Avances en la línea de investigación sobre PCC del grupo de Procesadores de Lenguajes y Métodos Formales de la UNRC

Departamento de Computación. Univesidad Nacional de Río Cuarto[1]
Laura Córdoba *lauracastelino@hotmail.com*
Laura Tardivo *lauratardivo@dc.exa.unrc.edu.ar*
Francisco Bavera *fbavera@dc.exa.unrc.edu.ar*
Marcelo Arroyo marroyo@*dc.exa.unrc.edu.ar*
Jorge Aguirre jaguirre@*dc.exa.unrc.edu.ar*

En este trabajo se presenta una de las líneas del grupo de investigación "Procesadores de Lenguajes y Métodos Formales", perteneciente al Departamento de Computación de la UNRC. El aspecto fundamental de dicha línea es la creación de un framework para proponer soluciones alternativas que contribuyan al logro de técnicas de generación y ejecución eficiente de Código Móvil Seguro. Se presenta a C-Metric, una herramienta para medir métricas sobre programas C. C-Metric cuenta con un analizador sintáctico y semántico de programas C, que genera un árbol sintáctico abstracto sobre el cual se evaluan las diferentes métricas. La construcción de dicho árbol contempla todo el proceso a que somete al programa fuente el entorno de compilación C (preprocesador, bibliotecas estándar, etc).

Introducción

El desarrollo de la computación y las telecomunicaciones han permitido que las dos tecnologías estén integradas tanto sistemas de computación como en otros dispositivos de uso corriente. Muchos de esos dispositivos son programables y pueden recibir programas y actualizaciones desde fuentes externas. Entre los ejemplos mas visibles se pueden mencionar celulares, computadoras de mano, tarjetas inteligentes, sistemas de computación distribuidos, basados en agentes móviles y ciertas tecnologías alrededor del lenguaje Java.

Esta tecnología permite una gran flexibilidad, pero el hecho que un dispositivo huésped deba ejecutar componentes de software recibidos de fuentes externas plantea serios problemas, principalmente relativos a su seguridad. Uno de los problemas a resolver es la detección de código intrusivo o maligno que intente comprometer algún recurso del sistema huésped o receptor. Es bien conocido el daño que pueden hacer aquellos programas llamados "virus" en ciertas aplicaciones y es fácil imaginar los efectos nocivos que puede acarrear una actualización remota masiva de software, en cualquiera de los campos mencionados.

Existen diversos enfoques para superar los problemas de seguridad ocasionados por el Código Móvil, algunos basados en la autenticación del emisor, usando criptografía, otros en verificaciones dinámicas y otros en verificaciones estáticas, dentro de este último se distingue la técnica de Proof Carrying Code (PCC), que permite a un consumidor de código determinar fehacientemente y de manera estática si un programa se comportará de manera segura para su entorno, antes de ejecutarlo. PCC es una técnica introducida por G. Necula y P. Lee [Nec96] para garantizar código móvil seguro mediante la demostración de que el código recibido satisface la política de seguridad impuesta por el receptor. PCC ha generado una activa línea de investigación, con una importante producción, que se analiza en el *survey* producido en el grupo [Bav04]. Dicha línea de investigación aún presenta muchos problemas abiertos y dificultades a resolver para lograr una efectiva aplicación industrial.

La línea de investigación que enmarca a este trabajo se ocupa de aplicar al PCC las técnicas del Análisis estático de flujo de datos, en busca de soluciones con comportamiento lineal.

[1] Este trabajo ha sido realizado en el marco de un proyecto subsidiado por la SCyT de la UNRC.

Al comienzo del desarrollo de esta línea el grupo obtuvo un framework para la generación de Código Móvil Seguro basado en Análisis Estático y un prototipo de los entornos de compilación y ejecución para un subconjunto de C (CCMini) definido ad hoc. Durante el transcurso del trabajo se obtuvieron dos tesis de maestría y varias publicaciones [Bav04][Bav05][Bav06][Arr05][Nor04]. Se tenía el convencimiento de que el comportamiento temporal del framework era lineal respecto del tamaño del programa de entrada, y se había verificado que así era para una muestra de programas escritos por voluntarios en CCMini. Era necesario verificar que esto se cumpliría para una gran muestra de programas reales. Se requería entonces trabajar con bibliotecas C estándar abandonando el prototipo realizado. Para ello se logro caracterizar mediante una métrica dependiente de pocos valores estáticos a una familia de programas (se los llamó Programas Linealmente Anotables [PLA]), para los cuales se demostró que el tiempo de compilación y verificación del framework es lineal respecto del programa fuente. Luego se calcularon mediante distintos procedimientos (usando awk y otros herramientas del ambiente Unix) los parámetros requeridos para caracterizar la pertenencia a la familia PLA del kernel de Linux y otras bibliotecas de GNU (con más de cuatro millones de líneas código) verificándose que todos ellos eran PLA. Luego se inició la construcción de una herramienta general para medir estáticamente métricas sobre programas C, a la que se denominó CMetric, tarea que es descripta en el presente trabajo. Existen diversos trabajos sobre el análisis estático de programas C con otros objetivos. Uno de ellos es Lint [Joh86], un program checker de C que examina archivos fuente para detectar y reportar incompatibilidades de tipos, inconsistencias entre definiciones de funciones y llamadas a función, potenciales errores de programación, etc. Lint fue desarrollado para superar la falta de verificaciones de los primeros compiladores de C y recientemente ha evolucionado en SPLint [Eva02], el cual permite verificar estáticamente la validez de programas con respecto a aserciones introducidas por el usuario. Además la herramienta es extensible, permitiendo al usuario incorporar otros tipos de análisis no cubiertos inicialmente por la herramienta.

La magnitud del desarrollo de CMetric superó ampliamente lo previsto; debido a la complejidad del entorno de compilación C y a la vastedad de los programas aceptados por la gramática de C (que ha sorprendido a miembros del grupo de soporte de GNU a los que se les ha mostrado los resultados).

Seguridad en la programación en C

El lenguaje de programación C surge en la década del 70 para dar portabilidad al Sistema Unix, época en la que surgieron diversos lenguajes destinados a la programación de bajo nivel, hasta entonces realizada en assembly, como BCPL, STAB, BLISS y otros. Desde entonces, el lenguaje ha cambiado de manera considerable. En 1983 el American National Standards Institute (ANSI) estableció un comité con el objetivo de construir una definición del lenguaje de programación C no ambiguo e independiente de la arquitectura. Como resultado se obtuvo el ANSI C.

Desde sus orígenes hasta nuestros días, el lenguaje de programación C ha sido utilizado en diferentes ambientes de desarrollo y en sistemas computacionales simples y complejos, incluyendo navegadores, editores de texto y sistemas operativos modernos.

Si bien son muchas las posibilidades que provee C, también es cierto que el lenguaje y las librerías que soporta pueden permitir a los programadores agregar vulnerabilidades de seguridad a su código de manera inadvertida por ellos.

La construcción de sistemas seguros en C involucra un desafío realmente importante por parte de los programadores. Gran parte de los problemas que comprenden ataques de seguridad están relacionados con defectos en la implementación del software. Accesos a arreglos ilegales,

operaciones inválidas con punteros, violaciones al sistema de tipos, pueden provocar la ejecución inadecuada de un programa e incluso provocar fallas incalculadas.
Si sabemos de la existencia de todas estas posibles fallas, podemos preguntarnos por qué siguen ocurriendo. Tal vez pueda deberse a la falta de conocimiento de los programadores acerca de lo que debe ser correcto como solución. Tal vez ni siquiera son concientes de los posibles problemas hasta haber escrito el código por completo y haber ejecutado algunos casos de prueba para los cuales el sistema falla. En éste último caso estamos hablando del testing del sistema por ensayo y error, pudiendo resultar flujos de programa sin chequear. Estas situaciones evidenciadas durante el desarrollo de CMetric concurren a dar evidencias de la necesidad de contar con soportes automátcos de la seguridad de código que se ejecuta.

Técnicas de Chequeo
Existen dos grandes grupos de técnicas de análisis para detectar errores de funcionamiento de los programas: las técnicas de análisis dinámico y las de análisis estático.
Las técnicas de análisis dinámico introducen chequeos en tiempo de ejecución para reducir el riesgo de que el código posea vulnerabilidades de seguridad. Una de las desventajas de estas técnicas es el aumento del tiempo de ejecución de los programas por la sobrecarga introducida por las verificaciones. Otras técnicas de análisis dinámico requieren hardware especial y un sistema operativo relativamente complejo que lo soporte.
Las técnicas de análisis estático tienen un enfoque distinto. Permiten obtener una aproximación concreta al comportamiento dinámico de un programa antes de ser ejecutado. Existe una amplia variedad de técnicas de análisis estático. Podemos encontrar desde compiladores tradicionales que con poco esfuerzo realizan verificaciones de tipos, hasta verificadores de programas que requieren especificaciones formales completas y utilizan demostradores de teoremas.
C-Metric utiliza las técnicas de análisis estático para computar las métricas sobre los programas.

Analizador de métricas C-metric.
Para realizar el analizador de métricas sobre programas C, se comenzó por obtener la gramática del lenguaje. En una primera instancia, se verificó que la gramática obtenida fuese correcta. Para ello se realizó una inspección sobre una muestra de cuatro millones de líneas código integrada por bibliotecas de kernel Linux 2.6.11 gentoo, fuentes de abiword 2.5.5, Mozilla 1.7.8, GCC 3.3.5 y open office 1.4.4.

La herramienta que se eligió para la implementación de C-metric utiliza el formalismo de *Definiciones Guiadas por Sintaxis*. El uso de este formalismo permitió luego establecer una correspondencia directa entre cada constructor sintáctico y su correspondiente implementación. Se utilizó entonces el generador de analizadores sintácticos Yacc en conjunto con el analizador lexicográfico Lex.
Para la implementación del C-Metric se utilizó el método de desarrollo Top Down, partiendo de una clara especificación de las estructuras de datos y llegando al código mediante refinamientos sucesivos de pseudo-código.
Mediante el estudio de la representación del sistema de tipos de C, se establecieron las estructuras de datos necesarias que sirvieron para almacenar los datos relativos a identificadores de variables, nombres de procedimientos y tipos definidos por el usuario (estructuras, uniones o enumeraciones, e incluso las redefiniciones de tipo introducidas por la sentencia ***typedef***). Se eligió representar cada bloque definido en el programa con una tabla de identificadores para ese bloque.

Para comenzar el diseño de la herramienta se hizo necesario un estudio completo del sistema de compilación de C, incluyendo la interface entre el preprocesador y el compilador C.

Etapas en la compilación de un programa C.

El proceso de compilación involucra cuatro etapas sucesivas: preprocesamiento, compilación, ensamblado y *link-edition.*.
En la etapa de preprocesamiento se transforma el código fuente original en código fuente puro. Es decir, se expanden las macros, se incluyen librerías, se realiza un preprocesado racional (se enriquece el lenguaje antiguo con recursos mas modernos), se extiende el lenguaje y todo aquello que en el código de entrada sea representativo de una abreviatura para facilitar la escritura del mismo.
En la etapa de compilación se somete el código fuente puro a un análisis lexicográfico, un análisis sintáctico y un análisis semántico, se genera código intermedio que se optimiza para poder producir código de salida en algún lenguaje ensamblador.
La etapa de ensamblado recibe el código fuente compilado y produce el llamado código objeto, un archivo binario en lenguaje de máquina ejecutable por el procesador.
La última etapa se encarga de vincular el código objeto con los módulos que necesita, produciendo como salida el código binario ejecutable, ya sea dinámico (los módulos se cargan en memoria y se ejecutan en run-time) o estático (los binarios de la funciones se incorporan al código binario ejecutable).

Implementación del parser de C
La implementación del parser de C consistió en definir qué acciones semánticas debían ejecutarse en cada producción de la gramática. Para ello, se analizó qué sentencia del lenguaje generaba cada producción o conjunto de producciones, verificando en cada caso qué chequeos de tipos define el ANSI C. Durante el recorrido del parser por la gramática, C-metric genera una estructura de datos con la definición de los tipos del programa fuente.

Detección de errores
Durante la etapa de compilación existen dos tipos de errores que pueden clasificarse en errores sintácticos y errores semáticos.
Gracias a las posibilidades de la herramienta elegida para el desarrollo del parser, los errores de sintaxis son detectados de manera automática.
Los errores de tipo son reportados en la medida en que se violen las restricciones de tipo que especifica el ANSI C. Cada acción semántica, además de actualizar las estructuras de datos del parser, realiza un chequeo sobre los tipos especificados para las sentencias del programa fuente.
Si alguna restricción es violada, se informa la línea del error y una descripción de la naturaleza del mismo.
El algoritmo de equivalencia de tipos utilizado en la verificación del sistema de tipos respeta el Sistema de Tipos de C realizando comparación estructural sobre los tipos básicos y comparación por nombre sobre las uniones, enumeraciones y estructuras. La especificación es la siguiente:
- Dos *listas de especificadores de tipo* son equivalentes si contienen el mismo conjunto de *especificadores* de tipo, teniendo en cuenta que algunos *especificadores* pueden ser implicados por otros (por ejemplo, long es implicado por long int). Estructuras, uniones y enumeraciones con diferentes nombres constituyen tipos distintos, y aquellos que no posean nombre definen un tipo único.
- En general, *dos tipos* son equivalentes si sus *listas de especificadores de tipos* coinciden estructuralmente luego de expandir cualquier renombre introducido por la sentencia typedef, y luego de eliminar los *especificadores* (identificadores) de parámetro de función. Las dimensiones de los arreglos y el tipo de los parámetros son significativos.

Generación del árbol de sintáctico abstracto.

Un árbol sintáctico abstracto es una representación abstracta del código fuente que jerarquiza la estructura semántica y permite realizar distintos análisis estáticos, tales como análisis de flujo de control y análisis de flujo de datos. También puede ser utilizado para generar una gran cantidad de optimizaciones del código.
Cada sentencia de la gramática esta representada por un nodo en el árbol, el cual además posee toda la información correspondiente a dicha sentencia.
Durante el parsing del programa fuente, se va construyendo el árbol sintáctico abstracto, sobre el cual se realizarán diferentes inspecciones.
La construcción del árbol sintáctico abstracto da la posibilidad de establecer una independencia con respecto al parser en si, y además es posible extender las funcionalidades del C-Metric permitiendo que los usuarios definan sus propias métricas con un mínimo conocimiento del lenguaje C.

Referencias

- [Nec96] G. Necula, P. Lee, "Proof – Carrying Code", Technical Report CMU – CS – 96 – 165, Carnegie Mellon University, Noviembre 1996.
- [Nec96] G. Necula, P. Lee, "Safe Kernel Extensions Without Run-Time Checking", Second Symposium on Operating System Design and Implementation (OSDI '96), Seattle, 1996.
- [Joh86] S.C. Johnson, Lint, a C Program Checker, USENIX UNIX Supplementary Documents, November 1986.
- [Nor04] A Framework for Execution of Secure Mobile Code based on Static Analysis. N. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum. XXIV International Conference of the Chilean Computer Science Society (SCCC 2004). Arica Chile, noviembre de 2004. ISBN 0-7695-2185-1,pags 59-66, IEEE-CS PRESS .
- [Bav06] Secure Mobile Code and Control Flow Analysis. F. Bavera, J. Aguirre, M. Nordio. XII Congreso Argentino de Ciencias de Computación, Anales del CACIC 2006. UNSL San Luis 17 al 21 de octubre de 2006, pags. 1813-1824.
- [Bav05] CCMini: A Prototype of Certyfing Compiler based on Annotated Abstract Syntax Tree. Francisco Bavera, Martín Nordio, Ricardo Medel, Jorge Aguirre, Gabriel Baum. Workshop de Ingeniería de Software y Base de Datos (WISBD), Congreso Argentino de Ciencias de la Computación, CACIC 2005. Universidad Nacional de Entre Ríos, Concordia (Argentina). 17-20 de Octubre, 2005.
- [Bav04]Un Survey sobre Proof-Carrying Code, F. Bavera, N. Nordio, R. Medel, J. aguirre, G. Baum. 33 JAIIO 2004, SADIO, FAMAF, Córdoba, Septiembre 2004.Publicación en CD, ISSN 1666 1141.
- [Bav04] Optimización del Prototipo del Entorno de Ejecución de PCC-SA. Francisco Bavera, M. Nordio, J. Aguirre, G. Baum, R. Medel. Publicado en Anales del X Congreso Argentino de Ciencias de la Computación, CACIC 2004.Universidad de la Matanza, Buenos Aires, Argentina. Octubre 2004.
- [Eva02] D, Evans and D. Larrochelle. "Improving Security Using Extensible Lightweight Static Analysis." IEEE Software, Jan-Feb 2002.