Optimización del Prototipo del Entorno de Ejecución de PCC-SA*

Francisco Bavera[1], Martín Nordio,[1] Ricardo Medel,[1,2,†]
Jorge Aguirre[1], Gabriel Baum[1,3]

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

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

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

Resumen

Proof-Carrying Code based on Static Analysis (PCC-SA) es un entorno de ejecución de código móvil seguro. PCC-SA combina *Proof-Carrying Code* (PCC) y análisis estático con el fin de 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 alto nivel, un *árbol sintáctico abstracto* (ASA) anotado con información de tipos. Este tipo de representación intermedia permite realizar diversos análisis estáticos para generar y verificar la información necesaria, y gran cantidad de optimizaciones al código generado. La principal ventaja de esta técnica 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 presenta el diseño de las optimizaciones del prototipo de PCC-SA (desarrollado originalmente con el solo fin de probar la factibilidad de PCC-SA). La meta de estas actividades es realizar un primer paso hacia la obtención de un ambiente de ejecución de código móvil seguro que pueda ser usado industrialmente.

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

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

†El trabajo de este autor ha sido subsidiado por la NSF en el marco del proyecto #0093362 – *CAREER: A Formally Verified Environment for the Production of Secure Software.*

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 la seguridad del código móvil. Una técnica que despertó mucho intéres en el último tiempo es *Proof-Carrying Code* (PCC) [14]. PCC es una técnica propuesta por Necula y Lee [14] para garantizar código móvil seguro. Esta técnica ha generado activas líneas de investigación con una importante producción [4, 5, 6, 8, 10, 14, 15, 16, 18, 20], pero aún presenta muchos problemas abiertos. 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) [17] es un entorno de ejecución de código móvil seguro. PCC-SA combina PCC y análisis estático con el fin de 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. 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 [12] 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 un código intermedio de más alto nivel, un *árbol sintáctico abstracto* (ASA) 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 gran cantidad de 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.

Con el fin de corroborar la factibilidad de este enfoque se desarrolló un prototipo del entorno PCC-SA. Este prototipo permitió concluir que el enfoque es una muy buena alternativa para brindar una solución efectiva y eficiente para el problema del código móvil seguro. El prototipo toma el programa fuente y lo traduce en un árbol sintáctico abstracto (ASA) sobre el cual se realizan distintos análisis de flujo de control y de datos. Estos análisis estáticos permiten obtener una aproximación concreta del comportamiento dinámico del programa antes de ser ejecutado. Con los resultados obtenidos se verifica la seguridad del código y se construye un esquema de prueba. El esquema de prueba es enviado al consumidor de código (junto con el ASA). Si a partir del esquema de prueba se puede verificar la seguridad del codigo, entonces el codigo asi certificado se puede ejecutar en forma segura.

Cabe resaltar que en muchos casos es necesario insertar verificaciones dinámicas, no sólo por las limitaciones de un análisis estático particular, sino porque los problemas a resolver son no computables. 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. Es decir, el mecanismo usado para verificar que el código es seguro es una combinación de verificaciones estáticas - en tiempo de compilación - y de verificaciones dinámicas - en tiempo de ejecución -. Con esto se amplía el rango de programas seguros aceptados, incluyendo aquellos sobre los cuales el análisis estático no puede asegurar nada.

Este prototipo de PCC-SA es un primer paso a la obtención de un entorno de ejecución para código móvil seguro que pueda ser usado industrialmente. Por lo cual es necesario seguir desarrollandolo. En este trabajo se plantean una serie de optimizaciones tendientes a obtener un entorno que pueda ser una alternativa potencial para ser usada en ambientes con código móvil inseguro. Las optimizaciones tienen como fin proporcionar un ambiente que produzca código más compacto y eficiente.

Este trabajo está estructurado de la siguiente manera: en la sección 2 se presenta el prototipo

del entorno de ejecución PCC-SA, junto con una breve descripción de sus componentes y un breve análisis de la factibilidad de este prototipo. En la sección 3 se presenta la optimización realizada al *Generador de Anotaciones*. Se diseñó un nuevo algoritmo para calcular las cotas de las variables, el cual es presentado en esta sección. En la sección 4 se presentan las optimizaciones realizadas en pos de reducir el tamaño del esquema de prueba y de minimizar las actividades que debe realizar el consumidor para verificar la seguridad del código. En la sección 5 se presentan las optimizaciones que se realizan al código generado (el ASA). Los trabajos relacionados se presentan en la sección 6. En la sección 7 se enuncian las conclusiones y trabajos futuros.

2 El Prototipo de PCC-SA

La figura 1 muestra la interacción de los componentes del prototipo. Siguiendo la notación de Necula [14] 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 [17] para más detalles de PCC-SA y del prototipo desarrollado.

Figura 1: Vista global del prototipo del entorno de ejecución de *Proof-Carrying Code based on Static Analysis*.

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 arbol sintactico 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 politica 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 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*. 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. 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*.

2.1 Ejemplo del Proceso Realizado por el Prototipo

En la figura 2 se puede ver un programa fuente Mini. El mismo es una función que luego de inicializar un arreglo retorna la suma de sus elementos.

```
int ArraySum ( int index(0,0) ) {
    int [10] data;          /* Define el Arreglo*/
    int value=1;            /* Define una variable para inicializar el Arreglo */
    int sum=0;              /* Define la variable sumatoria que calcula la
                               sumatoria del Arreglo */
    while (index<10) {      /* Inicializa el arreglo */
        data[index]=value;
        value=value+1;
        index=index+1;
    }
    while (index>0) {       /* Calcula la sumatoria */
        sum=sum+data[index-1];
        index=index-1;
    }
    return sum;
}
```

Figura 2: Ejemplo de un programa Mini.

Como puede apreciarse, el lenguaje Mini tiene una sintaxis similar al lenguaje C. Sin embargo, debido a que se busca analizar el comportamiento de los accesos a arreglos en los programas, se definió un lenguaje cuya característica principal es la manipulación de arreglos. Otras características de C, irrelevantes para el presente trabajo, no fueron incluídas en Mini a fin de utilizar un lenguaje lo mas simple posible.

Básicamente, un programa Mini es una función que toma al menos un parámetro y retorna un valor. Tanto los parámetros como los valores que retorna deben ser de uno de los tipos básicos (enteros y booleanos). Además, cuenta con arreglos unidimensionales, cuyos elementos pueden ser de un tipo básico. Cabe aclarar que la complejidad de utilizar arreglos unidimensionales es la misma que si se incluyen estructuras de datos más complejas, tales como matrices.

Originalmente Mini era un subconjunto de C, pero luego fue extendido a fin de permitir acotar los parámetros de tipo entero. Para esto, la declaración del parámetro va acompañada por un rango que indica el menor y el mayor valor que puede tomar el argumento de entrada. Por ejemplo, si el perfil de una función es `int func(int a(0,10))`, entonces el valor del parámetro `a` verificará la condición $0 \leq \mathtt{a} \leq 10$. Con esta extensión se pretende incrementar la cantidad de variables y expresiones acotadas que intervienen en los programas. Para una descripción mas detallada del lenguaje fuente, consultar [7].

El prototipo, específicamente el compilador, toma el programa y genera un *árbol sintáctico abstracto* (ASA). En la figura 3 se puede ver el ASA generado para el programa de la figura 2. Ésta es una representación abstracta del código fuente y permite realizar distintos análisis estáticos, tales como análisis de flujo de control y análisis de flujo de datos. También puede ser utilizado para realizar gran cantidad de optimizaciones sobre el código.

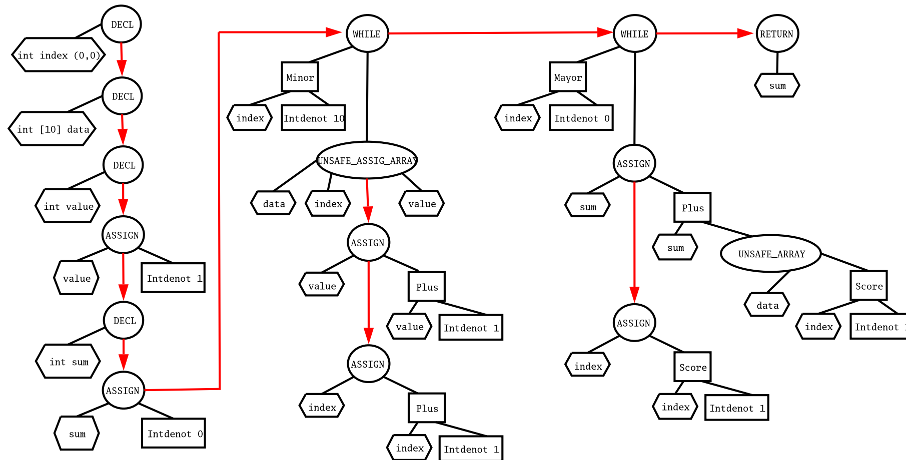

Figura 3: ASA del programa Mini de la figura 2.

La estructura de los árboles sintácticos abstractos (ASA) que se utiliza es similar a la de los ASA's tradicionales, aunque aquéllos no permiten incluir anotaciones de código. Estas anotaciones reflejan el estado de los objetos que intervienen en el programa. Por ejemplo, contienen información sobre inicialización de variables, invariantes de ciclos y cotas de variables.

Cada sentencia está representada por un ASA. Los nodos de una sentencia, además de la etiqueta que lo caracteriza, contienen información o referencias a las sentencias que la componen y una referencia a la sentencia siguiente. Cada expresión es representada por un grafo.

En la figura 3 los círculos representan sentencias, los hexágonos variables y los rectángulos expresiones. Las flechas representan flujo de control y las líneas representan atributos de las sentencias. Las declaraciones se representan en el ASA con la etiqueta DECL. Las asignaciones con ASSIGN, las asignaciones de arreglos con UNSAFE_ASSIG_ARRAY, las sentencias iterativas con WHILE y el retorno con RETURN. Por ejemplo, podemos observar que el primer ciclo esta constituído por una condición (la expresión $index < 10$) y el cuerpo del mismo. El cuerpo del ciclo está formado por tres asignaciones. En la primera se le asigna al arreglo $data$ en la posición $index$ el valor de la variable $value$. En la segunda asignación a la variable $value$ se le asigna la expresión $value + 1$.

Se utilizan dos etiquetas distintas cuando se hace referencia al acceso a un arreglo: `unsafe` y `safe`. Estas etiquetas significan que no es seguro el acceso a ese elemento del arreglo y que es seguro el acceso, respectivamente. Esta manera de representar los accesos a arreglos permite eliminar las verificaciones dinámicas con sólo modificar la etiqueta del nodo.

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. Este camino de prueba es construido sobre el ASA.

En la figura 4 se muestran algunas de las anotaciones y el esquema de prueba producido para el programa de la figura 2. Los rectángulos grises representan anotaciones y las líneas de puntos el camino de prueba generado. La anotación $index(0, 9)$ significa que el valor de la variable $index$ se encuentra entre 0 y 9. Las anotaciones INV : representan las invariantes de los ciclos. Por ejemplo, INV : $index(0, 9)$ significa que la invariante de ciclo es $0 \leq index \leq 9$. El predicado $Inductive(index, 1)$ significa que la variable $index$ es inductiva y se incrementa en 1 en cada iteración. Una variable es inductiva si se incrementa (o decrementa) en una constante en cada iteración del ciclo y este incremento (decremento) se realiza en un solo lugar del ciclo.

Nótese que las figuras 3 y 4 difieren en las etiquetas de los nodos correspondientes a accesos a arreglos. En la primera figura los accesos se consideraban inseguros, pero dado que GenAnot pudo determinar que estos accesos son seguros, entonces en la figura 4 las etiquetas de estos nodos fueron

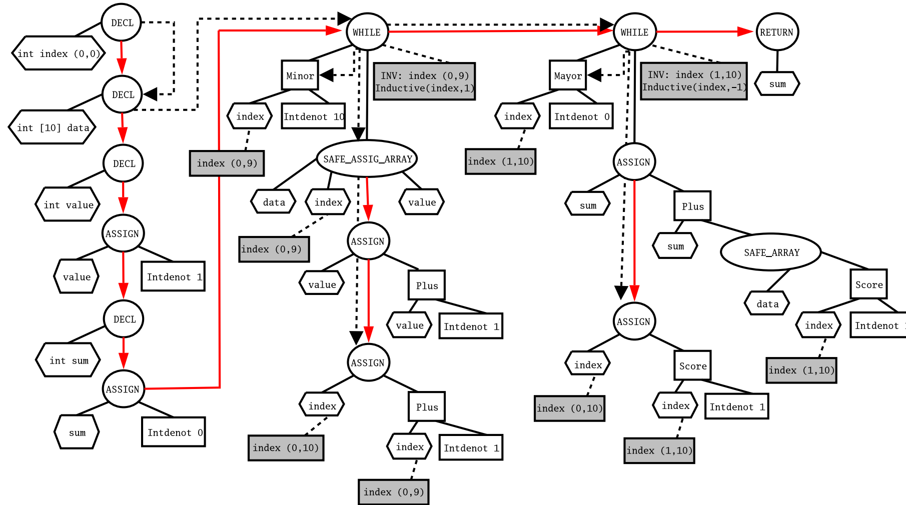

Figura 4: Esquema de Prueba del ASA anotado para el programa Mini 2.

modificadas para indicar la eliminación de la verificación dinámica.

El ASA anotado de la figura 4 es enviado al consumidor el cual, luego de verificar la consistencia del ASA recorre el esquema de prueba verificando la seguridad. Por último, realiza un recorrido exhaustivo del ASA corroborando que no se haya omitido ningún punto crítico.

2.2 Análisis de la Factibilidad del Uso del Prototipo

El prototipo desarrollado muestra la posibilidad de utilizar técnicas de análisis de flujo de control y flujo de datos en conjunción con las ideas de PCC para la implementación de entornos de ejecución para código móvil seguro.

Este prototipo provee características 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 verifica el código estáticamente. Cabe resaltar que una de las características más importantes del prototipo 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.

El prototipo sólo son descartados aquellos programas para los cuales se pudo determinar formalmente que su comportamiento es inseguro. En los otros casos, introduce verificaciones dinámicas en todos los puntos de seguridad incierta. Sin embargo, una observación informal de programas seleccionados al azar muestra que generalmente los accesos a arreglos se realizan mediante variables inductivas, lo cual sugiere que se puede determinar en la mayoría de los casos prácticos la validez de los accesos, evitando así la necesidad de insertar chequeos dinámicos.

El compilador certificante, denominado CCMini, puede ser utilizado independientemente del prototipo PCC-SA.

A continuación se enumeran las principales desventajas del prototipo: (1) al igual que otras implementaciones de PCC, es muy sensible a los cambios de las políticas de seguridad; (2) dado que PCC es un proceso cooperativo, el productor de código debe estar involucrado en la definición de la seguridad del consumidor de código; (3) establecer la política de seguridad es una tarea costosa; (4) garantizar la correctitud de la arquitectura es un procedimiento muy costoso y (5) traducir una propiedad de seguridad al análisis estático que la verificará no es trivial.

3 Optimización del Generador de Anotaciones: GenAnot

A diferencia de otros compiladores certificantes, el componente **GenAnot** genera las anotaciones del código y realiza las verificaciones simultáneamente. Para lograr esto se realizan análisis de flujo de control y de datos sobre el árbol sintáctico abstracto (ASA). Estos análisis determinan la inicialización y el rango de las variables utilizadas. Esto último permite asegurar si un valor es válido como índice de un arreglo. Además, en ciertos casos se determina las invariantes de ciclo y la precondición y postcondición de las funciones.

La implementación realizada de este módulo utiliza una *tabla de símbolos* como estructura de datos auxiliar para efectuar los análisis. Esta *tabla de símbolos* está implementada sobre una tabla de hash y almacena información de las variables. Cada variable esta indexada por su nombre (el índice esta dado por la aplicación de una función de hash). Cuando se recupera información de alguna variable se debe aplicar la función de hash para obtener el índice y luego realizar una busqueda lineal entre los elementos de igual índice.

Para reducir el tiempo de búsqueda y el espacio de memoria que se requiere cuando se utiliza esta estructura se propone utilizar como estructura de datos una *pila de contextos* de cada variable. Cada elemento de estas *pilas de contextos* contienen información del estado de cada variable en determinado contexto.

Cada ocurrencia de variable referenciará información de tipo, estado de inicialización y una pila de contextos (en la que cada elemento tendrá información del estado de la variable en cada contexto). Por ejemplo, variables enteras mantendrán información de cotas.

En la figura 5 se presenta el algoritmo para calcular cotas con esta nueva estructura. Se considera que: (1) las constantes enteras son valores con rango definido y están acotados por su valor; (2) inicialmente el rango de las variables no está definido, salvo en el caso de los parámetros acotados; y (3) inicialmente todos los accesos a arreglos son inseguros. Además, se utilizan las variables auxiliares **nivel** y **variables**. La primera es de tipo entero y mantendrá información del nivel de bloques abiertos y la segunda es un vector de vectores de referencias a variables que mantendrá información de las variables que son modificadas en cada nivel abierto.

Básicamente, el algoritmo recorre el ASA y en cada asignación define la cota de la variable a la que se le realiza la asignación. La *pila de contextos* se utiliza cuando el flujo del programa analizado se divide en ramas. Por ejemplo en el caso de un `if` se identifican los rangos de las variables en las dos ramas (la rama del cuerpo del `then` y la rama del cuerpo del `else`). El rango de las variables para el resto de programa está definido por la unión de los rangos de las variables de la rama del cuerpo del `then` y la rama del cuerpo del `else`. Donde la unión realiza lo siguiente para cada variable: si en las dos ramas la variable está acotada el rango resultante es el conformado por el valor mínimo y el valor máximo de los rangos originales. En el caso de que alguna de las dos variables tengan un rango indefinido el resultado es un rango indefinido.

En el caso de los accesos a arreglos, si la cota del índice esta dentro de los límites del arreglo entoinces se marca el acceso como seguro. Si la cota excede los límites del arreglo entonces se rechaza el programa por ser inseguro. Y si no se pudo determinar la cota del índice no se realiza ninguna acción.

Esta optimización redunda en una mejora en el tiempo de ejecución del **GenAnot**. Esto se debe a que por cada ocurrencia de una variable solo debemos acceder al tope de su pila para saber sus cotas. En el prototipo original primero se debe computar la función de hash para encontrar la posición de la tabla donde buscar la variable. Luego de una busqueda lineal en esa posición entonces se accede a la cota de la variable buscada. Además, la estructura de datos utilizada por el **GenAnot** es utilizada por los demás módulos del entorno. Por lo tanto las mejoras introducidas son globales al entorno.

Otro mejora reside en la eliminación de la necesidad de copiar, unir e intersecar tablas de hash. Este proceso se realizaba cuando se procesaban sentencias **if** y **while**. En cambio con este algoritmo solo se necesita contar con referencia a las variables que se modifican en el ciclo y solo se realizan estos procesos para estas variables (no se realiza para todas). Esto se traduce: (1) en un menor tiempo de ejecución porque se eliminarón los procesos de copia, unión e intersección de tablas de hash; y (2) en un menor espacio de memoria utilizado, por la eliminación de la copia de la tabla

1. **nivel** es igual a cero y **variables** está vacío.
2. Una expresión e_1 *op* e_2 se dice que está acotada si todos sus operandos están acotados y su cota está definida por: $Max = Maximo(e_1.Min\ op\ e2.Min, e_1.Min\ op\ e2.Max, e_1.Max\ op\ e2.Min, e_1.Max\ op\ e2.Max)$
$Min = Minimo(e_1.Min\ op\ e2.Min, e_1.Min\ op\ e2.Max, e_1.Max\ op\ e2.Min, e_1.Max\ op\ e2.Max)$
3. Recorrer todo el árbol:
   (a) En cada acceso a arreglo **A[exp]**: Determinar $(e_1.Min, e_2.Max)$ cota de **exp**. Si $e_1.Min$ es mayor al límite inferior y $e_2.Max$ es menor al límite superior del arreglo **A**, entonces marcar el acceso al arreglo como seguro. Si la cota excede los límites del arreglo entondces se rechaza el programa por ser inseguro. Y si no se pudo determinar la cota del índice no se realiza ninguna acción.
   (b) En cada **return exp**, analizar **exp** buscando accesos seguros a arreglos (en caso de encontrar algún acceso a arreglo hacer 3a).
   (c) En cada asignación **x = exp**:
      i. Si **x** es un acceso a arreglo hacer 3a y analizar **exp** buscando accesos a arreglos.
      ii. En otro caso: Analizar **exp** para determinar su cota. // Si **nivel** es distinto de cero y **x** no está en **variables[nivel]**: insertar **x** en **variables[nivel]** e insertar un nuevo *contexto* en la pila de **x**. Actualizar la cota de **x** con la cota de **exp** (modificar la cota del tope de la pila de **x**.
   (d) En cada condicional **IF cond then else**:
      i. Analizar la expresión **cond** para determinar su cota y la expresión **cond** buscando accesos a arreglos (en caso de encontrar algún acceso a arreglo hacer 3a).
      ii. Incrementar **nivel** en uno. Procesar el cuerpo del **then**.
      iii. Incrementar **nivel** en uno. Procesar el cuerpo del **else**.
      iv. Por cada variable **x** que este en **variables[nivel]** o en **variables[nivel-1]**:
      Definir la cota del contexto actual de **x** como la unión de las cotas de los dos contextos anteriores (los contextos tope y subtope de **x**).
      v. Decrementar nivel en dos.
   (e) En cada iteración **WHILE cond cuerpo**:
      i. Analizar la expresión **cond** para determinar su cota y analizar la expresión **cond** buscando accesos seguros a arreglos (en caso de encontrar algún acceso a arreglo hacer 3a).
      ii. Incrementar **nivel** en uno. Procesar el cuerpo del ciclo, **cuerpo**:
         A. Determinar cotas invariantes del ciclo y determinar variables inductivas.
         B. Determinar cantidad de iteraciones del ciclo. Si se puede determinar entonces acotar variables inductivas y procesar **cuerpo** (saltar a 3).
         C. Por cada variable **x** que este en **variables[nivel]**:
         Definir la cota del contexto actual de **x** como la unión de las cotas de los dos contextos anteriores (los contextos tope y subtope de **x**).
      iii. Decrementar nivel en uno.

Figura 5: Algoritmo para Generar las Anotaciones

de hash. En la implementación original en ciertos puntos del programa se mantenían varias tablas de hash simultáneamente (en el caso de varios ciclos anidados).

4 Optimizaciones del Esquema de Prueba

Estas optimizaciones buscan reducir la cantidad de información enviada al consumidor como así también reducir el tiempo y la complejidad del proceso de verificación de la seguridad del código realizado por el consumidor de código.

4.1 Optimizar el Esquema de Prueba

El esquema de prueba generado por el prototipo original identifica las variables críticas (aquellas que intervienen como índices de arreglos y sus dependencias) y considera como punto crítico del programa todos aquellos puntos 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. Este camino de prueba es construido sobre el ASA.

Pero este esquema de prueba considera toda asignación de las variables críticas entonces el esquema hace referencia a asignaciones que no son relevantes en la definición del valor de los índices con que se accede a un arreglo. El algoritmo de la figura 6 presenta el nuevo algoritmo que genera el esquema de prueba. El mismo está basado en las *pilas de contextos* introducidas en la sección 3. El esquema de prueba construído solo incluye las asignaciones relevantes de las variables críticas.

El algoritmo de la figura 6 utiliza las variables auxiliares **nivel** y **variables**. La primera es de tipo entero y mantendrá información del nivel de bloques abiertos y la segunda es un vector de vectores de referencias a variables que mantendrá información de las variables que son modificadas en cada nivel abierto.

1. Al comenzar todos los nodos del ASA estan sin marcar. **nivel** es igual a cero. **variables** está vacío.
2. Recorrer todo el árbol:
   (a) En cada acceso a arreglo seguro **A[exp]**: Marcar el nodo. Analizar **exp**: marcar cada nodo del ASA referenciado por el tope de la pila de cada variable que esta en **exp**. Si el tope referencia a otros contextos entonces marcar los nodos que referencian estos.
   (b) En cada **return exp**, analizar **exp** buscando accesos seguros a arreglos.
   (c) En cada asignación **x = exp**:
      i. Si **x** es un acceso seguro a arreglo hacer 2a y analizar **exp** buscando accesos a arreglos.
      ii. En otro caso: Si **nivel** es distinto de cero y **x** no está en **variables[nivel]**: insertarla.
      Insertar un elemento en la pila de **x** que referencia al nodo de la asignación.
      Analizar **exp** construyendo una lista de referencias a nodos del ASA. Dichas referencias son aquellas que se encuentran en el tope de la pila de cada variable en **exp**. Esta lista se inserta en el tope de **x**. Con esto obtenemos todos los nodos de los cuales depende el valor de **x** en este punto del programa.
   (d) En cada condicional **IF cond then else**:
      i. analizar la expresión **cond** buscando accesos seguros a arreglos.
      ii. Incrementar **nivel** en uno. Analizar el cuerpo del **then**.
      iii. Incrementar **nivel** en uno. Analizar el cuerpo del **else**.
      iv. Por cada variable **x** que este en **variables[nivel]** o en **variables[nivel-1]**:
      Insertarle a **x** un elemento nuevo en su pila que referencie los dos contextos anteriores (lso elementos que eran tope y subtope). Luego eliminar **x** de **variables[nivel]** y de **variables[nivel-1]** según corresponda.
      v. Si se marco algún nodo dentro del cuerpo del **IF** entonces marcar el nodo.
      vi. Decrementar nivel en dos.
   (e) En cada iteración **WHILE cond cuerpo**:
      i. analizar la expresión **cond** buscando accesos seguros a arreglos.
      ii. Incrementar **nivel** en uno. Analizar el cuerpo del ciclo, **cuerpo**.
      iii. Por cada variable **x** que este en **variables[nivel]**: Insertarle a **x** un elemento nuevo en su pila que referencie el contexts anterior (el elemento que era tope). Luego eliminar **x** de **variables[nivel]**.
      iv. Si se marco algún nodo dentro del cuerpo del **WHILE** entonces marcar el nodo.
      v. Decrementar nivel en uno.
3. Recorrer todo el árbol armando el esquema de prueba con todos los nodos marcados .

Figura 6: Algoritmo para Generar el Esquema de Prueba

En la figura 7 se muestra un fragmento de un programa Mini en el cual se puede apreciar la diferencia entre el esquema de prueba generado por el algoritmo de la figura 6 y el algoritmo implementado originalmente. En el ejemplo vemos que la variable i es la única utilizada como índice para acceder a una posición del arreglo (línea 7); también podemos apreciar que el valor de i esta determinado por la asignación de la línea 6. Pero que i el valor de i en otros puntos del programa depende de las variables j y k.

El algoritmo original incluía en el esquema de prueba todos los puntos del programa en el cual se referenciaban variables que intervienen como índices de arreglos (y aquellas de las cuales dependen los índices). Por lo tanto, el esquema de prueba generado para el programa del ejemplo

incluye todos los puntos del programa.

El algoritmo de la figura 6, en cambio, identifica las asignaciónes críticas y construye el esquema de prueba a partir de estas. Es decir, el esquema de prueba generado para el mismo ejemplo solo incluye los puntos 6 y 7 del programa.

```
         ....
/*1*/    j = 10;
/*2*/    k = 1;
/*3*/    i = j - 10;
/*4*/    if (i<k){ i = k - 1; }
/*5*/    else    { i = j; }
/*6*/    i = 0;
/*7*/    a[i] = 100
         ....
```

Figura 7: Ejemplo de un programa Mini.

4.2 Remplazo de Ciclos de Iteración Indefinida por Ciclos de Iteración Fija

En el prototipo original, el consumidor de código, para verificar el esquema de prueba debe analizar el cuerpo de los ciclos (originalmente solo se permitian sentencias while) para identificar variables inductivas, y si es posible, calcular la cantidad de iteraciones que realiza el ciclo. Si se puede determinar la cantidad de veces que itera el ciclo, este puede ser reescrito como una sentencia for.

Para un ciclo que itera n veces, podemos redefinirlo como: for(int i=0; n; i=i+1), donde i es una variable nueva (que no existe en el programa, también se suele denominar "fresca"). int i=0 es la declaración de una variable nueva i inicializada en 0; n significa que itere hasta que i sea igual a n y i=i+1 define el incremento de i en cada iteración.

Con esto se evita el costo de tener que identificar las variables inductivas y calcular la cantidad de veces que itera un ciclo. Solo es necesario calcular las cotas de las variables que intervienen en el cuerpo del ciclo y que se encuentran en el esquema de prueba.

5 Optimización del Código Generado

Las optimizaciones de código son: **(1)** Propagación de constantes, **(2)** Eliminación de código muerto, **(3)** Eliminar asignaciones constantes en los ciclos, **(4)** Eliminación de expresiones redundantes.

Optimizando el código generado se obtiene:

1. Una representación más compacta del mismo. Esto trae aparejado una menor cantidad de código a ser procesado por el prototipo lo cual reduce el tiempo de procesamiento y requiere menor cantidad de memoria para el almacenamiento de las estructuras de datos. Además, una menor cantidad de código reduce el tamaño del envío al consumidor.

2. Un código más eficiente a ser ejecutado por el consumidor de código.

Las optimizaciones mencionadas son típicos procedimientos realizados por compiladores tradicionales y se encuentra una gran cantidad de bibliografía relacionada (consultar, por ejemplo, [1, 3]) por lo cual no se considera importante redundar en detalles.

6 Trabajos Relacionados

Todas las investigaciones en Proof-Carrying Code y Análisis Estático basado en flujo de control y de datos son de vital importancia para esté trabajo. Como así también técnicas de verificación y validación de software.

E. Albert et al. [2] introducen un enfoque con un esquema similar a PCC basado en interpretación abstracta [11]. El intérprete abstracto genera una tabla con información a partir del programa fuente. A partir de esta tabla y la política de seguridad (especificada con dominios semánticos) se genera la *condición de verificación*. Si esta se puede verificar entonces se envía al consumidor el código junto con la tabla. El consumidor solo debe generar la condición de verificación y verificarla. Pero queda por resolver la correspondencia del programa enviado con la tabla de información. A. Myers [13] afirma que interpretación abstracta es una poderosa metodología para diseñar análisis estáticos correctos por construcción.

J. Bergeron et al. [9] proponen decompilar binarios no confiables, para luego, construir el grafo de flujo de control y de datos (entre otros). Entonces realizan diversos análisis estáticos sobre estos grafos para verificar la seguridad de los binarios.

V. Haldar et al. [12] utilizan un ASA comprimido para codificar los programas seguros. Si bien mencionan que está codificación es segura por construcción, los resultados que obtuvieron solo les permite afirmarlo para verificar *escape analysis*.

Los compiladores certificantes son una alternativa prometedora para generar código anotado con la información necesaria para realizar diversas verificaciones estáticas. Estos fueron introducidos por G. Necula y P. Lee [14] que desarrollaron Touchstone. Java Bytecode Verification y Lenguajes Ensambladores Tipados son ejemplos de estos.

7 Conclusiones y Trabajos Futuros

Se obtuvo el diseño de un prototipo para PCC-SA más robusto y eficiente. Este entorno provee características 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 verifica el código estáticamente. Cabe resaltar que una de las características más importantes del prototipo 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.

Se está trabajando en la extensión del prototipo de modo que pueda ser usado para aplicaciones "reales". Si bien 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 y PCC. En esté momento se está extendiendo el lenguaje fuente y la política de seguridad, y se están incorporando nuevos análisis estáticos al entorno.

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 realize 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 [19] 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 constrastando los autómatas (que especifican deteminada propiedad) el flujo de control y de datos determinado por el ASA de cada programa.

Referencias

[1] A. Aho, R. Sethi, J. Ullman, "Compilers: Principles, Techniques and Tools". Adisson Wesley. 1988.

[2] E. Albert, G. Puebla, M. Hermenegildo, “An Abstract Interpretation -based Approach to Mobile Code Safety”. Electronic Notes in Theoretical Computer Science. COCV’04. 2004.

[3] A. Appel, “Modern Compiler Implementation in Java”. Cambridge University Press. 1999.

[4] A. Appel, A. Felty, “A Semantic Model of Types and Machine Instructions for Proof-Carrying Code”, in *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL’00), pp. 243–253, ACM Press, Boston, Massachusetts (USA), January 2000.

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

[6] A. Appel, E. Felten, “Models for Security Policies in Proof-Carrying Code”. Princeton University Computer Science Technical Report TR-636-01, March 2001.

[7] F. Bavera, M. Nordio, J. Aguirre, M. Arroyo, G. Baum, R. Medel, “CCMini: Un Prototipo de Compilador Certificante”. Reporte técnico del grupo Porcesadores de Lenguajes (no publicado). 2004.

[8] 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.

[9] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, N. Tawbi. “Static Detection of malicious Code in Executable Programs”. LSFM Research Group, Departamento de Informatica, Universidad de Laval, Canada. 2001.

[10] 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.

[11] P. Cousot, R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Program by Construction or Approximation of Fixpoint”. in *Proceedings ACM Symposium on Principles of Programming Language*. 1977.

[12] 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, March 2002.

[13] A. Myers, A. Sabefeld, “Language-Based Information-Flow security ”. IEEE Journal on Selected Areas in Communications, Vol. 21, No 1. January 2001.

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

[15] G. Necula, R. Schneck, “Proof-Carrying Code with Unstrusted Proof Rules”, in *Proceedings of the 2nd International Software Security Symposium*, November 2002.

[16] G. Necula, R. Schneck, “A Sound Framework for Untrusted Verification-Condition Generators”, in *Proceedings of IEEE Symposium on Logic in Computer Science* (LICS’03), July 2003.

[17] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum, “A Framework for Execution of Secure Mobile Code based on Static Analysis”. Aceptado para su publicación en los Proceddings de XXIV Conferencia Internacional de la Sociedad Chilena de Ciencia de la Computacin IEEE-CS Press, a realizarse en Tarapaca, Chile. Noviembre de 2004. Disponible en http://dc.exa.unrc.edu.ar/lenguajes/Files/pcc-sa_SCCC.ps

[18] M. Plesko, F. Pfenning, “A Formalization of the Proof-Carrying Code Architecture in a Linear Logical Framework”, in *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento (Italy), 1999.

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

[20] R. Schneck, G. Necula, “A Gradual Aproach to a More Trustworthy, yet Scalable, Proof-Carrying Code”, in *Proceedings of International Conference on Automated Deduction* (CADE’02), pp. 47–62, Copenhagen, July 2002.