Grupo de Procesadores de Lenguajes - Línea: Código Móvil Seguro*

Francisco Bavera † Martín Nordio † Jorge Aguirre † Marcelo Arroyo †
Gabriel Baum‡ Ricardo Medel§

Resumen

En el último tiempo *Proof-Carrying Code* (PCC) ha despertado un gran interés surguiendo numerosas líneas de trabajo. PCC establece una infraestructura que permite garantizar que los programas se ejecutarán de manera segura. En esta alternativa, el productor de código adjunta al código móvil una demostración, mediante la cual el consumidor del código puede verificar su seguridad antes de la ejecución del programa. Análisis estático es un técnica con un gran potencial para producir la evidencia necesaria para garantizar código móvil seguro. Basándonos en estas dos técnicas se presenta una arquitectura para garantizar la ejecución de código móvil de manera segura. Además se presenta el prototipo implementado que demuestra la factibilidad del uso de la técnica.

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 poderoso 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 que las aplicaciones son seguras, una técnica que despertó mucho intéres en el último tiempo es PCC [6, 1]. 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 (no requiere autentificación del productor).

Este tipo de esquemas, que producen software confiable durante la compilación, pueden ser dividido en dos pasos. El primer paso consiste en compilar el código fuente e introducir anotaciones en el archivo objeto (es decir bytecode para JAVA, assembly para C, o cualquier otro lenguaje intermedio). En la actualidad se pretende que estas anotaciones sean generadas e introducidas de manera automatizada para que PCC pueda ser utilizado en la industria. En el segundo paso, el código anotado es tomado por un verificador que chequea que el código anotado no viole las condiciones de seguridad impuestas. Si la verificación tuvo exito es seguro ejecutar la aplicación final.

Muchos trabajos sobre PCC estan basados en sistemas de tipos, pero existen muchas casos en los cuales la política de seguridad deseada 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. Esta falta de eficiencia se refiere tanto a nivel de razonamiento sobre propiedades del código como a nivel de performance. Además muchos aspectos relativos a seguridad no pueden ser representados por sistemas de tipos.

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

†Universidad Nacional de Río Cuarto,{pancho,nordio,jaguirre,marroyo}@dc.exa.unrc.edu.ar
‡Universidad Nacional de La Plata, gbaum@sol.info.unlp.edu.ar
§Stevens Institute of Technology (New Jersey, EE.UU.), rmedel@cs.stevens-tech.edu

La información necesaria para garantizar estos aspectos de seguridad puede ser generada mediante el análisis estático de flujo de los programas [3, 4]. De esta manera se aprovechan los beneficios del esquema PCC y del análisis estático. En esta línea de trabajo se definió la arquitectura de un entorno de ejecución de código móvil seguro.

Este trabajo está estructurado de la siguiente manera: en la sección 2 se presenta la arquitectura junto con un breve análisis de las motivaciones y ventajas de la misma. En la sección 3 se comentan las características más relevantes del prototipo implementado para corroborar la factibilidad del enfoque. Por último se encuentran algunas conclusiones extraídas.

2 Nuestra Propuesta

Proof-Carrying Code based on Static Analysis (PCC-SA) está centrado en poder brindar 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. Esta falta de eficiencia se refiere tanto a nivel de razonamiento sobre propiedades del código como a nivel de performance. V. Haldar, C Stork y M. Franz [4] argumentan que en PCC estas ineficiencias son causadas por la brecha semántica que existe entre el código fuente y el código móvil de bajo nivel utilizado. Por estas razones, PCC-SA utiliza como código intermedio de más alto nivel como por ejemplo un *grafo de flujo de control* o un *árbol sintáctico abstracto* anotado con información de tipos. Además utiliza análisis estático para generar y verificar esta información. Sobre la representación intermedia se realizan diversos análisis estáticos y optimizaciones de código.

Análisis estático es una técnica muy conocida en la implementación de compiladores que en los últimos años ha despertado interés en distintas áreas tales como verificación y re-ingeniería de software. La verificación de propiedades de seguridad puede ser realizada por medio de análisis estáticos debido a que este permite obtener una aproximación concreta del comportamiento dinámico de un programa antes de ser ejecutado. Existen una gran cantidad de técnicas de análisis estático, que balancean el costo entre el esfuerzo de diseño y la complejidad requerida, que pueden ser usados para realizar verificaciones de seguridad - por ejemplo, *array-bound checking* o *escape analysis* - [3].

Si bien las técnicas de análisis estático requieren un esfuerzo mayor que el realizado por un compilador tradicional - que solo realiza verificación de tipos y otros análisis simples de programas - el esfuerzo requerido, comparándolo con verificación formal de programas, es mucho menor. No hay que dejar de mencionar que se debe llegar a un acuerdo entre precisión y escalabilidad. Se debe asumir el costo de que en algunos casos se pueden rechazar programas por ser inseguros cuando en realidad no lo son.

Es conveniente aclarar que si bien el análisis estático es un enfoque importante para garantizar seguridad no soluciona todos estos problemas. En muchos casos este no puede reemplazar controles en tiempo de ejecución, testing sistemático o verificación formal. Además no se avizora ninguna técnica que pueda eliminar todos los riesgos de seguridad, y por las ventajas que presenta el análisis estático debería ser parte del proceso de desarrollo de aplicaciones seguras [3]. Esta técnica podría ser utilizada, por ejemplo, en conjunción con PCC tradicional. Este enfoque, si bien usa análisis estático sigue los lineamientos de PCC y Compiladores Certificantes.

En este enfoque se pueden insertar chequeos en tiempo de ejecución para ampliar el rango de programas seguros en los cuales el análisis estático no puede asegurar nada. Es decir, el mecanismo usado para verificar que el código es seguro puede ser una combinación de verificaciones estáticas - en tiempo de compilación - y de chequeos dinámicos - en tiempo de ejecución -. Cabe resaltar que, en muchos casos, es necesario insertar estos chequeos en tiempo de ejecución no solo por las limitaciones de un análisis estático particular sino porque los problemas a resolver son no computables como por ejemplo garantizar la seguridad al eliminar la totalidad de *array-bound checking* que en su caso general es equivalente al problema de la parada.

La figura 1 muestra la interacción del sistema propuesto. Al igual que en [6] 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.

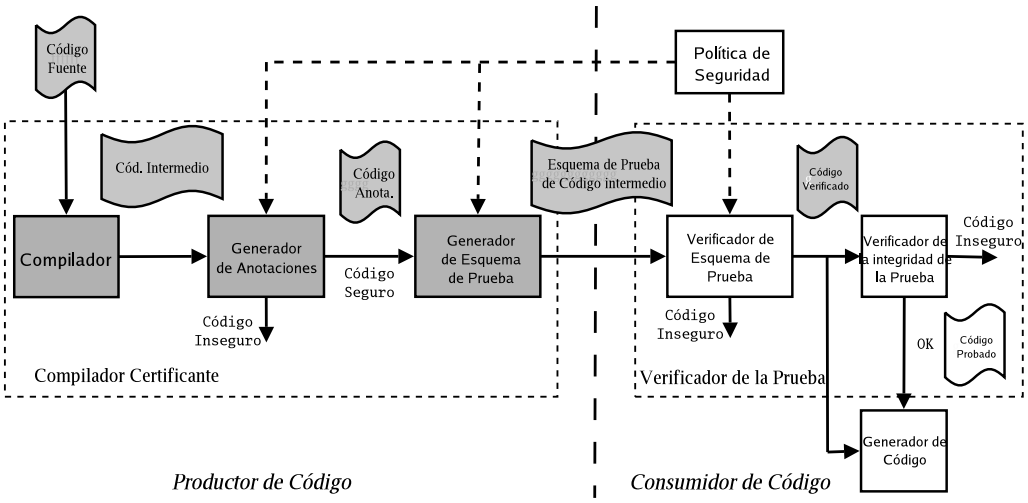

Figura 1: Vista Global de la arquitectura *Proof-Carrying Code based on Static Analysis*.

El **Compilador** toma como entrada el código fuente y produce el código intermedio. Este código intermedio es una representación abstracta del código fuente y puede ser utilizado independientemente del lenguaje fuente y de la política de seguridad. El **Compilador** es un compilador tradicional que realiza los típicos análisis léxico y sintáctico verificando únicamente el tipo de las expresiones. El **Generador de Anotaciones** efectúa diversos análisis estáticos generando la información necesaria para producir las anotaciones del código intermedio de acuerdo a la política de seguridad. 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 la seguridad de código. 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 llevado a cabo por el *productor de código* es realizado por el **Generador del Esquema de Prueba**. Este, 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 almacena en el código intermedio. El **Compilador**, el **Generador de Anotaciones** y el **Generador del Esquema de Prueba** conforman al **Compilador Certificante**.

El *consumidor de código* recibe el código intermedio con las anotaciones del código 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. 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 halla 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 en su defecto contienen un chequeo en tiempo de ejecución. Esto se debe a que, si el código del productor hubiese sido modificado en el proceso de envio al consumidor, el esquema de prueba puede no haber contemplado 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**.

3 El Prototipo Desarrollado

Con el objetivo de verificar la factibilidad del enfoque propuesto se desarrolló un prototipo de la arquitectura descripta en el punto 2. Como primera medida se definió los aspectos de seguridad que se deseaban garantizar, es decir, la política de seguridad. Esta puede ser expresada en dos

reglas:

1. toda variable está inicializada al momento de ser utilizada.

2. todo acceso a una posición de un arreglo es válida.

Luego se definió el lenguaje fuente que es básicamente un subconjunto de C. Un programa fuente es una función que debe tomar por lo menos un parámetro y debe retornar un valor. Tanto los parámetros como los valores que retorna deben ser de tipo básico. Los tipos básicos son int y boolean, a diferencia de C en el cual cero significa falso y cualquier entero distinto de cero significa verdadero; además cuenta con arreglos unidimensionales de tipo int o boolean. Otra diferencia con C es que la asignación es una sentencia y no una expresión. Además, opcionalmente los parámetros de tipo entero pueden ser acotados al ser declarados. Para esto la declaración del parámetro va acompañada por un rango que indica el menor y el mayor valor que puede tomar el dato de entrada.

El próximo paso consistió en establecer el código intermedio que se utilizaría. Como representación intermedia se definió un árbol sintáctico abstracto ya que esta representación permite realizar distintos análisis estáticos (también, si así se deseara, puede ser usado para generar una gran cantidad de optimizaciones).

En la implementación se limita el análisis al cuerpo de las funciones para hacer el análisis más eficiente y escalable a programas largos. Otra consideración a mencionar es el punto medio considerado entre *flow-sensitive analysis* - el cual considera todos los flujos posibles del programa - y *flow-insensitive analysis* - el cual ignora el control de flujo -. Si bien se considera el control de flujo en algunos casos, como por ejemplo en los ciclos, se limita a reconocer ciertos patrones en el código.

Luego se implementó el **Generador de las Anotaciones del Código Intermedio**. El objetivo de este módulo es generar la información necesaria para determinar la inicialización de las variables y el rango de las variables utilizadas. Este último objetivo permite asegurar si un valor es válido como índice de arreglo. Además se determinará la precondición y la postcondición.

Para lograr con los objetivos antes mencionados se identificarán todas las posibles secuencias de ejecución de los programas. Es decir, el grafo permitirá generar información acerca de que pasará en tiempo de ejecución. Esto se realiza en tres pasos:

1. Identificación de rangos de variables que no son modificadas dentro de un ciclo.

2. Identificación de rangos de variables que son modificadas por valores acotados dentro de un ciclo.

3. Identificación de rangos de variables inductivas que son modificadas dentro de un ciclo.

El último módulo del productor implementado, el **Generador del Esquema de Prueba**, identifica las variables críticas (aquellas que intervienen como índices de arreglos y sus dependencias) y los puntos del programa en que son referenciadas. Con esta información produce el esquema de prueba, el cual es el camino mínimo que el consumidor debe recorrer para verificar la seguridad del código.

El **Verificador del Esquema de Prueba** chequea la consistencia del grafo recibido. Luego, recorriendo el camino de prueba, verifica que toda variable este inicializada y que todos los accesos dentro del camino de prueba sean seguros. Todos los nodos visitados por esta entidad son marcados. Finalmente, el **Verificador de la Integridad de la Prueba** realiza un recorrido exhaustivo del grafo para garantizar que la entidad anterior visitó todos los nodos que contienen una variable crítica. Si el programa es seguro el **Generador de Código** produce el código objeto a partir del grafo.

4 Conclusiones y Trabajos Futuros

Se han desarrollado los lineamientos de un mecanismo de seguridad basado en técnicas de compilación y análisis estático. Para garantizar la seguridad del consumidor de código esta técnica

sigue las ideas de Proof-Carrying Code. La factibidad de este enfoque ha sido probado mediante el desarrollo e implementación de un prototipo del ambiente. El prototipo está conformado por un compilador certificante para un subconjunto del lenguaje C y el verificador de la prueba que garantizan las siguientes propiedades de seguridad:

1. toda variable está inicializada al momento de ser utilizada.
2. todo acceso a una posición de un arreglo es válida.

Como consecuencia de la información necesaria para poder garantizar el punto 2 se obtiene la evidencia que permite determinar, en muchos casos, las cotas de las variables en cada punto de ejecución del programa. Otra consecuencia es poder determinar, en ciertos casos, la postcondición.

Si bien no siempre se puede garantizar en tiempo de compilación que el acceso a un arreglo es válido, el compilador inserta chequeos en tiempo de ejecución, con lo cual, se garantizan las propiedades de seguridad. Además, como generalmente los accesos a arreglos se realizan mediante variables inductivas, el compilador puede determinar en la mayoría de los casos la validez de los accesos sin insertar chequeos. Para poder determinar la certeza de esta afirmación, se realizará un testing del compilador con la participación de un grupo de programadores (estudiantes, docentes y investigadores del Dpto. de Computación de la UNRC). Como trabajo futuro se desarrollará una herramienta que permitar conocer cual es el porcentaje de programas que requieren verificaciones dinámicas de las propiedades de seguridad que se garantizan para un lenguaje de programación del mundo real, como por ejemplo, bibliotecas C.

Si bien el lenguaje fuente del compilador es simple puede ser extendido facilmente. Sería interesante extender el lenguaje permitiendo invocación a funciones. Esto se puede realizar sin mayores dificultades ya que, al obtener la postcondición de una función, esta se puede acotar; además si consideramos punteros se podría verificar la existencia de *aliasing* por medio de análisis de alias basado en flujo (*alias analysis based on flow*) y realizar *scape analysis*.

Otro aspecto a tener en cuenta es el desarrollo un lenguaje assembly tipado para que en una etapa posterior, el compilador lo generara. También, sería muy importate contar con un sistema formal que verifique las propiedades de seguridad.

Referencias

[1] F. Bavera, M. Nordio, R. Medel, J. Aguirre, G. Baum "Avances en Proof-Carrying Code", en *Proceedings del CACIC'03*, UNLP, La Plata, Argentina, 2003. http://dc.exa.unrc.edu.ar/lenguajes/Files/AvancesDePCC.ps

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

[3] D. Evans and D. Larochelle, Improving Security Using Extensible Lightweight Static Analysis. IEEE Software, pp. 42-51, Enero-Febrero 2002

[4] V. Haldar, C. Stork, M. Franz, Tamper-Proof Annotations - by Construction. Technical Report 02-10, Departament of Information and Computer Science, University of California, Irvine, Marzo 2002.

[5] L. Hornof, T. Jim, "Certifying Compilation and Run-Time Code Generation", en *Proceedings of ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation* (PEPM), pp. 60–74, ACM Press, San Antonio, Texas (EE.UU.), Enero 1999.

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

[7] G. Necula, P. Lee, "The Design and Implementation of a Certifying Compiler", en *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'98), pp. 333–344, ACM Press, Montreal (Canadá), Junio 1998.