Análisis de Flujo de Control y Código Móvil Seguro
¿Una Alternativa Prometedora? *

Francisco Bavera, Martín Nordio, Jorge Aguirre
Departamento de Computación, Universidad Nacional de Río Cuarto
Río Cuarto, Argentina
{pancho,nordio,jaguirre}@dc.exa.unrc.edu.ar

Resumen

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. *Seguridad basada en el Lenguaje Fuente* es un conjunto de técnicas, basadas en lógicas formales y sistemas de tipos, desarrolladas para garantizar y demostrar estáticamente que el software posee ciertas cualidades. En particular, esta técnica se concentra en el análisis de cualidades críticas, tales como seguridad de tipos y seguridad de memoria. Para que estas técnicas puedan ser aplicadas industrialmente es necesario llegar a una implementación eficiente, totalmente efectiva y que genere certificados (pruebas de seguridad) de tamaño reducido. En este trabajo se presenta evidencia que demuestra que la combinación de análisis estático de flujo de control y verificaciones dinámicas (siguiendo los lineamientos de *Seguridad basada en el Lenguaje Fuente*) puede constituir una aproximación que permita implementaciones eficientes que generen pruebas de tamaño lineal con respecto a la longitud del código de entrada.

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

*Este trabajo ha sido realizado en el marco de proyectos subsidiados por la SECyT de la UNRC y por la Agencia Nacional de Promoción Científica y Tecnológica.

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. Los problemas de seguridad que presentan las técnicas de código móvil no confiables suponen riesgos costosos para los sistemas.

Aún existen muchos problemas a ser resueltos para hacer confiable la práctica del uso seguro de código móvil. El foco del análisis de este problema consiste en establecer garantías acerca del comportamiento de programas no confiables, por lo que es preciso que el consumidor de código tenga en cuenta las siguientes cuestiones: ¿Cómo puede, el consumidor de código, asegurarse que el código no lo dañará, por ejemplo, corrompiendo sus estructuras de datos? ¿Cómo puede asegurarse que el código no usará demasiados recursos o que no los usará por un período de tiempo demasiado extenso? ¿Cómo puede asegurarse que el código no permitirá accesos maliciosos a sus recursos? Finalmente, ¿Cómo puede el consumidor hacer estas aseveraciones sin realizar esfuerzos excesivos y sin tener efectos perjudiciales en la performance global de su sistema?

Existen diversos enfoques tendientes a garantizar que las aplicaciones son seguras. Entre los cuales se destacan testing sistemático [8, 25], introducción de verificaciones dinámicas [4, 23] y análisis estático [5, 6, 7, 10, 13, 15].

El testing sistemático es costoso en cuanto al tiempo que consume pero se pueden encontrar problemas imposibles de detectar automáticamente. Sin embargo, estas actividades dependen de expertos y generalmente el testing no es muy efectivo para encontrar vulnerabilidades de seguridad.

Modificar programas para insertar verificaciones dinámicas o ejecutar las aplicaciones en un ambiente limitado (por ejemplo *sandboxing*) reduce el riesgo de que el código posea vulnerabilidades de seguridad. Una desventaja de estas técnicas es el aumento del tiempo de ejecución de los programas por la sobrecarga introducida por las verificaciones.

Las técnicas de análisis estático tienen un enfoque distinto. Permiten obtener una aproximación concreta del comportamiento dinámico de un programa antes de ser ejecutado. Desde el punto de vista de la seguridad esto es una ventaja adicional significativa. Existe una amplia variedad de técnicas de análisis estático. Podemos encontrar desde compiladores tradicionales que con relativamente poco esfuerzo realizan simples verificaciones de tipos. Hasta, en el otro extremo, verificadores de programas que requieren especificaciones formales completas y utilizan demostradores de teoremas. Estos últimos son efectivos y pueden utilizarse para verificar propiedades complejas de los programas pero son costosos (en tiempo y complejidad) y en la mayoría de los casos no pueden ser automatizados.

La certificación de código es una técnica desarrollada para garantizar y demostrar estáticamente que el software posee ciertas cualidades. En particular, esta técnica se concentra en el análisis de cualidades críticas, tales como seguridad de tipos y seguridad de memoria. La idea básica consiste en requerir que el productor de código realice una prueba formal (el certificado) que evidencie de que su código satisface las propiedades deseadas. El código producido puede estar destinado a ser ejecutado localmente por el productor, que en este caso es también su consumidor, o a migrar y ser ejecutado en el entorno del consumidor. En el primer caso tanto el entorno de compilación y de ejecución son confiables. En el segundo caso, el único entorno confiable para el consumidor es el propio, dado que, el código puede haber sido modificado maliciosamente o no corresponder al supuesto productor. En el primer caso la prueba realizada de que el código generado cumple la política de seguridad constituye ya una certificación suficiente. En cambio el segundo caso se inscribe en la problemática de seguridad del código móvil y entre cuyas técnicas para garantizar la seguridad del consumidor se encuentran las variantes de *Seguridad basada en el Lenguaje Fuente* [15, 13, 10, 11, 17].

La idea de *Seguridad basada en el Lenguaje Fuente* consiste en mantener la información relevante obtenida del programa escrito en un lenguaje de alto nivel en el código compilado. Esta información extra (denominada el certificado) es creada en tiempo de compilación y adjuntada

al código compilado. Cuando el código es enviado al consumidor su certificado es adjuntado con el código. El consumidor puede entonces realizar la verificación analizando el código y su certificado para corroborar que cumple con la política de seguridad establecida. Si el certificado pasa la verificación entonces el código es seguro y puede ser ejecutado. La principal ventaja de este enfoque reside en que el productor de código debe asumir el costo de garantizar la seguridad del código (generando el certificado de seguridad). Mientras que el consumidor solo debe verificar si el certificado cumple con la política de seguridad [11].

La mayoría de la bibliografía sobre *Seguridad basada en el Lenguaje Fuente* está basada en la introducción de lógicas y sistemas de tipos que garanticen la seguridad buscada. Las principales desventajas de utilizar esta representación se ven reflejadas en la expresividad, eficiencia y en el tamaño de la prueba generada.

En muchos casos la política de seguridad no puede ser verificada eficientemente por un sistema 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 [6] argumentan que en *Proof-Carrying Code* [16] estas ineficiencias son causadas por la brecha semántica que existe entre el código fuente y el código móvil de bajo nivel utilizado. Además, en muchos casos las pruebas generadas tienen un tamaño exponencial con respecto a la longitud del programa de entrada. Lo cual es un problema de gran importancia para el código móvil

Análisis estático de flujo de control y de datos [2, 5, 6, 7] 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. Este tipo de análisis estático permite obtener una aproximación concreta del comportamiento dinámico de un programa antes de ser ejecutado. Con está técnica se puede realizar la verificación de propiedades de seguridad. Existe una gran cantidad de estos análisis, 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* - [5].

El análisis estático es una técnica importante para garantizar seguridad pero no soluciona todos los problemas asociados a esta. En muchos casos no puede reemplazar controles en tiempo de ejecución, testeo sistemático o verificación formal. Sin embargo, estas limitaciones no le son exclusivas ya que no se avizora ninguna técnica que por si sola pueda eliminar todos los riesgos de seguridad. Riesgos que, por lo general, conducen a problemas indecidibles. Con lo cual, el análisis estático debería ser parte del proceso de desarrollo de aplicaciones seguras [5] y combinarse con otras técnicas, por ejemplo, con *Proof-Carrying Code* tradicional.

Si bien las técnicas de análisis de flujo de control y de datos requieren un esfuerzo mayor que el realizado por un compilador tradicional (que sólo 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.

En este trabajo se compara el framework de PCC-SA [17] (*Proof-Carrying Code based on Static Analysis*) con otros frameworks que garantizan código móvil seguro. Se pretende generar evidencia necesaria para confirmar que la utilización de análisis estático de flujo de control y de datos es un enfoque interesante. No solo interesante desde el punto de vistas de la performance, sino también por su efectividad y por el tamaño de la información requerida para generar la prueba.

PCC-SA genera un *árbol sintáctico abstracto* anotado con la información necesaria para verificar las propiedades de seguridad. El proceso de verificación es realizado sobre este árbol sintáctico abstracto utilizando técnicas de análisis estático. En particular, se utilizan las técnicas de análisis de flujo de control y de datos. Para ampliar el rango de programas seguros se combinan estos análisis estáticos con verificaciones dinámicas. Estas son insertadas en aquellos casos donde no se puede determinar estáticamente el estado del programa en algún punto.

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, en el prototipo de PCC-SA desarrollado, 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. Es importante mencionar que PCC-SA inserta menos verificaciones dinámicas que Java y que no necesita ser asistido por anotaciones realizadas por el programador (como es el caso de otros entornos de código móvil).

Además, PCC-SA, posee las siguientes características que son de gran relevancia en un ambiente de código móvil: seguridad, independencia de la plataforma, verificaciones de seguridad simples, generación de pruebas de manera automática, pruebas pequeñas y provisión de la información necesaria para efectuar optimizaciones sobre el código. Además el consumidor de código cuenta con una infraestructura pequeña, confiable y automática con la cual verificará el código estáticamente.

1.1 Estructura del Paper

Este trabajo esta estructurado de la siguiente manera en la sección 2 se presenta el framework utilizado y el análisis de su performance y efectividad con respecto a otros frameworks. En la sección 3 se presenta el prototipo de PCC-SA, las experiencias realizadas con él y los resultados obtenidos. Por último, en la sección 4 se presentan las conclusiones a las que se llegaron y los trabajos futuros que se plantean realizar.

2 El Framework Utilizado

La figura 1 muestra la interacción de los componentes del entorno propuesto. 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 para el consumidor de código.

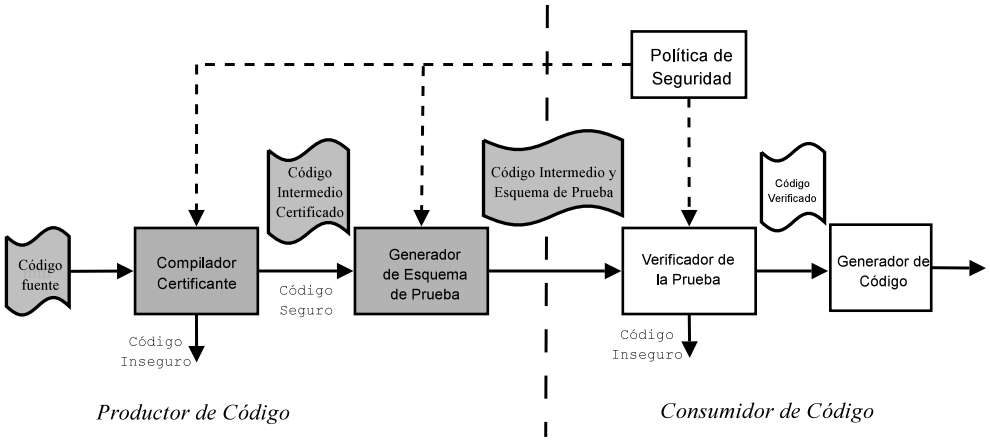

Figura 1: *Vista global de PCC-SA.*

El *Compilador Certificante* 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. Luego, este 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 del 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*. 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 *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. El *Verificador de la Prueba* es el encargado de corroborar mediante el esquema de prueba generado por el *productor de código* que el código satisface las anotaciones. Por último, este módulo 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 o bien contienen un chequeo en tiempo de ejecución. Esto se debe a que, si el código del productor hubiese sido modificado en el proceso de envío al consumidor, el esquema de prueba puede no haber contemplado ciertos puntos del programa potencialmente inseguros.

Desde el punto de vista del consumidor, en esta arquitectura la vida del código móvil abarca tres etapas:

1. Certificación, el productor de código compila el código y genera una prueba de que el programa fuente cumple con la política de seguridad. En el caso general, la certificación es esencialmente una forma de verificación del programa con respecto a la especificación descripta por la política de seguridad. Además, una prueba de verificación exitosa se produce y se codifica adecuadamente para obtener la prueba de seguridad. La prueba y el código compilado componen el código móvil. El productor de código puede almacenar el código resultante para usos futuros, o puede entregarlo al consumidor de código para su ejecución.

2. Validación, en esta segunda etapa, un consumidor de código valida la prueba de un programa dado y carga el código para la ejecución. La validación y aprobación es eficiente y la realiza un algoritmo seguro. El consumidor debe confiar solamente en la implementación de este algoritmo, además de confiar en la integridad de su política de seguridad.

3. Ejecución, finalmente, en la última etapa del proceso, el consumidor de código ejecuta el programa, posiblemente, muchas veces.

Esta organización en etapas permite que el proceso de validación se realice estáticamente y sólo una vez para un programa dado, independientemente del número de veces que éste se ejecute. Esto tiene ventajas importantes, sobre todo en aquellos casos en los que la validación es costosa o consume mucho tiempo.

2.1 Análisis del Framework Utilizado

2.1.1 Seguridad basada en el Lenguaje Fuente

Proof-Carrying Code (PCC) [15, 16], *Type Assembly Language* (TAL) [13, 20, 24], *Efficient Code Certification* (ECC) [10, 11] y PCC-SA [17] tienen como un punto en común usar la información generada durante el proceso de compilación para que el consumidor pueda verificar la seguridad del código eficientemente. Aunque difieren en expresividad, flexibilidad y eficiencia

El certificado involucrado en cada enfoque toma distintas formas. Con PCC, el certificado es una prueba en lógica de primer orden de ciertas condiciones de verificación, y el proceso de verificación involucra verificar que el certificado es una prueba válida en el sistema lógico adoptado. En PCC las pruebas son realizadas sobre un código móvil de bajo nivel (como por ejemplo un lenguaje *assembly* tipado). Con TAL el certificado es una anotación de tipos y el proceso de verificación consiste en realizar una verificación de tipos. La utilización de lenguajes *assembly* tipados tiene la ventaja de que la verificación de que el código satisface la política de seguridad es un proceso simple. En cambio, en PCC-SA, el proceso de verificación es mas costoso. Con ECC, el certificado es una anotación en el código objeto que indica la estructura e intención del código con información básica de tipos.

En PCC-SA las pruebas son realizadas sobre un código intermedio. Este código intermedio por su mayor nivel de abstracción mantiene propiedades del programa fuente que permiten generar y verificar la información necesaria para demostrar que el código satisface con las propiedades deseadas de forma más directa. Esta información necesaria es generada utilizando análisis estáticos

de flujo de control y de datos. Las pruebas PCC-SA son codificadas como esquemas de pruebas. Estos esquemas de pruebas representan los puntos críticos que el consumidor de código debe chequear para demostrar que el código es seguro.

La arquitectura de PCC-SA es similar a la de PCC. Tanto en PCC-SA como en PCC la infraestructura confiable es pequeña. Además, en PCC-SA y PCC el peso de la carga de garantizar la seguridad está en el productor y no en el consumidor de código. Pero tanto PCC-SA como PCC son sensibles a cambios de las políticas de seguridad. Realizar dichos cambios es un proceso costoso y dificultoso.

PCC-SA mejora uno de los principales problemas de PCC: el tamaño de la prueba. En la mayoría de los casos las pruebas de PCC son exponenciales respecto al tamaño del código. Mientras que en PCC-SA las pruebas son lineales respecto al tamaño del código. Esto se debe a la utilización de los esquemas de pruebas.

Una desventaja de PCC-SA sobre PCC es que en PCC-SA el consumidor de código debe generar el código objeto antes de ejecutar la aplicación, mientras que en PCC se envía un *assembly* de más bajo nivel. El costo de generar el código objeto en PCC sería menor que en PCC-SA.

2.1.2 Análisis Estático de Flujo de Control

Tradicionalmente la aplicación de las técnicas de análisis estático de flujo de control y de datos para garantizar seguridad se realizada localmente. Es decir, la información generada por los análisis realizados no se preserva en el código generado. Por ejemplo, Splint [5] es un compilador certificante que utiliza análisis estático de flujo de control. Para realizar los análisis requeridos y la posterior certificación se basa en anotaciones que deben ser introducidas en el código fuente por el programador. Splint garantiza que el código es seguro si él lo compiló. Pero, no genera ninguna evidencia para verificar tal afirmación.

Otro ejemplo (un ejemplo desde la perspectiva del consumidor) que se puede mencionar es la propuesta de J. Bergeron et al. [2] que propone decompilar binarios no confiables, para luego, construir el grafo de flujo de control y de datos (entre otros). Entonces se realizan diversos análisis estáticos sobre estos grafos para verificar la seguridad de los binarios. Este enfoque no asume que el consumidor de código este inmerso en un framework, por lo cual, la verificación se realiza sin ninguna información provista por el productor.

PCC-SA explota el echo de que existe una gran cantidad de análisis de flujo de control [5, 6, 7, 10, 12, 13, 14, 15], 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* - [5]. Si bien las técnicas de análisis de flujo de control y de datos requieren un esfuerzo mayor que el realizado por un compilador tradicional (que sólo realiza verificación de tipos y otros análisis simples de programas) el esfuerzo requerido, comparándolo con verificación formal de programas, es mucho menor.

Además, PCC-SA no necesita la asistencia de anotaciones y genera la información necesaria para que el consumidor de código pueda verificar la seguridad de los programas. Dicha verificación se puede realizar con un costo menor al necesario para que el productor genere la certificación.

3 El Prototipo de PCC-SA Utilizado

El prototipo de PCC-SA [1, 18] permite certificar propiedades de seguridad de gran importancia. Por ejemplo, certifica la ausencia de accesos inválidos a los arreglos, la ausencia de el accesos a punteros no inicializados, punteros null y certifica que no se libera posiciones de memoria referenciadas por mas de un puntero. Este prototipo toma como entrada un subconjunto de C (con algunas extensiones) y genera un *árbol sintáctico abstracto* (ASA) anotado con información del estado de cada punto del programa.

El prototipo de PCC-SA asegura que solo serán descartados aquellos programas que son ciertamente inseguros. Si no puede determinar la seguridad de un programa inserta verificaciones dinámicas que garantizan la ejecución segura del programa. Esto no sólo ocurre 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* dinámicos.

El prototipo puede ser usado en tres escenarios:

- El código producido esta destinado a ser ejecutado localmente por el productor, que en este caso es también su consumidor.
- El código producido esta destinado migrar y ser ejecutado en el entorno del consumidor.
- El consumidor obtiene código fuente no confiable y verifica su seguridad.

En el primer caso tanto el entorno de compilación y de ejecución son confiables. En el segundo y tercer caso, el único entorno confiable para el consumidor es el propio, dado que, el código puede haber sido modificado maliciosamente o no corresponder al supuesto productor. Estos tres posibles escenarios dan la pauta que el entorno del productor y del consumidor pueden ser usados independientemente del framework.

3.1 Comparación con Otros Prototipos

El tamaño de las pruebas generadas por el prototipo de PCC-SA es lineal con respecto a la longitud de los programas. Esto es una ventaja importante sobre otros entornos, tal como el prototipo de PCC Touchstone [16], que generan pruebas hasta de tamaño exponencial con respecto al programa. Además, se demostró que la complejidad de los algoritmos de PCC-SA que generan la certificación son lineales con respecto al tamaño de los programas. No sucede lo mismo en el caso de aquellas herramientas que utilizan un demostrador de teoremas (como Touchstone).

Si lo comparamos con javac, inserta una cantidad mucho menor de verificaciones dinámicas. Además no necesita la asistencia de anotaciones en el código, como es el caso de Splint [5], para realizar la verificación estática de la seguridad del código. Splint es un compilador certificante que utiliza análisis estático de flujo de control. Para realizar los análisis requeridos y la posterior certificación se basa en anotaciones que deben ser introducidas en el código fuente por el programador.

Los compiladores certificantes Special J [3], TIL [22], FLINT/ML [19] utilizan lenguajes ensambladores tipados para expresar el código intermedio, pero dicha información es eliminada luego de la generación de código. Los compiladores Popcorn [13] y Cyclone [9] son compiladores certificantes cuyo lenguaje de salida es el lenguaje ensamblador tipado TAL [13], por lo que incluyen información de tipos aún en el código objeto. En cambio, PCC-SA genera un árbol sintáctico abstracto anotado con la información de tipos necesaria para verificar las propiedades de seguridad. Otra innovación reside en la forma en que se genera esta información. Mientras que los primeros utilizan un sistema de tipos o un framework lógico, PCC-SA realiza análisis estáticos sobre el código intermedio generado. PCC-SA prueba que no se viole la política de seguridad siguiendo el flujo y utilizando las reglas semánticas que expresan formalmente la política de seguridad de sus constructores.

3.2 Experiencias Realizadas

Para analizar la eficiencia y eficacia, de utilizar análisis estático de flujo de control y de datos para garantizar la seguridad del código móvil, se realizó una serie de pequeños experimentos utilizando el prototipo de PCC-SA. Estos experimentos consistieron en utilizar como entrada del prototipo algunos programas y analizar aspectos como: tiempo de generación, tamaño de las pruebas y porcentaje de accesos a arreglos que no contienen chequeos en tiempo de ejecución. Estos programas se solicitaron a programadores ajenos al proyecto de PCC-SA. Estos programadores solo tenían como requisito proporcionar programas que manipularan arreglos. De esta forma se trató de obtener una muestra objetiva y medianamente representativa de programas que manipulan arreglos.

A continuación, en la figura 2, se presentan los resultados obtenidos del análisis del comportamiento del prototipo con algunos programas seleccionados. Se presenta el programa, el tamaño del programa y la prueba en bytes, el porcentaje de accesos a arreglos sin verificaciones en tiempo de ejecución, y el tiempo en segundos que le llevó realizar la generación del código y la verificación de la política de seguridad.

Los algoritmos de ordenación (figura 2) se seleccionaron para realizar las experiencias porque, si bien, son sencillos también son lo suficientemente complejos para analizar. Además son algoritmos muy utilizados. Se puede ver que para todos los algoritmos el tamaño de la prueba obtenida es menor al tamaño del programa. En el peor de los casos, el tamaño de la prueba representa alrededor del 20% del tamaño total del programa y la prueba. El prototipo tardo menos de 0.5 segundos en generar y certificar las condiciones de seguridad impuestas. Se puede decir que este tipo de certificaciones pueden ser realizadas en un lenguaje de programación real sin perder performance. El porcentaje de accesos a arreglos sin chequeos en tiempo de ejecución varía dependiendo el algoritmo de ordenación. En el único caso donde se insertan chequeos en tiempo de ejecución es en *Shell Sort*. Esto es debido a que la condición del ciclo tiene en cuenta los valores del arreglo lo cual lleva a la imposibilidad de calcular la cantidad de veces que se ejecuta el cuerpo del ciclo.

Programa	*Prog. y Prueba*	*T. Prog.*	*T. Prueba*	*Arreglos sin RTC*	*Tpo. de Comp.*	*Tpo. de Verif*
Insert Sort	1712 bytes	1381 bytes (88.67%)	331 bytes (11.33%)	100%	0.380 seg.	0.330 seg.
Bubble Sort	1922 bytes	1558 bytes (81.06%)	364 bytes (18.94%)	100%	0.429 seg.	0.331 seg.
Selected Sort	1795 bytes	1412 bytes (78.66%)	383 bytes (21.34%)	100%	0.430 seg.	0.323 seg.
Shell Sort	2197 bytes	1760 bytes (80.11%)	437 bytes (19.89%)	60%	0.433 seg.	0.371 seg.

Figura 2: Experiencias con Algoritmos de Ordenación.

Estos experimentos permiten suponer que si bien el costo de compilación es mayor que el de un compilador tradicional este framework brinda beneficios interesantes:

- En la mayoría de los casos detecta la seguridad del código intermedio sin incluir chequeos en tiempo de ejecución. Además, en los programas seleccionados, se observó que la mayoría de accesos a arreglos fue realizada mediante variables inductivas.
- El tiempo de compilación y verificación resultó lineal al tamaño de los programas usados.
- El tiempo de verificación es menor al tiempo de compilación.
- Genera un esquema de prueba cuyo tamaño es menor a la longitud de los programas.
- La complejidad de la verificación de las pruebas generadas (para estos programas) es lineal.

3.3 Observación Informal de Programas `C`

Con el fin de poder extraer conclusiones sobre ciertas características de programas reales se realizó una observación informal de programas `C`. Los programas seleccionados fueron: (1) el navegador de internet `Mozilla 1.7.8`, (2) el compilador de `C gcc 3.3.5`, (3) el procesador de texto `AbiWord 2.5.5` y (4) el `kernel 2.6.11` del sistema operativo `linux` utilizado por `Gentoo`. Todas estas aplicaciones tienen su código libre bajo licencia `GNU`.

En la figura 3 se pueden observar algunos de los resultados obtenidos de la observación realizada.

Los resultados permiten observar que en aplicaciones reales es un echo excepcional encontrar más de seis ciclos anidados en el código de una función. Hay que resaltar que se encontraron seis ciclos anidados en un solo archivo de cada aplicación analizada (además, es el mismo código en ambas aplicaciones). En la mayoría de los archivos, como muestra la tabla en la última columna, se encontró que el nivel de anidamiento ronda generalmente entre cero y tres.

También se puedo ver, en la mayoría de los casos, que la cantidad de variables utilizadas en el cuerpo de una función no supera las 50 variables. Hay que mencionar, que existen algunos métodos que tienen alrededor de 3000 líneas y que utilizan menos de 70 variables. Otros resultados, que no

Programa	Cantidad de Archivos	Cantidad de Líneas	Máximo Nivel de Anidamiento	Cantidad de Anidamientos					
				1	2	3	4	5	6
Mozilla	1365	782.275	6	4186	825	102	20	0	4
gcc	3435	827.385	4	3381	512	46	7	0	0
AbiWord	1601	861.335	6	4749	930	115	24	0	4
Kernel	5196	3.618.436	4	16994	1989	155	14	0	0

Figura 3: Resultados de la Observación Informal de Aplicaciones `C`.

se muestran el la tabla anterior, muestran que el máximo nivel de anidamiento no supera al nivel 10. Como así también que aproximadamente dos tercios de los ciclos corresponden a sentencias `for`. Si bien en `C` las sentencias `for` pueden ser utilizadas con una gran variedad de alternativas la mayoría (alrededor del 80%) corresponden al patrón de variables inductivas utilizado para acotar ciclos.

Estas observaciones permiten aseverar que efectivamente son ciertas las afirmaciones realizadas en cuanto a la efectividad del entorno. Además, acotar la cantidad de anidamiento de ciclos permite probar que la complejidad temporal de la verificación de las propiedades de seguridad se comporta de forma lineal con respecto a la longitud del código de entrada [1].

4 Conclusiones y Trabajos Futuros

Se presentó evidencia de los beneficios de utilizar análisis estático de flujo de control y de datos para garantizar la ejecución segura de código móvil. Está contribución se desprende de la experimentación con el framework *Proof-Carrying Code based-on Static Analysis* (PCC-SA) [1, 18]. Este framework, está basado en PCC y técnicas de análisis estático de flujo de control y de datos. Estos análisis balancean el costo entre el esfuerzo de diseño y la complejidad requerida.

Este framework provee características de gran relevancia: seguridad, independencia de la plataforma, verificaciones de seguridad simples, generación de pruebas de manera automática, pruebas pequeñas y provisión de la información necesaria para efectuar optimizaciones sobre el código. Cabe resaltar que una de las características más importantes de PCC-SA reside en el tamaño lineal de las pruebas (con respecto al programa producido). En la mayoría de los casos, el tamaño de las pruebas es menor al tamaño de los programas. Sólo en el peor caso la prueba tiene la misma longitud que el programa. No sucede lo mismo en PCC. No hay que dejar de notar que esta característica es fundamental en un ambiente de código móvil.

Una ventaja importante del prototipo de PCC-SA sobre otros prototipos de PCC, tal como Touchstone, reside en que la complejidad de los algoritmos de PCC-SA que generan la certificación son lineales con respecto al tamaño de los programas. Si lo comparamos con javac, inserta una cantidad mucho menor de verificaciones dinámicas. Además no necesita la asistencia de anotaciones en el código, como es el caso de Splint, para realizar la verificación estática de la seguridad del código. Aunque las técnicas de análisis estático requieren un mayor esfuerzo que los análisis realizados por un compilador tradicional, el esfuerzo requerido es significativamente menor que el requerido por la verificación formal de programas.

PCC-SA permite trabajar con políticas de seguridad más amplias que las que permite PCC y el Java ByteCode Verifier. Esto se debe al uso de análisis estático en lugar de sistemas de tipos.

Si bien el lenguaje fuente de Touchstone (un prototipo de PCC) es mas completo que el del prototipo de PCC-SA desarrollado, el lenguaje fuente del verificador puede ser fácilmente extendido sin mayores dificultades. Además, el núcleo mas relevante de Touchstone es equivalente al prototipo desarrollado. Las ventajas y desventajas del prototipo sobre Touchstone son similares a las de PCC sobre PCC-SA.

Si bien consideramos que aún queda mucho por investigar y desarrollar este es un primer paso, muy promisorio, para lograr un prototipo que permita generar aplicaciones móviles seguras basadas

en estas técnicas de análisis estático y PCC.

4.1 Trabajos Futuros

Se está trabajando en la extensión del prototipo de PCC-SA de modo que pueda ser usado para aplicaciones "reales". Si bien, como se mencionó anteriormente, se considera que aún queda mucho por investigar y desarrollar, este es un promisorio primer paso para para lograr un prototipo que permita generar aplicaciones móviles seguras basadas en análisis estático, PCC y verificaciones dinámicas. En esté momento se está extendiendo el lenguaje fuente y la política de seguridad, y se están incorporando nuevos análisis estáticos al entorno.

Se está estudiando la posibilidad de dividir el esquema de prueba en n esquemas de prueba que puedan ser verificados independientemente y de manera concurrente.

Un punto muy interesante por explorar consiste en incorporar aritmética de punteros y evaluar el alcance de los distintos análisis de flujo de control y datos para determinar la seguridad del código.

Se pretende determinar automáticamente la parte variante de los ciclos. Con esto se desea poder generar información que permita verificar que todos los ciclos progresan (o por lo menos la mayoría de ellos). Esta información es muy importante al momento de depurar los programas. Se están estudiando los análisis necesarios y la efectividad de los mismos.

Se desea estudiar los resultados del compilador si se aplican los análisis estáticos en cascada. Es decir, si una vez realizados todos los análisis se vuelven a realizar utilizando los resultados anteriores. Es necesario establecer la relación costo/beneficio de aplicar sucesivamente los análisis y cual es la cantidad de repeticiones promedio necesarias para obtener un alto grado de efectividad.

El análisis de llamadas a funciones esta basado en el análisis de cada función independientemente. Habría que analizar los resultados que traería aparejado analizar cada función cuando es invocada y con la información obtenida hasta ese punto.

Consideramos de mucho interés indagar en tres direcciones: análisis abstracto, lenguaje/s ensamblador/es tipado/s y especificación de políticas de seguridas por medio de autómatas.

Incorporando análisis abstracto al prototipo se lograría verificar nuevas propiedades de seguridad. Además, se contaría con la posibilidad de generar certificados que podrían ser verificados por un intérprete abstracto. Con esto se evitaría que el consumidor realice los mismos análisis que el productor de código.

Incorporando un lenguaje/s ensamblador/es tipado/s se espera poder definir un entorno basado en análisis estático (de control de flujo y datos) y en un sistema de tipos formal. Para esto pretendemos determinar qué propiedades se pueden verificar con cada uno de estos enfoques, o con cuál de ellos se pueden realizar más eficientemente.

Especificando las políticas de seguridad con autómatas [21] se obtendría flexibilidad en cuanto a las propiedades que se pueden verificar. En esta línea se está evaluando la posibilidad de verificar las propiedades de seguridad contrastando los autómatas (que especifican determinada propiedad) el flujo de control y de datos determinado por el ASA de cada programa.

Bibliografía

[1] F. Bavera. "Compilando y Certificando Código mediante Análisis Estático de Flujo de Control y de Datos". Tesis de Maestría. Maestría en Informática, INCO, Facultad de Ingeniería, Universidad de la República, Montevideo, Uruguay. En revisión final. 2005. http://dc.exa.unrc.edu.ar/docentes/pancho/fbavera/papers/tesis_maestria_FB.ps

[2] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, N. Tawbi. "Static Detection of malicious Code in Executable Programs". LSFM Research Group, Departament of Informatic, University of Laval, Canada. 2001.

[3] 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.

[4] Ulfar Erlingsson, Dexter Kozen. “SASI Enforecement of Security Policity: A Retrospective”. Reporte técnico. Abril de 1999.

[5] D. Evans y D. Larochelle. “Improving Security Using Extensible Lightweight Static Analisis”. IEEE Software. Enero/Febrero 2002.

[6] V. Haldar, C. Stork y M. Franz. “The Source is the Proof”. Departament of Computer Science, University of California. 2002

[7] C. Hanking y T. Jensen. “Security and Safety through Static Analisys”. Project homepage http://www.doc.ic.ac.uk/ siveroni/secsafe.

[8] William E. Howden. “Program Testing versus Proofs of Correctness”. En *Proc. de Software Testing, Verification and Reliability (STVR), Volume 1.* pp 5-15. 1992.

[9] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang. “Cyclone: A safe dialect of c”. En *USENIX Annual Technical Conference*, Monterey, California (EE.UU.). Junio 2002.

[10] D. Kozen. “Efficient Code Certification”. Tech. Report 98-1661, Cornell Univ.. 1998.

[11] D. Kozen. “Language-Based Security”. En M. Kutylowski, L. Pacholski, and T. Wierzbicki, editors, Proc. Conf. Mathematical Foundations of Computer Science (MFCS’99), Lecture Notes in Computer Science v. 1672, pp. 284-298, Springer-Verlag. 1999.

[12] Robert Morgan. Building an Optimizing Compiler. Editorial Digital Press. ISBN 1-55558-179-X. 1998.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic. “TALx86: A Realistic Typed Assembly Language”. En *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, ACM Press, Atlanta, Georgia (EE.UU.). Mayo 1999.

[14] Steven Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers. ISBN 1-55860-320-4. 1997.

[15] G. Necula, P. Lee. “Proof-Carrying Code”. Technical Report CMU-CS-96-165, Carnegie Mellon University. Noviembre 1996.

[16] G. Necula. “Compiling with Proofs”. PhD thesis, Carnegie Mellon University. Septiembre 1998.

[17] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum. “A Framework for Execution of Secure Mobile Code based on Static Analysis”. En *XXIV International Conference of the Chilean Computer Science Society*. Universidad de Tarapaca, Arica (Chile), Noviembre 2004, pp. 59–66. IEEE Computer Society Press. 2004.

[18] M. Nordio. “Verificación de la Seguridad del Código Foráneo mediante Análisis Estático de Control de Flujo y de Datos ”. Tesis de Maestría. Maestría en Informática, INCO, Facultad de Ingeniería, Universidad de la Republica, Montevideo, Uruguay. Abril 2005.

[19] Z. Shao. “An Overview of the FLINT/ML Compiler”. En *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation* (TIC’97), ACM Press, Amsterdam (Holanda). Junio 1997.

[20] F. Smith, D. Walker, G. Morrisett, “Alias Types”. En *Gert Smolka, editor Ninth European Symposium on Programming*, volume 1782 of Incs, pp. 366–381, Spring-Verlag. 2000.

[21] Fred B. Scheneider. “Enforceable security policies”. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department. Septiembre 1998.

[22] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee. “TIL: A Type-Directed Optimizing Compiler for ML”. En *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI’96), pp. 181–192, ACM Press, Philadelphia, Pennsylvania (EE.UU.). Mayo 1996.

[23] R. Wahbe, AS. Lucco, T. Anderson y S. Graham. “Efficient Software-based Fault Isolation”. En *Proc. 14th Symp. Operating System Principles*, pp 203-216. ACM. Diciembre de 1993.

[24] H. Xi, R. Harper. “A Dependently Typed Assembly Languaje”. Reporte Técnico, OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology. 1999.

[25] Stuart H. Zweben, Wayne D. Heym, Jon Kimmich. “Systematic Testing of Data Abstractions Based on Software Specifications”. En *Proc. de Software Testing, Verification and Reliability (STVR), Volume 1*. pp 39-55. 1992.