Entorno de Código Móvil Seguro*

Jorge Aguirre[1], **Gabriel Baum**[1,3] **Marcelo Arroyo**[1]
Francisco Bavera[1], **Ricardo Medel,**[1,2] **Martín Nordio,**[1]
Cecilia Kilmurray[1], **Johanna Mussolini**[1]

(1) Departamento de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, Argentina
{jaguirre,marroyo,pancho,nordio,ckilmurray,johanna}
@dc.exa.unrc.edu.ar

(2) Stevens Institute of Technology,
New Jersey, EE.UU.,
rmedel@cs.stevens-tech.edu

(3) LIFIA, Universidad Nacional de La Plata
La Plata, Argentina
gbaum@sol.info.unlp.edu.ar

Resumen

En este trabajo se presenta la línea Código Móvil Seguro del grupo de investigación "Procesadores de Lenguajes" del Departamento de Computación de la Universidad Nacional de Río Cuarto. Se presenta una técnica para garantizar código móvil seguro, denominada *Proof-Carrying Code based on Static Analysis* (PCC-SA), cuya principal ventaja es que el tamaño de las pruebas generadas es lineal respecto a la longitud de los programas. A fin de demostrar la aplicabilidad de esta tecnica, se han implementado prototipos de un compilador certificante y un verificador de codigo basados en ella. En este trabajo tambien se presentan nuevas lineas de trabajo iniciadas, entre las cuales se destacan la extensión del prototipo de compilador certificante, la paralelización de PCC-SA y un modelo de seguridad para programas concurrentes.

Palabras Clave: Código Móvil Seguro, Análisis Estático, Verificación de Código, Certificación de Código, Compiladores Certificantes.

1 Introducción

La interacción entre sistemas de software por medio de código móvil es un método poderoso que permite instalar y ejecutar código dinámicamente. De este modo, un servidor puede proveer medios flexibles de acceso a sus recursos y servicios internos. Pero este método presenta un riesgo grave para la seguridad del receptor, ya que el código móvil puede utilizarse también con fines maliciosos, dependiendo de las intenciones de su creador o un eventual interceptor.

Existen diversos enfoques tendientes a garantizar la seguridad en un ambiente de codigo movil. Una técnica que despertó mucho intéres en el último tiempo es *Proof-Carrying Code* (PCC). *Proof-Carrying Code* (PCC), una tecnica propuesta por Necula y Lee [5] para garantizar codigo movil

*Este trabajo ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Córdoba Ciencia y por la NSF.

seguro, ha despertado mucho interes y ha generado activas lineas de investigacion con una importante produccion [1, 2, 3, 5, 6]. Sin embargo, PCC esta limitado a propiedades que puedan expresarse en un sistema de tipos formal. La idea básica de PCC consiste en requerir al productor de código la evidencia necesaria, en este caso una prueba formal, de que su código satisface las propiedades deseadas. Esta evidencia constituye un certificado que puede ser verificado independientemente, es decir, no requiere autentificación del productor.

Proof-Carrying Code based on Static Analysis (PCC-SA) [7] es un entorno de ejecución de código móvil seguro basado en PCC y analisis estatico. El objetivo es proporcionar una solución en aquellos casos en los cuales la política de seguridad no puede ser verificada eficientemente por un sistema de tipos formal, como es el caso de verificar inicialización de variables y accesos válidos a arreglos. PCC-SA utiliza un código intermedio de más alto nivel que PCC: un *árbol sintáctico abstracto* (AST) anotado con información del estado del programa. Este tipo de representación intermedia permite realizar diversos análisis estáticos para generar y verificar la información necesaria, y permitiendo ademas aplicar ciertas optimizaciones al código generado.

La principal ventaja de PCC-SA reside en que el tamaño de la prueba generada es lineal con respecto al tamaño de los programas. Además, la complejidad de la generación de las anotaciones y la verificación de la seguridad del código también es lineal con respecto al tamaño de los programas.

En este trabajo se presentan las líneas de investigación abordadas en el campo del código móvil. En primer lugar se presentan los resultados obtenidos en el último año: el entorno de ejecución PCC-SA (sección 2.1), el prototipo de compilador certificante desarrollado (sección 2.2) y el prototipo verificador de la prueba desarrollado (sección 2.3). En la sección 3 se presentan los trabajos inciados que incluyen la extensión y optimización del prototipo de compilador certificante 3.1, la paralelización del entorno de ejecución PCC-SA (sección 3.2) y una extensión de PCC en entornos concurrentes (sección 3.3).

2 Resultados Obtenidos

2.1 El Entorno de Ejecución PCC-SA

La figura 1 muestra la interacción de los componentes del entorno PCC-SA. Siguiendo la notación de Necula [5] los cuadrados ondulados representan código y los rectangulares representan componentes que manipulan dicho código. Además, las figuras sombreadas representan entidades no confiables, mientras que las figuras blancas representan entidades confiables. Consultar [7] para más detalles de PCC-SA y del prototipo desarrollado.

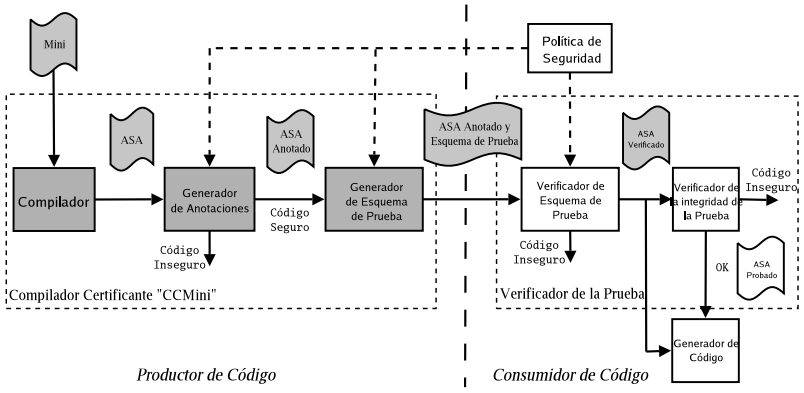

Figura 1: Vista global del entorno de ejecución de PCC-SA.

En nuestra implementacion, el *Compilador* toma como entrada el código fuente, escrito en un simple lenguaje imperativo denominado Mini, y produce el código intermedio, en forma de un árbol sintáctico abstracto. Este código intermedio es una representación abstracta del código fuente

que puede ser utilizado independientemente del lenguaje fuente y de la política de seguridad. El *Compilador* es un compilador tradicional que realiza los típicos análisis lexicográficos, sintácticos y semánticos. El *Generador de Anotaciones* (*GenAnot*) efectúa diversos análisis estáticos, generando la información necesaria para producir las anotaciones del código intermedio de acuerdo a la política de seguridad. La política de seguridad que definimos garantiza seguridad de tipos y de memoria, además de garantizar que no se leen variables no inicializadas y que no se accede a posiciones no definidas sobre los arreglos. La formalización de la política de seguridad es presentada en [7]. En aquellos puntos del programa en donde las técnicas de análisis estático no permiten determinar fehacientemente el estado, se insertan chequeos en tiempo de ejecución para garantizar de esta forma seguridad durante la ejecución del codigo. En caso de encontrar que en algún punto del programa forzosamente no se satisface la política de seguridad, el programa es rechazado. El último proceso del lado del *productor de código* es realizado por el *Generador del Esquema de Prueba*. Éste, teniendo en cuenta las anotaciones y la política de seguridad, elabora un esquema de prueba considerando los puntos críticos y sus posibles dependencias. Esta información se encuentra almacenada en el código intermedio. El *Compilador*, el *Generador de Anotaciones* y el *Generador del Esquema de Prueba* conforman el *Compilador Certificante*.

El *consumidor de código* recibe el código intermedio con anotaciones y el esquema de prueba. Este esquema de prueba es básicamente el recorrido mínimo que debe realizar el *consumidor de código* sobre el código intermedio y las estructuras de datos a considerar, a fin de verificar el codigo recibido. El *Verificador del Esquema de Prueba* es el encargado de corroborar el esquema de prueba generado por el *productor de código*. Por último el *Verificador de la Integridad de la Prueba* verifica que el esquema de prueba haya sido lo suficientemente fuerte para poder demostrar que el programa cumple con la política de seguridad. Esto consiste en verificar que todos los puntos críticos del programa fueron chequeados por el *Verificador del Esquema de Prueba* o bien contienen un chequeo en tiempo de ejecución. Esto es necesario porque si el código hubiese sido modificado durante su envío, el esquema de prueba puede no contemplar ciertos puntos del programa potencialmente inseguros. El *Verificador de Esquema de Prueba* y el *Verificador de la Integridad de la Prueba* componen el *Verificador de la Prueba*.

2.2 Prototipo de Compilador Certificante

Con el objetivo de experimentar y analizar el enfoque propuesto, se desarrolló un prototipo del *Compilador Certificante* descripto en la sección 2.1. Este prototipo se denominó CCMini. Como primera medida se definieron los aspectos de seguridad que se deseaban garantizar, es decir, la política de seguridad. La política de seguridad definida para el prototipo consiste en que toda variable se encuentra inicializada al momento de ser utililizada y en que todo índice de arreglo es un índice válido. El próximo paso consistió en establecer el código intermedio que se utilizaría, un árbol sintáctico abstracto (AST). Luego se definió el lenguaje fuente del compilador, denominado Mini. Finalmente se implementaron los módulos correspondientes al *Generador de Código Intermedio*, el *Generador de Anotaciones*, el *Certificador* y el *Generador del Esquema de Prueba*.

Con esta primera implementación se pretende por un lado poder analizar la factibilidad de su inclusión dentro del framework de PCC-SA y por el otro analizar su efectividad y eficiencia.

2.3 Prototipo de Verificador de la Prueba

Para probar la factibilidad de uso de PCC-SA, se desarrolló un prototipo de verificador de la prueba. El prototipo es un verificador que permite descargar programas escritos en un código intermedio y ejecutarlo de manera segura. Debido a que ya se habian definido los aspectos de seguridad que se deseaban garantizar y el AST, el primer paso de esta implementación consistió en desarrollar el *Verificador del Esquema de Prueba*, el *Verificador de la Integridad de la Prueba*. Luego se diseñó el *Generador de Código*. El prototipo fue implementado usando el lenguaje de programación Java.

Al igual que CCMini, el diseño del prototipo de verificador de la prueba asegura que sólo serán descartados aquellos programas que son ciertamente inseguros. Si el prototipo no puede determinar la seguridad de un programa, este inserta chequeos en tiempo de ejecución en los puntos críticos.

Utilizando el prototipo desarrollado, se realizaró una serie de experimentos [1]. En la mayoría de los casos, el prototipo determinó la seguridad de los programas sin insertar chequeos en tiempo de ejecución. En cuanto al tamaño de las pruebas, los experimentos mostraron que el tamaño de las pruebas es lineal al tamaño de los programas.

3 Trabajos Iniciados

3.1 Extensión y optimización del prototipo de compilador certificante

Con el objetivo de determinar la factibilidad de uso de PCC-SA en aplicaciones "reales", se está trabajando en la extensión del prototipo de compilador certificante CCMini. Para esto, se extenderá el lenguaje fuente de modo de incluir el tratamiento de punteros, invocación de funciones y otras sentencias de flujo de control (for y repeat). Esto traerá aparejado la necesidad de extender la política de seguridad para considerar nuevos puntos críticos. Además, parte del trabajo a realizar consistirá en optimizar el proceso de generación de anotaciones utilizando *Pilas de Contextos* [7].

El diseño del compilador certificante ser simple y sencillo de manera tal que si éste es aplicado en un ambiente de código móvil seguro, el verificador no consuma demasiados recursos. Esta es una caracteristica importante, ya que el entorno podria ser aplicado a ambientes con recursos limitados como por ejemplo las tarjetas inteligentes (*SmartCards*).

3.2 Paralelización de PCC-SA

En busca de mejorar la performance del framework PCC-SA, se está trabajando en el desarrollo de un nuevo entorno denominado Parallel Proof-Carrying Code based on Static Analysis (Parallel PCC-SA) que permitirá verificar la prueba de seguridad en paralelo. La idea básica de Parallel PCC-SA consiste en requerir al *productor de código* una prueba de que el código satisface las propiedades de seguridad deseadas dividida en q esquemas de pruebas las cuales pueden ser verificadas en paralelo.

La figura 2 muestra la interacción de los componentes del entorno. Al igual que en PCC-SA, el *Compilador Certificante* toma como entrada el código fuente y produce como resultado una representación intermedia anotada. El *Generador de Esquemas de Pruebas* es el encargado de generar el esquema de prueba y dividir el esquema de prueba en q esquemas de pruebas para que puedan ser verificados independientemente. Una vez obtenidos estos esquemas de pruebas, los mismos son enviados al consumidor de código para realizar su verificación y la ejecución del código.

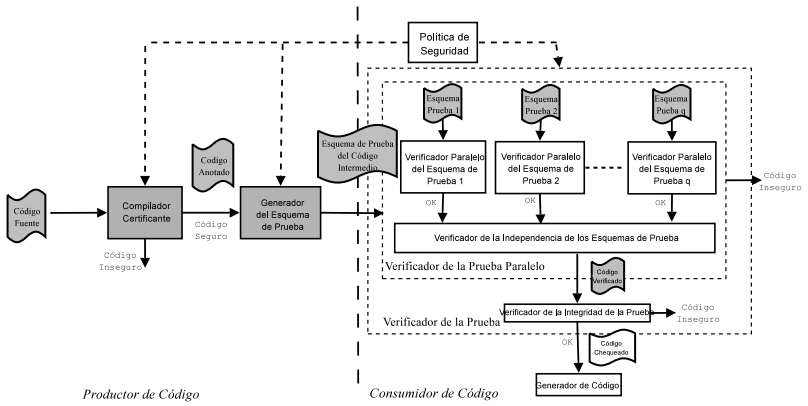

Figura 2: Estructura del entorno Parallel PCC-SA.

[1] Mas detalles de los experimentos realizados consultar [8]

3.3 Proof-Carrying Code en Entornos Concurrentes

El objetivo es definir un entorno de código móvil que garantice la seguridad de programas Java multi-thread. Los programas podrán contar con una especificación en el Java Modeling Language (JML), el cual es un lenguaje de especificación formal que permite especificar contratos en programas Java por medio de invariantes de clases y pre y post-condiciones de métodos. Además, estos programas deberán respetar el nuevo modelo de memoria de Java (JMM), que es parte de Java 5.0 [4] y soluciona las limitaciones que tenía el modelo anterior. Según Pugh [4], el modelo anterior prohibe la aplicación de muchas optimizaciones de código, mientras que el acceso no sincronizado a objetos inmutables (`final`) puede no ser *thread-safe*.

En esta línea se comenzó a trabajar en un proceso de compilación para una extensioń del JML introducida por Rodrígez et. al [9]. Los constructores provistos por esta extensión permiten especificar propiedades de *no-interferencia* y *atomicidad* de métodos en programas Java multi-threaded. Además, permiten realizar la verificación en forma modular.

El proceso de compilación genera un archivo `.class` que incluye la especificación JML en un atributo que instancia una estructura de datos predefinida por la Java Virtual Machine Specification JVMS. Se podría utilizar un compilador estándar Java para generar el `.class` y luego insertar la especificación JML. Sin embargo, si se considera el nuevo JMM se deben tener en cuenta las restricciones impuestas y analizar las acciones que debe realizar el compilador para no violar ni la especificación JML ni el modelo de memoria.

Un componente muy importante de esta línea es el desarrollo de los métodos para poder razonar usando las especificaciones del JML extendido sobre bytecode. Dichos métodos deben permitir verificar la especificación y generar la prueba para que pueda ser utilizada en un ambiente de código móvil.

Referencias

[1] A. Appel, "Foundational Proof-Carrying Code", in *Proceedings of the 16^{th} Annual Symposium on Logic in Computer Science*, pp. 247–256, IEEE Computer Society Press, 2001.

[2] A. Bernard, P. Lee, "Temporal Logic for Proof-Carrying Code", in *Proceedings of Automated Deduction* (CADE-18), Lectures Notes in Computer Science 2392, pp. 31–46, Springer-Verlag, 2002.

[3] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline, "A certifying compiler for Java", in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'00), pp. 95–105, ACM Press, Vancouver (Canada), June 2000.

[4] Jeremy Manson, William Pugh, Sarita V. Adve. "The Java memory model". POPL 2005. pp 378-391.

[5] G. Necula "Compiling with Proofs" Ph.D. Thesis School of Computer Science, Carnegie Mellon University CMU-CS-98-154. 1998.

[6] G. Necula, R. Schneck, "Proof-Carrying Code with Unstrusted Proof Rules", in *Proceedings of the 2^{nd} International Software Security Symposium*, November 2002.

[7] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum, "A Framework for Execution of Secure Mobile Code based on Static Analysis". En los Proceddings de XXIV Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computacin IEEE-CS Press, Tarapaca, Chile. Noviembre de 2004.

[8] M. Nordio, "Verificación de la Seguridad del Código Foraneo mediante Análisis Estático de Control de Flujo y de Datos". Tesis de Maestría
en Informática. PEDECIBA Informática, Instituto de Computación, Universidad de la República, Montevideo, Uruguay. Marzo 2005.

[9] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Robby. "Extending Sequential Specification Techniques for Modular Specification and Verification of Multi-Threaded Programs". ECOOP 2005.