padre, las protecciones de seguridad provisto por el lenguaje Java resultaban inútiles.

- Conversión de tipos no válida, JDK <=1.02, Navigator <=2.02 y Explorer 4.0b1 (Hopwood @ Oxford, Jun. 1996): las clases se consideraban sólo por su nombre, sin tener en cuenta el espacio de nombres donde residía

cada una. Por lo tanto, se podían confundir clases de espacios de nombres diferentes y con el mismo nombre, pero con definiciones diferentes.

- Fallo en máquina virtual, JDK <=1.02 (SUN, Mar. 1997): sin descripción.

- Fallo relacionado con código firmado, JDK <=1.1 y HotJava (Dean, Felten, Wallach @ Princeton, Apr. 1997): por una mala implementación del método *getSigners()*, los applets firmados podían modificar la lista de firmas de una clase, posibilitando así el incremento de privilegios cuando se añadían usuarios de más privilegio.

- Fallos en el verificador de bytecode, JDK <=1.1 y Explorer <=4.0 (Proyecto Kimera, May-Jun. 1997): se desarrolla un sistema de verificación que encuentra 24 errores en la implementación de SUN y 17 errores en la implementación de Microsoft.

- Fallo de implementación en el ClassLoader, Navigator <4.5 (LaDue y Dean, Felten, Wallach @ Princeton, Jul. 1998): dos fallos combinados

permiten el cambio de tipos a variables arbitrarias. La primera, permite crear subclases de la clase *AppletClassLoader*, y la segunda, sobreescribir definiciones de algunas clases del sistema, como *Throwable*.

- Cambio de tipos por error en el verificador, JDK 1.1.x (Karsten Sohr @ Marburg, Mar. 1999): el verificador no comprobaba la última instrucción del método verificado cuando ésta estaba en el exception handler.

- Código no verificado, JDK 1.1-1.17 (Haar @ Jive, Abr. 1999): existía una manera de construir clases no verificadas, pero no fue publicada.

- Race conditions durante la carga de clases, Explorer 4-5 (Dean@PARC y Wallach@Rice, Abr. 1999): una *race condition* en el *URLClassLoader* permitía interrumpir el proceso de carga de una clase cuando se estaba realizando la búsqueda de una clase del sistema, forzando así la carga de clases de sistema provista por el atacante.

- Cambio de tipos por error en el verificador, MSIE 4-5 (Karsten Sohr @

Marburg, Oct. 1999): el verificador no realizaba análisis del flujo de instrucciones cuando éstas estaban en el exception handler.

- Lectura de archivos sobrepasando las verificaciones de la máquina virtual, Explorer 4.x-5.x (Nakamura @ NEC Japan, Feb. 2000).

- Error de implementación de clases de red, Navigator 4.0-4.74 (Brumleve, Ago. 2000): un applet podía sobrecargar los métodos *open/close* de *ServerSocket* y *Socket*, posibilitandola comunicación con cualquier host.

- Error de implementación de SecurityManager y clases de red, Navigator 4.0-4.74 (Brumleve, Ago. 2000): heredando métodos de *URLConnection* y *URLInputStream*, se podía engañar al *SecurityManager* haciéndole creer que tenía suficientes privilegios como para abrir conexiones arbitrarias.

- Llamadas a componentes ActiveX, Explorer 4.x-5.x (Oct. 2000): se podían crear y ejecutar componentes ActiveX desde de applets Java.

- Errores de seguridad durante la carga de clases, JDK 1.1.x-12.x (SUN, Nov. 2000): se podía llamar a clases restringidas desde clases en las que no se podía confiar. No hay detalles publicados.

- Cambio de tipos por fallos en el verificador de bytecode, JDK 1.x y Explorer 4-6 (Trusted Logic, Mar 2002): llamadas a la superclase de clases no relacionadas con la superclase original.

- Confusión de tipos por fallos en el verificador, MSIE 4.01 (LSD, Oct. 2002): el verificador no realizaba correctamente la operación de unión de direcciones de retorno, por lo que se seguían flujos de ejecución incorrectos.

- Bug en JIT de Symantec, Netscape 4.0-4.8 (LSD, Oct. 2002): posibilidad de ejecución de código máquina arbitrario al tener acceso al registro EAX por un fallo en la conversión de bytecode a código máquina.

- Implementación incorrecta de clases del sistema, Netscape 4.0-4.8 (LSD, Oct. 2002): el verificador de bytecode hace un análisis lineal del código, llegando a simular instrucciones que nunca se ejecutan y pudiéndose

crear instancias parcialmente inicializadas de clases. Gracias a la implementación del SecurityManager, se consigue acceso a los archivos y red.

- Implementación incorrecta de clases del sistema, Netscape 4.x (LSD, Oct. 2002): el constructor de la clase sun.jdbc.odbc.JdbcOdbc hace una llamada a System.loadLibrary de forma insegura, posibilitando la carga de librerías en la JVM.

- Class Spoofing, MSIE [4|5|6] (LSD, Oct 2002): por un error en el verificador de bytecode, se pueden crear clases inicializadas incluso si la superclase genera excepciones. Puede ser utilizado para provocar Class Spoofing, y a su vez, una confusión de tipos arbitrarios.

“Reflections on trusting trust”

- Publicación[163] basada en la charla que Ken Thompson, uno de los padres del UNIX, realizó en la entrega del premio Turing.

- Expone el “hipotético” caso de un compilador modificado para que cuando se compile el programa *login*, éste aceptase una clave predeterminada.

- Ocurrió realmente en los primeros Unix: se les llamaba frecuentemente para arreglar otros sistemas y disponer de acceso root al sistema les haría más fácil la tarea, por lo que introdujeron modificaciones en el compilador para que cada vez que se compilaba *login*, se le añadiese una puerta trasera difícil de detectar.

- La situación actual con Java y las máquinas virtuales no es muy diferente: se confía ciegamente en su correcto funcionamiento sin demostración alguna de que éstas funcionen correctamente.

Tipos de máquinas virtuales

1. Máquinas virtuales defensivas: lenta comprobación dinámica, en tiempo de ejecución.
   - Comprueba las restricciones de tipo antes de ejecutar cada instrucción, además de los accesos a arrays, el correcto funcionamiento del stack, etc...
2. Verificación de bytecode durante la carga
   - El análisis estático del código busca garantizar la propiedad de *type-safety*, permitiendo una ejecución más rápida por una máquina virtual no defensiva, que ha trasladado gran parte de las comprobaciones que se realizan en tiempo de ejecución en una máquina virtual defensiva.

- Alternativas pausibles para smartcards:

- ⋆ Verificación ligera de bytecode [110, 144, 146]: almacenar el stack y los tipos de los registros como certificados con el código, aliviando parte del proceso de verificación. Sin embargo, el código aumenta notablemente. Formalizada y demostrada válida en [102, 101]. Utilizado en la KVM.
- ⋆ Verificación restringida junto transformación del código [110]: reescribir automáticamente el código para que sea más fácil de verificar, junto a la restricción de que sólo exista un tipo de registro global para disminuir el consumo de memoria. El aumento del tamaño del código es muy pequeño: 2 %.
- ⋆ Especificaciones formales y demostraciones de consistencia de tipos (Nipkow & Pusch).
- ⋆ Generar código de verificadores de bytecode y máquinas virtuales no defensivas a partir de una máquina virtual defensiva (Barthes).

Análisis estático y semántico para asegurar el entorno de ejecución

- Análisis estáticos de la política de seguridad en la inspección de la pila
  - ⋆ usando interpretación abstracta de stacks de llamadas y *model checking* (Jensen)
  - ⋆ *constraint-based type-checking* (Pottier, Skalka, Smith)
- Análisis estáticos de la política de seguridad del control de accesos de JavaCard con interpretación abstracta y *model checking* (Chugonov).
- Semánticas formales y teoría ecuacionales para asegurar la inspección del stack (Gordon y Fournet).

Formalizando Java

Se puede dividir en diferentes aspectos:

- Semántica de Java
- Semántica del bytecode de Java
- Compilador Java->JVM (Notar que Java!=JVM, el proceso de compilación no es totalmente abstracto)
- Máquina virtual JVM

Alguno de los impedimentos que dificultan la formalización son:

- La referencia oficial de SUN no es formal, y en algunas partes resulta inconsistente, incompleta y ambigua[16]. Además, ni la propia implementación de referencia es consistente con la referencia oficial.

- Hay características que dependen del sistema operativo y versión de la JVM, como el comportamiento de los threads. Otras, como la recolección de basura, un preciso modelo de memoria y la carga de clases dinámica son difíciles de modelar.

Modelos semánticos a utilizar:

- Máquinas de estado abstractas **(ASM)**: lenguaje algorítmico para especificar y razonar sobre secuencias computacionales. Busca implementar cualquier algoritmo a su nivel de abstracción natural.

- Axiomática **(SM)**: compuesta de conjuntos de aserciones sobre las propiedades de un sistema y cómo van evolucionando a lo largo del tiempo. Concretamente, serían las pre-condiciones, post-condiciones e invariantes del código.

- Reescritura de contexto **(RC)**.

- De continuación o monádicas **(SC)**: basado en conceptos de lenguajes de programación funcional.
- Denotacional **(SD)**: busca recoger la semántica de un lenguaje de programación en base a funciones matemáticas de las cuales se pueden derivar demostraciones utilizando herramientas matemáticas , como la teoría de dominio. Recoge toda la semántica de un lenguaje imperativo bajo una firme base matemática.
- Natural **(SN)**: extensión de la semántica estructural-operacional como formalismo para expresar semánticas.
- Operacional **(SO)**: conjuntos de reglas recogiendo cómo evolucionará el estado de un sistema mientras ejecuta un programa.
- Estructural-operacional **(SOS)**: marco para dar semánticas operacionales a lenguajes de programación y especificación.
- Inclusiones semánticas en **(HOL)**.

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Consistencia en lóg. Hoare		[4]	$\times$	$\times$	$\surd$	SO	$\surd$
Estudiar semántica		[10]	$\surd$	$\surd$	$\times$	SD	$\times$
Especificaciones ejecutables	Centaur[33]	[12]	$\times$	$\times$	$\surd$	SN+SOS	$\times$
Estudiar semántica		[17]	$\surd$	$\times$	$\times$	SD	$\times$
Estudiar semántica		[21]	$\times$	$\surd$	$\surd$	ASM	$\times$
Estudiar semántica		[31]	$\times$	$\surd$	$\surd$	SOS	$\surd$
Estudiar semántica		[30]	$\times$	$\times$	$\surd$	SD	$\times$
Verificación		[45, 46]	$\times$	$\surd$	$\times$	SN+SOS	$\surd$
Verificación	SPIN[153]	[50]	$\surd$	$\surd$	$\surd$	embed	$\surd$
Consistencia de tipo		[55, 56, 58, 59]	$\times$	$\surd$	$\times$	RC	$\times$
Estudiar semántica		[54, 60]	$\times$	$\times$	$\times$	RC	$\surd$
Estudiar semántica		[57]	$\surd$	$\times$	$\times$	SA	$\surd$

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Estudiar semántica		[73]	$\times$	$\times$	$\times$	SN	$\times$
Modelo de memoria		[76]	$\times$	$\times$	$\surd$	SA	$\surd$
Verificación	SPIN[153]	[83]	$\surd$	$\times$	$\surd$	embed	$\surd$
Consistencia de tipo		[90]	$\times$	$\times$	$\times$	SN	$\surd$
Cálculo formal		[91]	$\times$	$\times$	$\times$	SOS	$\surd$
Verificación	PVS[139]	[95]	$\surd$	$\surd$	$\times$	embed	$\surd$
Verificación		[96]	$\times$	$\times$	$\times$	SOS	$\times$
Estudiar seguridad		[99]	$\times$	$\times$	$\surd$	HOL	$\times$
Verificación	Esc/Java[61]	[109]	$\surd$	$\surd$	$\surd$	embed	$\surd$
Modelo de memoria		[116]	$\times$	$\times$	$\surd$	Algebraica	$\times$
Modelo de memoria		[118]	$\times$	$\times$	$\surd$	SO	$\times$
Consistencia de tipo	Isabelle/HOL[92]	[127]	$\times$	$\times$	$\times$	SM	$\surd$

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Consistencia lóg. Hoare	Isabelle/HOL[92]	[129]	$\times$	$\times$	$\times$	SM	$\surd$
Inspección stack		[150]	$\times$	$\times$	$\times$	SO	$\times$
Semántica y verificación	AsmGofer[13]	[157]	$\surd$	$\surd$	$\surd$	ASM	$\surd$
Consistencia de tipos	DECLARE	[162]	$\times$	$\surd$	$\times$	SOS	$\surd$
Verificación	PVS,I/HOL[92, 139]	[168]	$\surd$	$\surd$	$\times$	embed	$\surd$
Consistencia de tipos	Isabelle/HOL[92]	[175]	$\times$	$\surd$	$\times$	SN	$\surd$
Consistencia, completitud	Isabelle/HOL[92]	[174]	$\times$	$\surd$	$\times$	SA	$\surd$
Estudiar semántica		[177]	$\times$	$\surd$	$\surd$	ASM	$\times$
Inspección stack		[179]	$\times$	$\times$	$\times$	Lóg. Modal	$\surd$
Inspección stack		[178]	$\times$	$\times$	$\times$	Traducción	$\surd$

Formalizando la máquina virtual

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Estudiar semánticas		[15]	√	√	×	SO	×
Estudiar semánticas		[16]	×	×	×	SO	×
Estudiar semánticas		[19]	×	×	√	SO	√
Estudiar semánticas		[24]	√	×	×	ASM	×
Consistencia de tipo	SpecWarec[152]	[40]	√	×	×	embed	×
Estudiar semánticas	ACL2[5]	[41]	√	×	×	embed	×
Estudiar semánticas	PVS[139]	[48]	√	×	×	embed	√
Corrección convertidor	Coq[43]	[51]	×	×	×	SO	√
Estudiar semánticas		[64]	√	×	×	HOL	×

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Estudiar semánticas		[65]	$\times$	$\surd$	$\times$	SO	$\times$
Estudiar semánticas		[67, 68]	$\times$	$\surd$	$\times$	SO	$\surd$
Consistencia de tipo	SpecWare[152]	[74]	$\surd$	$\times$	$\times$	embed	$\times$
Estudiar semánticas		[78]	$\times$	$\surd$	$\times$	SO	$\surd$
Estudiar semánticas		[165, 166]	$\surd$	$\times$	$\times$	SO	$\surd$
Estudiar semánticas	LETOS[113]	[80]	$\surd$	$\times$	$\times$	SO	$\times$
Estudiar semánticas		[97]	$\surd$	$\times$	$\times$	SO	$\times$
Consistencia de tipo	Haskell[82]	[98]	$\times$	$\surd$	$\times$	SC	$\times$
Consistencia de tipo	Isabelle[92]	[102]	$\times$	$\times$	$\times$	SM	$\surd$
Corrección convertidor	Atelier B[11]	[107]	$\times$	$\times$	$\times$	embed	$\surd$
Verificación	ACL2[5]	[122]	$\times$	$\times$	$\times$	embed	$\surd$
Estudiar semánticas		[128]	$\times$	$\surd$	$\times$	SC	$\times$

Objetivo	Herramienta	Ref.	Cla	Exc.	MT	Seman	Demos.
Corrección convertidor	SMV[151]	[14]	×	√	×	SMV	√
Corrección convertidor	Isabelle/HOL[92]	[137]	√	×	×	SO	√
Corrección convertidor	Isabelle/HOL[92]	[138]	√	×	×	SO	√
Corrección convertidor		[140]	×	√	×	SO	×
Corrección convertidor		[142]	√	×	×	SO	√
Corrección convertidor	Atelier B[11]	[143]	×	√	×	SO	√
Corrección convertidor		[146]	×	×	×	SM	√
Estudiar semánticas		[159]	×	√	×	SO	√
Estudiar semánticas		[160]	√	√	×	DS	×
Estudiar seguridad		[179]	×	×	×	HOL	√
Consistencia de tipos	Haskell[82]	[180]	√	√	×	SC	√

Caso concreto: Stärk,Schmid,Börger[157]

La formalización más extensa y completa de Java, la máquina virtual, compilador, verificador, cargador y el SecurityManager. Provee de la implementación independiente más rigurosa para el análisis y documentación mediante la verificación matemática y la validación experimental.

Realizada utilizando ASMs, concretamente con la herramienta ASMGofer/TkGofer, permite:

- ejecutar programas a partir de su código fuente Java (en diferentes tipos de VM)
- compilar programas Java a bytecode JVM

- ejecutar los programas en bytecode compilados (en diferentes tipos de VM)

Ya que se sigue un diseño modular, divide al lenguaje Java en las siguientes partes, cada una de ellas con una definición tanto estática como dinámica:

- $\mathbf{Java_I}$ = núcleo imperativo de Java
  - ⋆ *Statements* y expresiones sobre los datos primitivos de Java.
- $\mathrm{Java}_C = \mathbf{Java_I} + \textit{clases como módulos}$
  - ⋆ Inclusión de métodos, campos e inicializadores estáticos, junto a interfaces
- $\mathbf{Java_O} = \mathrm{Java}_C + \text{objetos y arrays}$
  - ⋆ Se añaden campos y métodos de instancia, la creación de nuevas instancias, sobrecarga de métodos, punteros nulos, comprobaciones y cambios de tipo.

- $\mathbf{Java_E} = \mathbf{Java_O}$+ excepciones
  - ★ Se concreta la interacción de las excepciones con las instrucciones *break*, *continue*, *return* y la inicialización de clases e interfaces. Sólo se consideran las excepciones en tiempo de ejecución y las definidas por el usuario; los errores pertenecen a la JVM, y por lo tanto, quedan fuera de Java.

- $\mathbf{Java_T} = \mathbf{Java_E}$+ threads con consistencia en la sincronización
  - ★ soporte para la ejecución de múltiples tareas con memoria compartida y local: métodos *start*, *interrupt*, *wait* y *notify*, además de sincronización, espera y notificación utilizando locks. El modelo de memoria se simplifica para no complicar la definición.

$$\text{execJava} = \text{execJava}_I \text{execJava}_C \text{execJava}_O \text{execJava}_E \text{execJava}_E \text{execJava}_T$$

$$
\begin{array}{rcl}
\text{execJavaThread} & \equiv & \text{choose } q \text{ in } dom(exec), runnable(q) \\
& \text{if} & q = thread \; and \; exec(q) = Active \\
& \text{then} & execJava \\
& \text{else} & \text{if } exec(q) = Active \text{ then} \\
& & cont(thread) := (frames, (method, retbody, pos, locals)) \\
& & thread := q \\
& & run(q)
\end{array}
$$

Theorem 1. *Java is* ***type-safe****, es decir, cuando un programa Java correctamente tipado se ejecuta:*

- las valores en tiempo de ejecución de los elementos de los campos y arrays, estáticos o de instancia, son compatibles con los tipos declarados
- las referencias a los objetos se mantienen en el heap

- las posiciones en tiempo de ejecución satisfacen las restricciones en tiempo de ejecución

- la evaluación de expresiones y variables devueltas mantienen el tipo compatible en tiempo de compilación

- se mantiene el tipo en tiempo de compilación de los *jmp*, *return, exec*, ...

- se mantiene la integridad del stack

Proof. Mediante inducción en el número de pasos de la ejecución de la máquina de estados abstracta Java.

Theorem 2. *Correctitud de la sincronización de hilos en Java*

- Los hilos *runtime* son hilos válidos.

- Si el estado de un hilo es “*Not Started*”, el hilo no está sincronizado con ningún objeto ni el conjunto de espera de ningún objeto.
- Si el estado de un hilo está sincronizándose, entonces el hilo todavía no está sincronizado en el objeto por el que compite.
- Si un hilo está sincronizado en un objeto, entones el objeto es una referencia válida en el *heap*.
- Si un hilo está esperando a un objeto, entonces está sincronizado y en el conjunto de espera de un objeto.
- Si se ha notificado a un hilo de un objeto, entonces éste ya no está en el conjunto de espera de un objeto: todavía sigue sincronizado en el objeto, pero ya no mantiene el *lock* sobre el objeto.
- Un hilo no puede estar en el conjunto de espera de dos objetos diferentes.

- Si un hilo ha terminado, ya no mantiene *lock* sobre ningún objeto.
- Si un hilo mantiene el *lock* de un objeto, entonces el contador del *lock* del objeto es el número de veces que aparece el objeto en la lista de objetos sincronizados en el hilo.
- No es posible que dos hilos diferentes mantenga el *lock* del mismo objeto al mismo tiempo.
- Si el contador del *lock* de un objeto es mayor que 0, entonces existe un hilo que mantiene el *lock* sobre el objeto.

Proof. Usando inducción en el número de pasos de la ejecución de la máquina de estados abstracta Java.

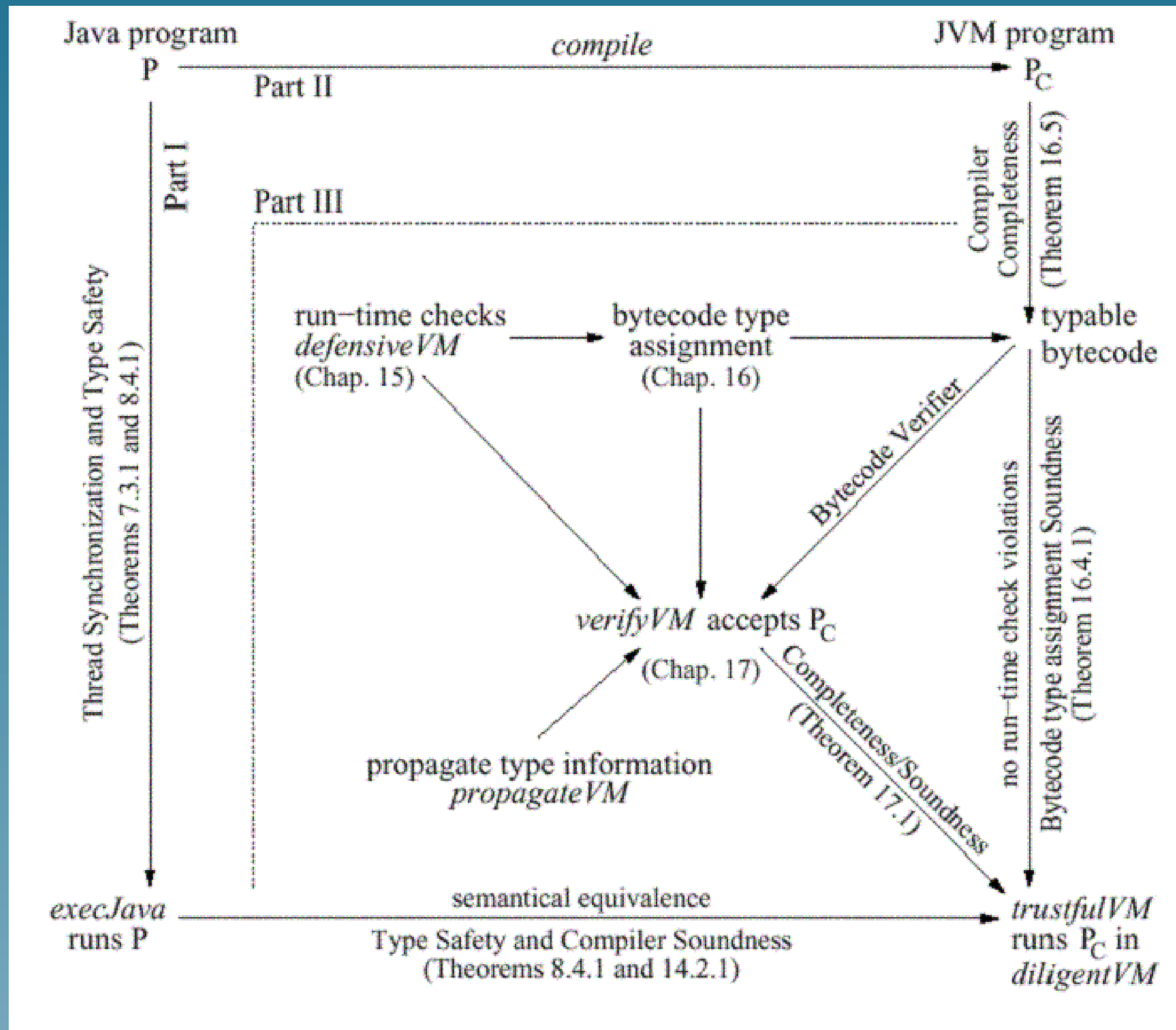
La máquina virtual Java se dividide en las siguientes partes:

- $\mathbf{JVM_I}$ = núcleo imperativo de la máquina virtual segura ($\mathrm{execVM_I}$), compilación de $\mathbf{Java_I}$
- $\mathrm{JVM_C}$= $\mathbf{JVM_I}$ + clases ($\mathrm{execVM_C}$), compilación de $\mathbf{Java_C}$
  - ⋆ $\mathrm{switchVM_C}$= manejo del frame stack actual por invocación de método, retorno o inicialización de clase
- $\mathbf{JVM_O}$ = $\mathrm{JVM_C}$ + objetos ($\mathrm{execVM_O}$), compilación de $\mathbf{Java_O}$
- $\mathbf{JVM_E}$ = $\mathbf{JVM_O}$ + excepciones ($\mathrm{execVM_E}$), compilación de $\mathbf{Java_E}$
  - ⋆ $\mathrm{switchVM_E}$ = manejo del stack para la captura de excepciones
- $\mathrm{JVM_N}$= $\mathrm{JVM_E}$+ métodos nativos del JDK (execVM_N)
- $\mathbf{JVM_T}$ = $\mathrm{JVM_E}$+ hilos concurrentes ($\mathrm{execVM_T}$), compilación $\mathbf{Java_T}$
- $\mathrm{execVM_D}$ y $\mathrm{switchVM}_D$= carga y linkado de clases

check_I, check_C, check_O, check_E, check_N, check_T= $lascomprobacionesestánimplement$
Además, existe una descomposición de la JVM según su papel dentro la seguridad:

- **trustfulVM**: define la funcionalidad de ejecución con *execVM* y *switchVM*
- **defensiveVM**: define las restricciones a comprobar utilizando *check* y llamando a *trustfulVM* para la ejecución. Antes de la ejecución de cada paso, se comprueban las restricciones estructurales que componen la funcionalidad del verificador, garantizando una ejecución segura.
- **diligentVM**: realiza la restricciones en tiempo de linkado utilizando *verifyVM* y llamando a *trustfulVM para la ejecución*
- **verifyVM**: utiliza a *check*, *propagateVM*, *succ*
- **dynamicVM**: carga dinámica y linkado de clases.

Las relaciones entre éstas se ilustran en el siguiente gráfico:



Theorem 3. *Correctitud de la compilación de Java a bytecode JVM: para programas P arbitrarios, son equivalentes la ejecución de P en Java y la ejecución de P compilado en la $trustfulVM$, incluyendo el correcto manejo de las excepciones.*

Proof. Inducción en el número de pasos de la ejecución de las máquinas de estado abstractas Java y JVM, usando el teorema 1.

□

Theorem. *Si los programas se ejecutan correctamente, los efectos semánticos de la ejecución de la* ***defensiveVM*** *y la* ***trustfulVM*** *son idénticos.*

Theorem 4. *Si un programa satisface las condiciones de asignación de tipos impuestas por* **check**, ***defensiveVM*** *ejecuta el programa sin violar ninguna comprobación en tiempo de ejecución*

Proof. Inducción en el número de pasos de la ejecución de la **defensiveVM**.

Theorem 5. *El bytecode generado por la compilación de un programa Java válido contiene asignaciones de tipo.*

Proof. Inducción usando un compilador certificante que asiga a cada instrucción del bytecode un frame tipado, el cual se puede demostrar que constituye una asignación de tipo para el código compilado.

Theorem 6. *El verificador de bytecode es consistente, es decir, para cualquier programa P, el verificador o rechaza a P o durante la verificación se satisfacen las condiciones de asignación de tipos para P.*

Theorem 7. *El verificador de bytecode es completo, es decir, si un programa P contiene la asignación de un tipo, entonces el verificador no rechaza a P y calcula la asignación de tipo más específica.*

Theorem 8. *El compilador genera código verificable.*

Proof.

Corollary 1. *Bajo unas restricciones concretas, cualquier programa Java bien formado y bien tipado,*

- tras su correcta compilación
- pasa el verificador
- y es ejecutado en la JVM
  - ★ sin violar ninguna comprobación en tiempo de ejecución
  - ★ y de forma correcta según la semántica Java

Preguntas abiertas

- Posibilidad de extender el análisis estático del bytecode para acotar el uso máximo de recursos. Especialmente interesante para JavaCards, donde no hay recolector de basura.
- Verificaciones completas del API[87, 112].
- Análisis estático para el control de accesoc[96, 135, 176] y flujo de la información[84, 3, 134, 171, 172].
- Completa verificación de la máquina virtual y el lenguaje.
- Verificación de un JIT.
- Extender los métodos anteriores a .NET

Referencias

[1] M. Abadi. “Protection in Programming-language translations”. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, 19-34, 1999.

[2] M. Abadi, M. Burrows, B. Lampson, G. Plotkin. “A calculus for access control in distributed systems”. *ACM Transactions on Programming Languages and Systems*, 15(4):706-734, 1993.

[3] M. Abadi, A. Banerjee, N. Heintze, J.G. Riecke. “A core calculus of dependency”. *26th Symposium Principles of Programming Languages*, 147-160, 1999.

[4] E. Ábrahám-Mumm, F.S. de Boer. “Proof outlines for threads in Java”. *11th International Conference on Concurrency Theory*, LNCS 1877, 229-242, 2000.

[5] ACL2. http://www.cs.utexas.edu/users/moore/acl2/

[6] O. Agesen, D. Detlefs, J.E.B. Moss. “Garbage collection and local variable type-precision and liveness in Java virtual machines”. *Programming Language Design and Implementation*, 269-279, 1998.

[7] O. Agesen, S.N. Freund, J.C. Mitchell. “Adding type parameterization to the Java(TM) language”. *SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 49-65, 1997.

[8] J. Aldrich, C. Chambers, E. Sirer, S. Eggers. “Static analyses for eliminating unnecessary synchronization from Java programs”. *Static Analysis Symposium*, 19-38, 1999.

[9] J. Alves-Foss, editor. “Formal Syntax and Semantics of Java”. LNCS 1523, 1999.

[10] J. Alves-Foss, F.S. Lam. “Dynamic denotational semantics of Java”. *Formal Syntax and Semantics of Java,* LNCS 1523, 201-240, 1999.

[11] Atelier-B.**http://http://www.atelierb.societe.com**

[12] L. Attali, D. Caromel, M. Russo. “A formal executable semantics for Java”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[13] AsmGofer. **http://www.tydo.de/AsmGofer**

[14] D. Basin, S. Friedrichs, J. Posegga, H. Vogt. “Java bytecode verification using model checking”. *11th International Conference on Computer Aided Verification*, LNCS 1663, 491-494, 1999.

[15] P. Bertelsen. “Semantics of Java byte code”. *Future Generation Computer Systems*, 16(7):841-850, 2000.

[16] P. Bertelsen. “Dynamic semantics of Java byte code”. *Future Generation Computer Systems*, 16(7):841-850, 2000.

[17] P. Bertelsen, S. Anderson. “The semantics of a core language derived from Java”. Technical Report, Technical University of Denmark, 1996.

[18] Y. Bertot. “Formalizing a JVML verifier for initialization in a theorem prover”. *Proc. Computer Aided Verification (CAV’01),* LNCS 2102, 14-24, 2001.

[19] G. Bigliardi, C. Lanece. “A type system for JVM threads”. Technical Report UBLCS-2000-06, University of Bologna, 2000.

[20] J. Bogda, U. Hölzle. “Removing unnecessary synchronization in Java”. *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 34-46, 1999.

[21] E. Börger, W. Schulte. “Defining the Java virtual machine as platform for provably correct Java compilation”. *23rd International Symposium Mathematical Foundations of Computer Science*, LNCS 1450, 17-35, 1998.

[22] E. Börger, W. Schulte. “Initialization problems for Java”. *Software Concepts and Tools*, 20(4):175-179, 1999.

[23] E. Börger, W. Schulte. “A programmer friendly module definition of the semantics of Java”. *Formal Syntax and Semantics of Java*, LNCS 1523, 353-404, 1999.

[24] E. Börger, W. Schulte. “Modular design for the Java virtual machine architecture”. *Architecture Design and Validation Methods*, 297-357, 2000.

[25] E. Börger, W. Schulte. “A practical method for specification and analysis of exception handling - a Java/JVM case study”. *IEEE Transactions on software engineering*, 26(9):872-887, 2000.

[26] E. Börger, J. Schmid. “Defining the Java Virtual Machine as platform for provably correct Java compilation”. *MFCS’98*, LNCS 1450, 17-35, 1998.

[27] G. Bracha. “A critique of security and dynamic loading in Java: a formalisation”. Sun Java Software, 1999.

[28] P. Brisset. “Vers un vérifier de bytecode Java certifié”. Seminar at Ecole Normale Supérieure, Paris, 1998.

[29] L. Casset and J.L. Lanet. “How to formally specify the Java bytecode semantics using the B method”. *Formal techniques for Java Programs, ECOOP Workshops*, 104-105, 1999.

[30] P. Cenciarelli. “Towards a modular denotational semantics of Java”. *Formal techniques for Java Programs, ECOOP Workshops*, 105, 1999.

[31] P. Cenciarelli, A. Knapp, B. Reus, M. Wirsing. “An event based structural operational semantis of multi threaded Java”. *Formal Syntac and Semantics of Java*, LNCS 1523, 157-200, 1999.

[32] P. Cenciareli, A. Knapp, B. Reus, M. Wirsins. “An event based structural operational semantics of multi threaded Java”. *Formal Syntax and Semantics of Java*, LNCS 1523, 157-200, 1999.

[33] The Centaur System. **http://www-soap.inria.fr/croap/centaur/centaur.html**

[34] Z. Chen. “Java Card Technology for Smart Cards: Architecture and programmer’s guide”. Adisson-Wesley, Reading, Massachusetts, 2000.

[35] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, S. Midkiff. “Escape analysis for Java”. *SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications,* 1-19, 1999.

[36] G. Chugunov, L.A. Fredlund, D. Gurov. “Model checking multi-applet Java Card applications”. *Smart Card Research and Advanced Applications Conference*, 2002.

[37] A. Coglio. “Simple verification technique for complex Java bytecode subroutines”. *4th ECOOP Workshop on Formal Techniques for Java-like Programs*, 2002.

[38] A. Coglio. “Improving the official specification of Java bytecode verification”. *Concurrency and Computation: Practice and Experience*, 15(2):155-179, 2003.

[39] A. Coglio, A. Goldberg. “Type safety in the JVM: some problems in JDK 1.2.2 and proposed solutions”. Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, 2000.

[40] A. Coglio, A. Goldberg, Z. Qian. “Towards a provably-correct implementation of the JVM bytecode verifier”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[41] R. M. Cohen. “The defensive Java virtual machine specification version 0.5”. Technical report, Computational Logic Inc., Austin, Texas, 1997. **http://www.cli.com/software/djvm/**

[42] R. M. Cohen. “Formal underpinnings of Java: some requirements”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[43] The Coq Proof Assistant. **http://coq.inria.fr/**

[44] C. Colby, P. Lee, G.C. Necula, F. Blau, K. Cline, M. Plesko. “A certifying compiler for Java”. *SIGPLAN Conference on Programming Language Design and Implementation*, 95-107, 2000.

[45] E. Coscia, G. Reggio. “An operational semantics for Java”. Technical report, DISI, University of Genova, Italy, 1998.

[46] E. Coscia, G. Reggio. “A proposal for a semantics of a subset of Multi-Threaded “good” Java programs”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[47] K. Crary, S. Weirich. “Resource bound certification”. *27th International Conference on Principles of programming languages*, 184-198, 2000.

[48] D. Dean. “The security of static typing with dynamic linking”. *4th International Conference on Computer and Communications Security*, 18-27, 1997.

[49] D. Dean, E. W. Felten, D.S. Wallach. “Java security: from HotJava to Netscape and beyond”. *Symposium on Security and Privacy*, 190-200, 1996.

[50] C. Demartini, R. Iosif, R. Sisto. “Modeling and validation of Java multithreaded applications using SPIN”. *4th SPIN Workshop*, 1998.

[51] E. Denney, Th. Jensen. “Correctness of Java card method lookup via logical relations”. *9th European Symposium on programming*, LNCS 1782, 104-118, 2000.

[52] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe. “Extended static checking”. SRC Research report 159, Compaq Systems Research Center, Palo Alto, California, 1998.

[53] S. Diehl. “A formal introduction to the compilation of Java”. *Software - practice and experience*, 28(3):297-327, 1998.

[54] S. Drossopoulou. “Towards an abstract model of Java dynamic linking and verification”. *ACM SIGPLAN Workshop on Types in Compilation*, 19, 2000.

[55] S. Drossopoulou, S. Einsenbach. “Java is type safe - provably”. *11th European Conference on Object Oriented Programming, ECOOP*, LNCS 1241, 389-418, 1997.

[56] S. Drossopoulou, S. Eisenbach. “Describing the semantics of Java and proving type soundness”. *Formal Syntax and Semantics of Java*, LNCS 1523, 41-82, 1999.

[57] S. Drossopoulou, S. Eisenbach, D. Wragg D. “A fragment calculus - towards a model of separate compilation, linking and binary compatibility”. *Logic in Computer Science*, 147-156, 1999.

[58] S. Drossopoulou, S. Einsenbach, S. Khurshid. “Is theJava type system sound?”, *Theory and Practice of object systems*, 5(1):3-24, 1999.

[59] S. Drossopoulou, T. Valkevych. “Java Exceptions Throw no Surprises”. Department of Computing, Imperial College, London, 2000.

[60] S. Drossopoulou, D. Wragg, S. Eisenbach. “What is Java binary compatibility?” *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 341-361, 1998.

[61] The Extended Static Checker for Java (ESC/Java). **http://research.compaq.com/SRC/esc/**

[62] M. Flatt, S. Krisnamurthi, M. Felleisen. “A programmer’s reduction semantics for classes and mixins”. *Formal Syntax and Semantics of Java*, LNCS 1523, 241-270, 1999.

[63] W. J. Fokkink, C. Verhoef. “A conservative look at operational semantics with variable binding”. *Information and Computation*, 146(1):24-54, 1998.

[64] P.W.L. Fong and R.D. Cameron. "Proof linking: an architecture for modular verification of dynamically-linked mobile code". *6th SIGSOFT International Symposium on the Foundations of Software Engineering, 222-230, 1998.*

[65] S.N. Freund. "The costs and benefits of Java bytecode subroutines". *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.

[66] S.N. Freund, J.C. Mitchell. "A formal framework for the java bytecode language and verifier". *Object-Oriented Programming Systems, Languages and Applications 1999*, 147-166, 1999.

[67] S.N. Freund, J.C. Mitchell. "A type system for object initialization in the Java bytecode language". *Conferences on Object-Oriented Programming Systems, Languages and Applications*, 310-328, 1998.

[68] S.N. Freund, J.C. Mitchell. "A formal framework for the Java bytecode language and verifier". *SIGPLAN Conference on Object-*

Oriented Programming, Systems, Languages and Applications, 147-166, 1999.

[69] S.N. Freund, J.C. Mitchell. “Specification and verification of Java bytecode subroutines and exceptions”. Technical Report CS-TN-99-91. Stanford University, 1999.

[70] S.N. Freund, J.C. Mitchell. “A type system for the Java bytecode language and verifier”. *Journal of Automated Reasoning, Special issue on bytecode verification*.

[71] J. S. Fritzinger, M. Mueller. *Java Security*. Sun Micro Systems Inc., Mountain View, California, 1996.

[72] E. Gagnon, L. Hendren. “Intra-procedural inference of static types”. Technical Report 1999-1, Sable Group, McGill University, Montréal, Canada, 1999.

[73] S. Glesner, W. Zimmermann. “Using many-sorted natural semantics to specify and generate semantic analysis”. *TC2 WG2.4 Working Conference on Systems Implementation 2000: Languages, Methods and Tools*, 249-262, 1998.

[74] A. Goldberg. “A specification of Java loading and bytecode verification”. *5th Conference on Computer and Communication Security*, 49-58, San Francisco, California, 1998.

[75] L. Gong. “Secure Java class loading”. IEEE-Internet Computing, 2(6):56-61, 1998.

[76] A. Gontmaker, A. Schuster. “Java consistency: non-operational characterizations for Java memory behavior”. *ACM Transactions on Computer Systems*, 2000.

[77] J. Gosling, B. Joy, G. Steele, G. Bracha. “The Java Language Specification”. Addison Wesley, Reading, Massachusetts, second edition, 2000.

[78] M. Hagiya, A. Tozawa. “On a new method for dataflow analysis of Java virtual machine subroutines”. *International Static Analysis Symposium*, LNCS 1503, 17-32, 1998.

[79] P.H. Hartel, L.A.V. Moreau. “Formalizing the safety of Java, the Java virtual machine and Java Card”. *ACM Computing Surveys*, 33(4):517:558, 2001.

[80] P.H. Hartel, M.J. Burtel, M. Levy. “The operational semantics of a Java secure processor”. *Formal Syntax and Semantics of Java*, LNCS 1523, 313-352, 1999.

[81] P.H. Hartel and E. de Jong. “A programming and a modelling perspective on the evaluation of Java card implementations”. *Java Card Workshop Proceedings*, 2000.

[82] The Haskell Programming Language. **http://www.haskell.org**

[83] K. Havelund, T. Pressburger. “Model checking Java programs using pathfinder”. *Software Tools for Technology Transfer*, 2(4), 1999.

[84] N. Heintze, J.G. Riecke. “The SLam Calculus: programming with secrecy and integrity”. *25th Symposium on Principles of Programming Languages*, 365-377, 1998.

[85] L. Henrio, B. Serpette. “A framework for bytecode verifiers: application to intra-procedural continuations”. Research report, INRIA, 2001.

[86] M. Huisman. “Reasoning about Java programs in higher order logic with PVS and Isabelle”, PhD thesis, University of Nijmegem, The Netherlands, 2001.

[87] M. Huisman, B. Jacobs, J. van den Berg. “A case study in class library verification: Java’s vector class”. *Software tools for technology transfer*, 2001.

[88] J. Hummel, A. Azavedo, D. Kolson, A. Nicolau. “Annotating the Java bytecodes in support of optimization”. *Concurrency: practice and experience*. 9(11):1003-1016, 1997.

[89] A. Igarashi, B. Pierce. “On inner classes”. *7th International Workshop on Foundations of Object-Oriented Languages*, 2000.

[90] A. Igarishi, B. Pierce. “On inner classes”. *7th International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2000.

[91] A. Igarishi, B. Pierce, P. Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". *Conference on Object-Oriented Programming, Systems, Languages, Applications*, 132-146, 1999.

[92] Isabelle/HOL. **http://www.cl.cam.ac.uk/Research/HVG/Isabelle/**

[93] B. Jacobs. "A formalisation of Java's exception mechanism". *10th European Symposium on programming*, LNCS, 2001.

[94] B. Jacobs, E. Poll. "A monad for basic Java semantics". *Algebraic Methodology and Software Technology*, LNCS 1816, 150-164, 2000.

[95] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, H. Tews. "Reasoning about classes in Java (preliminary report)". *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 329-340.

[96] T. Jensen, D. Le Métayer, T. Thorn. “Verification of control flow based security properties”. *IEEE Symposium on Security and Privacy*, 1999.

[97] T. Jensen, D. Le Métayer, T. Thorn. “Security and dynamic class loading in Java: a formalisation”. *International Conference on Computer Languages*, 4-15, 1998.

[98] M. Jones. “The functions of Java bytecode”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[99] L. Kassab, S. Greenwald. “Towards formalizing the Java security architecture in JDK 1.2”. *European Symposium on Research in Computer Security*, LNCS 1485, 191-207, 1998.

[100] T. Kistler, M. Franz. “A tree-based alternative to Java bytecodes”. Technical report 96-58, Department of Information and Computer Science, University of California, Irvine, 1996.

[101] G. Klein. “Verified Java bytecode verification”. PhD Thesis, Technische Universität München, 2003.

[102] G. Klein, T. Nipkow. “Verified lightweight bytecode verification”. *ECOOP 2000 Workshop on Formal Techniques for Java Programs*, 2000.

[103] G. Klein, T. Nipkow. “Verified bytecode verifiers”. *Theoretical Computer Science: 2002*, 13:1133-1151, 2001.

[104] G. Klein, M. Wildmoser. “Verified bytecode subroutines”. *Journal of Automated Reasoning, Special Issue on Bytecode Verification*.

[105] T.B. Knoblock, J. Rehof. “Type elaboration and subtype completion for Java bytecode”. *27th International Conference on Principles of Programming Languages*, 228-242, 2000.

[106] D. Kozen. “Efficient Code Certification”. *Cornell University*, 1998.

[107] J.-L. Lanet, A. Request. “Formal proof of smsrt card applets correctness”. *3rd International Conference on Smartcard research and advanced application*, LNCS 1820, 85-97, 1998.

[108] Last Stage of Delirium Research Group. “Java and Java Virtual Machine Security. Vulnerabilities and their Exploitation Techniques”. **http://lsd-pl.net**

[109] K.R.M. Leino, J.B. Saxe, R. Stata. “Checking Java Programs via guarded commands”. *SRC Research Report 1999-02*, 1999.

[110] X. Leroy. “Bytecode verification for Java smartcard”. *Software Practice & Experience*, 32:319-340, 2002.

[111] X. Leroy. “Java bytecode verification: algorithms and formalizations”. *Journal of Automated Reasoning, Special Issue on Bytecode Verification*.

[112] X. Leroy, F. Rouaix. "Security properties of typed applets", *Secure Internet Programming - Security Issues for Mobile and Distributed Objecrs*, LNCS 1603, 147-182, 1999.

[113] LETOS (A Lightweight Execution Tool for Operational Semantics). **http://www.ecs.soton.ac.uk/ phh/letos.html**

[114] S. Liang, G. Bracha. "Dynamic class loading in the Java virtual machine". *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 36-44, 1998.

[115] T. Lindholm, F. Yellin. "The Java Virtual Machine Specification". Addison Wesley, Reading, Massachusetts, 1996.

[116] J. Maessen, Arvind, X. Shen. "Improving the Java memory model using CRF". *SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1-12, 2000.

[117] D. Malkhi, M. Reiter. “Secure execution of Java applets using a remote playground”. *IEEE Transactions on Software Engineering*, 2000.

[118] J. Manson, W. Pugh. “Semantis of multithreaded Java”. Technical report, Department of Computer Science, University of Maryland, January 2001.

[119] G. McGraw, E.W. Felten. “Securing Java: Getting down to business with mobile code”. John Wiley and Sons, Chichester, UK, second edition, 1999.

[120] J. Meyer, T. Downing. “Java Virtual Machine”. O’Reilly, 1997.

[121] M. Montgomery, K. Krishna. “Secure object sharing in Java card”. *USENIX Workshop on Smartcard Technology*, 119-127, 1999.

[122] J.S. Moore. “Proving theorems about Java-like bytecode”. *Correct System Design - Recent Insights and Advances*, LNCS 1710, 139-162, 1999.

[123] S. Motré. “Formal model and implementation of the Java card dynamic security policy”. *Approaches Formelles dans l’Assistance au Développment de Logiciels*, 2000.

[124] G.C. Necula. “Proof-carrying code”. *24th Symposium on Principles of Programming Languages*, 106-119, 1997.

[125] H.R. Nielson, F. Nielson. “Semantics with applications: a formal introduction”. John Wiley and Sons, Chichester, UK, 1991.

[126] T. Nipkow. “Verified bytecode verifiers”. *Foundations of Software Science and Computation Structures*, LNCS 2030, 347-363, 2001.

[127] T. Nipkow, D. von Oheimb. "Java_{light} is Type-Safe - definitely". *25th International Conference on Principles of Programming Languages*, 161-170, 1998.

[128] R. O'Callahan. "A simple, comprehensive type system for Java bytecode subroutines". *26th International Conference on Principles of programming languages*, 70-78, 1999.

[129] A. Poetzsch-Heffter, P. Muller. "A programming logic for sequential Java". *8th European Symposium on programming*, LNCS 1576, 162-176, 1999.

[130] E. Poll, J. van den Berg, B. Jacobs. "Specification of the JavaCard API in JML". *4th Smartcard Research and Advanced Application Conference*, 135-154, 2000.

[131] E. Poll, J. van den Berg, B. Jacobs. “Formal specification of the JavaCard API in JML: the APDU class”. *Computer Networks Magazine*, 2001.

[132] J. Posegga, H. Vogt. “Java bytecode verification using model checking”. *Workshop Fundamental Underpinnings of Java*, 1998.

[133] J. Posegga, H. Vogt. “Bytecode verification for Java smart cards based on model checking”. *European Symposium on Research in Computer Security*, LNCS 1485, 175-190, 1998.

[134] F. Pottier, V. Simonet. “Information flow inference for ML”. *29th Symposium Principles of Programming Languages*, 319-330, 2002.

[135] F. Pottier, C. Skalka, S. Smith. “A systematic approach to static access control”. *Proceedings of the 10th European Symposium on Programming*, LNCS 2028, 30-45, 2001.

[136] W. Pugh. “The Java memory model is fatally flawed”. *Concurrency: practice and experience, 12(1):1-11*, 2000.

[137] C. Pusch. “Formalizing the Java virtual machine in Isabelle/HOL”. Technical Report TUM-I9816, Institut für Informatik, Technische Univ. München, 1998.

[138] C. Pusch. “Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL”. *5th Tools and Algorithms for Construction and Analysis of Systems*, LNCS 1579, 89-103, 1999.

[139] The Proof Verification System (PVS) Model. **http://pvs.csl.sri.com/**

[140] Z. Qian. “A formal specification of Java(tm) virtual machine instruction objects, methods and subroutines”. *Formal Syntax and Semantics of Java*, LNCS 1523, 271-312, 1999.

[141] Z. Qian. “Standard Fixpoint Iteration for Java Bytecode Verification”. Kestrel Institute, Palo Alto, CA 94304, 1999.

[142] Z. Qian, A. Goldberg, A. Coglio. “A formal specification of JavaTM class loading”. *SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, 325-336, 2000.

[143] A. Requet. “A B model for ensuring soundness of the Java card virtual machine”. *5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems*, 26-29, 2000.

[144] E. Rose. “Vérification de code d’octet de la machine virtuelle Java: formalisation et implatation”. PhD thesis, University Paris 7, September 2002.

[145] E. Rose. “Towards secure bytecode verification on a Java card”. Master’s thesis, DIKU, University of Copenhagen, 1998.

[146] E. Rose, K.H. Rose. "Lightweight bytecode verification". *OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.

[147] V. Saraswat. "Java is not type-safe". Technical report, AT&T, Florham Park, New Jersey, 1997. **http://www.research.att.com/ vj/bug.html**

[148] D.A. Schmidt. "Data flow analysis is model checking of abstract interpretations". *25th Symposium on Principles of Programming Languages*, 38-48, 1998.

[149] L. Shin, J.C. Mitchell. "Java bytecode modification and applet security". Technical report, Computer Science Department, Stanford University, 1998.

[150] Ch. Skalka, S. Smith. "Static enforcement of security with types". *5th SIGPLAN Conference on Functional Programming (ICFP)*, 34-45, 2000.

[151] The SMV System. **http://www.cs.cmu.edu/ modelcheck/smv.html**

[152] SpecWare. **http://www.specware.org/**

[153] The SPIN Model Checker. **http://spinroot.com/**

[154] V.C. Sreedhar, M. Burke, J.-D. Choi. “A framework for interprocedural analysis and optimization in the presence of dynamic class loading”. *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 176-207, 2000.

[155] R. Stärk. “Formal foundations of Java”. Course notes, University of Fribourh, 1999.

[156] R. Stärk. “Foundations of Java - Lecture Notes for Computer Science Students”. University of Fribourg, Switzerland, 1998.

[157] R. Stärk, J. Schmid, E. Börger. “Java and the Java Virtual Machine: Definition, Verification, Validation”. Springer-Verlag, Berlin, 2001.

[158] R.F. Stärk, J. Schmid. “Completeness of a bytecode verifier and a certifying Java-to-JVM compiler”. *Journal of Automated Reasoning*, *Special Issue on bytecode verification.*

[159] R. Stata, M. Abadi. “A type system for Java bytecode subroutines”. *25th International Conference of Principles of Programming Languages*, 149-160, 1998.

[160] K. Stephenson. “Towards an algebraic specification of the Java virtual machine”. *Prospects for hardware foundations. ESPRIT working group 8533, NADA - New Hardware Design Methods Survey Chapters*, LNCS 1546, 236-277, 1998.

[161] G.S. Stiles. “Safe and verifiable design of multithreaded Java programs with CSP and FDR”. *OOPSLA’98 Workshop on Formal Underpinnings of Java*, 1998.

[162] D. Syme. “Proving Java type soundness”. *Formal Syntax and Semantics of Java*, LNCS 1523, 83-118, 1999.

[163] K. Thompson. “Reflections on Trusting Trust”. *Communication of the ACM, Vol. 27, No. 8,* August 1984.

[164] T. Thorn. “Programming languages for mobile code”. *ACM Computing Surveys*, 29(3):213-239, 1997.

[165] A. Tozawa, M. Hagiya. “Careful analysis of type spoofing”. *JIT’99 Java-Informations-Tage*, 290-296, 1999.

[166] A. Tozawa, M. Hagiya. “Formalization and analysis of class loading in Java”. Technical report, Graduate School of Science, University of Tokyo, 1999.

[167] Trusted Logic. “Off-card bytecode verifier for Java Card”. Sun’s Java Card Development Kit, 2001.

[168] J. an den Berg, M. Huisman, B. Jacobs. E. Poll. “A type-theoretic memory model for verification of sequential Java programs”. *Recent Trends in Algebraic Development Techniques*, LNCS 1827, 1-21, 2000.

[169] J. van den Berg, B. Jacobs, E. Poll. “Formal specification and verification of JavaCard’s application identifier class”. *JavaCard Workshop*, 2000.

[170] G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[171] D. Volpano. G. Smith. “A type-based approach to program security”. *Proceedings of TAPSOFT’97, Colloquium on Formal Approaches in Software Engineering*, LNCS 1214, 607-621, 1997.

[172] D. Volpano, G. Smith, C. Irvine. “A sound type system for secure flow analysis”. *Journal of Computer Security*, 4(3):1-21, 1996.

[173] D. Volpano, G. Smith. “Language issues in mobile program security”. *Mobile agents and security*, LNCS 1419, 25-43, 1998.

[174] D. von Oheimb. “Axiomatic semantics for Java_{light}”. *ECOOP2000 Workshop on Formal Techniques for Java Programs,* 2000.

[175] D. von Oheimb, T. Nipkow. “Machine-checking the Java specification: proving type safety”. *Formal Syntax and Semantics of Java*, LNCS 1523, 119-156, 1999.

[176] D. Walker. “A type system for expressive security policies”. *27th Symposium on Principles of Programming Languages*, 254-267, 2000.

[177] C. Wallace. “The semantics of the Java programming language: preliminary version”. Technical report CSE-TR-355-97, University of Michigan EECS Department, 1997.

[178] D.S. Wallach. “A new Approach to Mobile Code Security”. PhD Thesis, Princeton University, 1999.

[179] D.S. Wallach, E.W. Felten. “Understanding Java stack inspection”. *Symposium on Security and Privacy*, 52-63, 1998.

[180] Ph. Yelland. “A compositional account of the Java virtual machine”. *26th International Conference on Principles of Programming Languages*, 57-69, 1999.

[181] F. Yellin. “Low level security in Java”. *Proceedings of the Fourht International Word Wide Web Conference*, 369-379, 1995.