PEDECIBA Informática
Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Compilación y Certificación de Código mediante Análisis Estático de Flujo de Control y de Datos

Francisco Bavera

Marzo de 2006

Orientador de Tesis: Jorge Aguirre
Supervisor: Alvaro Tasistro

Compilación y Certificación de Código
mediante Análisis Estático de Flujo de Control y de Datos

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT06-03
PEDECIBA
Instituto de Computación - Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Marzo de 2006

Resumen

En la última década el uso de las TIC ha irrumpido en forma explosiva en todas las áreas de actividad humana y han sido incorporadas a una, cada vez más, amplia gama de aplicaciones y artefactos. Paralelamente ha crecido de la misma forma el uso de Código Móvil; esto es, de código que es generado por un productor y usado por uno o numerosos consumidores. La distribución de software por Internet es el ejemplo más evidente, pero el uso de código móvil para la transmisión de actualizaciones o de nuevas versiones de código se extiende a la telefonía celular, a las tarjetas inteligentes y a todas las áreas donde pueda resultar importante la incorporación de nuevas funcionalidades a dispositivos controlados por software. La migración de código brinda una manera automática, de costo mínimo, para resolver la distribución o substitución de código para enormes cantidades de destinatarios. Sin embargo esta técnica también entraña graves riesgos, ya que el software migrado puede comprometer la seguridad de sus numerosos destinatarios, tanto por fallas de programación, como por maliciosas intenciones. La búsqueda de soluciones para este problema ha dado origen a activas líneas de investigación, dentro de las que se cuentan las referidas a la producción de Compiladores Certificantes, iniciadas con la introducción del Proof-Carrying Code (PCC) por G. Necula y P. Lee en 1996.

Un Compilador Certificante genera, además del código ejecutable, una prueba o demostración de que dicho código no puede violar la política de seguridad preestablecida para su consumidor. Al consumidor le basta entonces someter exitosamente al código recibido a su prueba, para poder asegurar que su ejecución le resultará segura. Algunas líneas de investigación sobre compilación certificante se basan en frameworks lógicos y utilizan demostradores de teoremas, otras en modelar la seguridad mediante un sistema de tipos. Las primeras tienen la dificultad de que las pruebas son de alta complejidad, generalmente exponenciales, y muchas veces necesitan asistencia humana; mientras que las últimas, aún, sólo pueden aplicarse a políticas seguridad sumamente restringidas.

Esta tesis se desarrolló dentro de un proyecto destinado a experimentar la construcción de compiladores certificantes basados en las técnicas de Análisis Estático de Flujo de Control y Datos, de difundido uso en la construcción de compiladores optimizantes. Se esperaba que este enfoque permitiera obtener soluciones lineales respecto de la longitud del programa fuente, tanto para la verificación de las pruebas obtenidas como para su generación. En dicho proyecto se definió un marco para el desarrollo de compiladores certificantes y sus correspondientes entornos de ejecución segura, al que se denominó *Proof-Carrying Code based-on Static Analysis* (PCC-SA).

La definición, prototipación y evaluación de PCC-SA se dividió en dos tesis de maestría del INCO, Universidad de la República, Uruguay. Ambas tesis compartieron el relevamiento del Estado del Arte, el diseño general, la definición del lenguaje fuente y la evaluación global del framework, mientras que una se ocupó específicamente del diseño y la prototipación de las componentes del productor de código y la otra de las componentes del consumidor.

La presente es la primera de dichas tesis (dirigida al entorno del productor). En ella se presenta a CCMini (*Certifying Compiler for Mini*), un compilador certificante para un subconjunto del lenguaje C, que garantiza que los programas que acepta no pueden leer variables no inicializadas y que en ellos no hay accesos a arreglos fuera de rango. Como lenguaje de código intermedio de CCMini se introducen los árboles sintácticos abstractos (ASA). Los ASA tiene una semántica clara que facilita la realización de los diversos análisis estáticos requeridos para generar la prueba de seguridad. Sobre el ASA del programa fuente, CCMini realiza el análisis estático, introduce anotaciones y verifica la seguridad del programa. Luego, si el programa es seguro, genera la información para el consumidor; información que consiste en el ASA anotado y en el esquema de la prueba. En los casos en que no se pueda determinar la seguridad de una computación especificada dentro del ASA (problema que no es decidible en general) CCMini anotará una indicación de verificación en tiempo de ejecución. Con esta información el receptor correrá la prueba y si esta es exitosa generará código objeto y podrá ejecutarlo en forma segura.

En la otra tesis del proyecto, que ya fue defendida, se mostraban varios casos de estudio que corroboraban la hipótesis de que tanto el proceso de generación de las pruebas como el de su

verificación tenían un comportamiento lineal respecto de la longitud de los programas fuente. En esta tesis se demuestra que en el peor caso las técnicas usadas tienen un comportamiento cuadrático.

Además se demuestra que la complejidad en casos reales (tanto del proceso de generación de la prueba, como el de verificación) es lineal respecto de la longitud del programa fuente. Esta demostración esta basada en la definición de una familia de programas C, a los que se llama linealmente acotables, y los cuales satisfacen algunas propiedades que se demostraron. Finalmente se inspecciona una conjunto de programas C de bibliotecas de uso universal, conjunto que comprende más de 4.000.000 de líneas de código y más de 90.000 funciones, convalidándose la hipótesis del comportamiento lineal de PCC-SA en la práctica ya que esta inspección permitió observar que estos programas pertenecen a la familia definida.

Palabras Clave: Compiladores Certificantes, Verificación de Programas, Seguridad, Código Móvil Seguro, Lenguajes de Programación, Proof-Carrying Code.

Agradecimientos

Primero, a quienes más quiero y amo:

A Yani por su amor y por compartir su vida conmigo.

A Guille y Benja por su capacidad infinita de amar y de hacer travesuras.

A mi papá porque para mí todavía es un superheroe y por sobre todo un super papá.

A mi mamá por ser como es.

A mis hermanos y a mis amigos porque siempre están. Gracias por su cariño, comprensión y por el tiempo compartido.

A Martín con quien no solo compartimos este trabajo sino también más de 10 años de estudios y mantenemos una gran amistad.

Al Profe no solo por ser mi director y un gran profesor sino porque es un ejemplo a seguir. Gracias por las oportunidades brindadas y las enseñanzas dadas.

A Nora, Tato y Carlos por sus consejos y el tiempo dedicado. Gracias por abrirme las puertas del InCo para realizar mis estudios.

A los revisores, en especial a Juanjo, por la atenta lectura de mi trabajo, las correciones, críticas y aportes a este trabajo.

A todos aquellos que me iluminaron con su aliento, enseñanzas y compania durante todo este trabajo: Pao, Naza, Chino, Mauricio, Marcelo ... (y a todos aquellos de los que me olvido en este momento).

Al InCo y al PEDECIBA por darme la oportunidad de realizar mis estudios.

Y por supuesto al DC mi lugar de trabajo.

A todos gracias totales!!!

Índice general

Índice de figuras

Capítulo 1

Introducción

Actualmente, la era de la información tiene su propia Caja de Pandora. En el cuantioso inventario de la caja, el código móvil es uno de los principales distribuidores de desgracias. 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 estos se destacan testing sistemático [36, 67, 78, 80], introducción de verificaciones dinámicas [18, 24, 75] y análisis estático [5, 25, 29, 33, 41, 48, 52, 68].

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 encuentra *Proof-Carrying Code* (PCC) [52].

Una alternativa prometedora para realizar la certificación de código consiste en utilizar *compiladores certificantes* [55]. La funcionalidad de un compilador certificante puede ser dividida en dos pasos. El primer paso consiste en compilar el código fuente a un lenguaje intermedio en el que se introducen anotaciones. Estas anotaciones guiarán la tarea de verificación de seguridad del paso siguiente e incluirán las condiciones de seguridad que debe satisfacer la ejecución del código en sus puntos críticos. El segundo paso demuestra que el código intermedio anotado cumple con las condiciones de seguridad impuestas y genera el esquema de la prueba que realizó. Este esquema de prueba consiste en los pasos realizados para verificar la seguridad. Si la verificación tuvo éxito, entonces es seguro ejecutar la aplicación final.

Existen ventajas adicionales a las concernientes a seguridad en la utilización de compiladores certificantes en lugar de compiladores tradicionales, ya que la mayor cantidad de información generada por un compilador certificante permite realizar distintas optimizaciones y proveer mayor información sobre el comportamiento del programa.

El concepto de compilación certificante surgió en los trabajos de G. Necula y P. Lee [55, 58] y se presenta como la alternativa más factible para lograr la automatización de la técnica de certificación de código. Los mencionados autores desarrollaron el compilador Touchstone, considerado como el primer compilador certificante. Posteriormente se produjeron grandes avances en la compilación certificante y como producto de estos avances surgieron los compiladores certificantes Special J [21], Cyclone [35, 39], TIL [74], FLINT/ML [71], Popcorn [48] y Splint [25]. También es considerado un compilador certificante el conocido compilador javac de Sun para Java, el cual produce Java bytecode. El propósito de javac es similar al de Special J, pero la salida del primero es mucho más simple que la del segundo porque el lenguaje de bytecode es mucho más abstracto que el generado por Special J. Por su parte, TIL y FLINT/ML son compiladores certificantes que mantienen información de tipos a través de la compilación, pero dicha información es eliminada luego de la generación de código. En cambio, los compiladores Popcorn y Cyclone son compiladores certificantes cuyo lenguaje de salida es el lenguaje ensamblador tipado TAL [48], por lo que incluyen información de tipos aún en el código objeto. 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.

La mayoría de la bibliografía sobre compiladores certificantes está basada en la introducción de sistemas de tipos que garanticen la seguridad buscada. No obstante hay aspectos relativos a la seguridad que no pueden ser representados por un sistema de tipos formal, particularmente garantizar que no se accede a variables no inicializadas ni se utilizan índices de arreglos no válidos. La posibilidad de violar esta última condición, ha permitido realizar famosas violaciones de seguridad recibiendo esta estrategia de violación el nombre de *buffer-overflow*.

Análisis estático de flujo de control y de datos [5, 15, 22, 25, 29, 33] 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*) [25].

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; estos riesgos, por lo general, conducen a problemas indecidibles. Por estas razones, el análisis estático debería ser parte del proceso de desarrollo de aplicaciones seguras [25] 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.

1.1. La Tesis

En esta tesis se presenta a CCMini (*Certifying Compiler for Mini*), un compilador certificante para un subconjunto del lenguaje C, que garantiza que los programas compilados no leen variables no inicializadas y que no acceden a posiciones no definidas sobre los arreglos. El proceso de verificación es realizado sobre árboles sintácticos abstractos (ASA) 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.

Utilizando análisis estático de flujo de control y de datos se puede brindar una alternativa para aquellos casos en los cuales 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 [29] argumentan que en *Proof-Carrying Code* [55] 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, en este trabajo, se utiliza un código intermedio de más alto nivel: un *árbol sintáctico abstracto* anotado con información de tipos. Esta variante tipada de ASA tiene una semántica clara y permite realizar diversos análisis estáticos para generar y verificar la información necesaria. Además, permite realizar optimizaciones de código.

Los compiladores certificantes antes mencionados, salvo Splint, utilizan lenguajes ensambladores tipados para expresar el código intermedio. En cambio, CCMini 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, CCMini realiza análisis estáticos sobre el código intermedio generado. CCMini 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.

Cabe resaltar que, en muchos casos, no siempre es posible determinar el estado de un punto dado en un programa utilizando análisis de flujo de control y de datos. 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, que en su caso general es equivalente al problema de la parada. Por eso se propone utilizar una técnica mixta, análisis estático de flujo de control y de datos e insertar verificaciones dinámicas (chequeos en tiempo de ejecución). Combinando estas técnicas se amplía el rango de programas seguros aceptados, incluyendo aquellos sobre los cuales el análisis estático no puede asegurar nada. 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).

El tamaño de las pruebas generadas por CCMini es lineal con respecto a la longitud de los programas. Esto es una ventaja importante sobre otros compiladores certificantes, tal como Touchstone, que generan pruebas hasta de tamaño exponencial con respecto al programa. Además, se

demostró que la complejidad de los algoritmos de CCMini que generan la certificación son lineales con respecto al tamaño de los programas. No sucede lo mismo en el caso de los compiladores 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, para realizar la verificación estática de la seguridad del código.

También se presenta aquí el diseño de las optimizaciones y extensiones de CCMini que permitirán aproximarse a una herramienta que pueda ser considerada como una opción atrayente para desarrollar aplicaciones seguras.

CCMini surgió como parte de un framework que garantiza la ejecución de código móvil de manera segura basado en las técnicas de *Proof-Carrying Code* (PCC) y análisis estático de control de flujo y de datos. Dicho framework, denominado *Proof-Carrying Code based-on Static Analysis* (PCC-SA) [62], fue diseñado como parte de estas actividades en el marco de un proyecto conjunto con Martín Nordio. En está tesis, también, se presenta PCC-SA y el diseño de la interfaz entre el productor y el consumidor.

PCC-SA pretende reducir el riesgo de seguridad que implica la interacción entre sistemas de software por medio de código móvil. El código móvil, como se mencionó anteriormente, 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.

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.

Además, tanto CCMini como PCC-SA, poseen 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.

1.2. El Proyecto Conjunto y el Trabajo Personal

Como se mencionó anteriormente parte del trabajo presentado se realizó dentro del marco de un proyecto conjunto con Martín Nordio. Este proyecto consistía en desarrollar un ambiente de PCC que garantizara la ejecución de manera segura de código móvil. El autor tuvo a su cargo el desarrollo del entorno del productor de código y Martín Nordio el del consumidor.

Como primera actividad se realizó una revisión bibliográfica de las técnicas denominadas *seguridad basada en el lenguaje fuente*, haciendo un fuerte hincapie en PCC. Se estudiaron sus lineamientos y variantes. Se encontraron varias líneas de investigación tendientes a obtener un sistema PCC eficiente, automático y escalable. Pero, se observó que ninguno se enfocaba a explotar las ventajas proporcionadas por las técnicas de análisis estático de flujo de control en conjunción con verificaciones dinámicas. Esta revisión fue una tarea importante dentro de las actividades desarrolladas (que motivó la publicación de un artículo [10]) en ella se basa el capítulo 2 (Código Móvil Seguro) y sus resultados se resumen en el apéndice A (Proof-Carrying Code).

Cuando se comenzó a diseñar el ambiente se planteó la necesidad de definir una política de seguridad interesante. Se llegó a la conclusión de que garantizar que todos los accesos a los arreglos sean válidos no solo es interesante desde el punto de vista de la seguridad sino que también es un problema complejo de resolver. Una vez determinadas las propiedades que se querían verificar (y teniendo en mente un ambiente PCC) se debía establecer el mecanismo para generar las anotaciones necesarias para poder demostrar la seguridad del código. Se decidió generar estas anotaciones

utilizando análisis estáticos de flujo de control y de datos. Para esto, resultaba natural utilizar como representación intermedia un *grafo de flujo de control* o un *árbol sintáctico abstracto*.

Llegado a este punto, se debía definir el método de verificación de las propiedades y acorde a este el de certificación. Según el enfoque ortodoxo de PCC, se debía definir una lógica y un sistema de tipos que nos permitiera verificar las propiedades sobre un lenguaje assembly tipado. En cambio, se planteó la posibilidad de explotar las posibilidades brindadas por las técnicas de análisis estático de flujo de control y de datos. Esto nos brindaría mayor expresividad y eficiencia (muchos de estos análisis tiene a lo sumo complejidad polinomial) con un menor esfuerzo de diseño. Además, en la revisión bibliográfica se observó que el énfasis se ponía en la utilización de técnicas complejas basadas en demostraciones lógicas y verificaciones de tipos en desmero del uso de verificaciones estáticas, lo cual tornaba interesante la exploración de este último camino.

Teniendo definido estos puntos se dividió el trabajo en dos desarrollos independientes. Por una parte, la correspondiente al productor de código y por la otra la correspondiente al consumidor. De estos trabajos se obtuvieron dos productos: un compilador certificante (CCMini) y un verificador de código foráneo, los cuales componen el framework de PCC-SA pero pueden ser utilizados independientemente.

Los capítulos 2, 3 y el apéndice A se desarrollaron en conjunto con Martín Nordio. Los capítulos 1, 4, 5, 6 y el apéndice B son desarrollos propios de esta tesis.

1.3. Estructura de la Tesis

Esta tesis esta estructurada de la siguiente manera: primero, en el capítulo 2 se presenta una introducción a la problemática del código móvil y se analizan las principales técnicas para garantizar seguridad basándose en el lenguaje fuente. Luego, en la sección2.2 se presenta el framework de PCC-SA desarrollado. En el capítulo 3, se especifica informalmente el prototipo de PCC-SA haciendo hincapie en la interfaz entre el productor-consumidor. Estableciendo las responsabilidades, la política de seguridad, la representación intermedia a utilizar y los análisis a realizar para certificar y verificar las propiedades deseadas. El compilador certificante CCMini se presenta en el capítulo 4. Se describen los módulos del prototipo implementado y se analizan los resultados obtenidos. En el capítulo 5 se plantean una serie de extensiones y optimizaciones de CCMini. Estas actividades pretenden ser una primer instancia para lograr un compilador certificante que pueda ser usado en el "mundo real". Por último, en el capítulo 6, se encuentran las conclusiones de este trabajo y los trabajos futuros que se planean realizar.

El apéndice A contiene una breve introducción a PCC y la descripción de las principales líneas de trabajo en este tema. El apéndice B describe aquellos trabajos que se consideran más estrechamente vinculados a esta tesis.

Capítulo 2

Código Móvil Seguro y PCC-SA

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.

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?

Los problemas de seguridad que presentan las técnicas de código móvil no confiables suponen riesgos costosos para los sistemas. Los efectos no deseados producidos por código móvil malicioso pueden clasificarse en tres clases:

- Modificaciones en los sistemas: puede afirmarse que cualquier código móvil que modifique el entorno de computación realiza modificaciones en el sistema. Estas modificaciones pueden ser tan sencillas como modificar un archivo o tan complejas como instalar un virus o un gusano en el sistema operativo del host; o alterar la configuración del sistema y/o de seguridad.
- Invasión de la privacidad: además de destruir o modificar información, el código móvil puede exportar cualquier tipo de información de los sistemas afectados hacia varios destinos de la red, en los que puede ser utilizada. La información exportada puede ser desde archivos hasta estadísticas y/o contraseñas.
- Rechazo de servicio: el rechazo de servicio consiste en realizar una serie de ataques que impiden que el host realice su trabajo habitual. El código remoto malicioso intentará reservarse toda la memoria disponible o crear miles de ventanas para impedir que los demás programas se ejecuten correctamente. En otros casos, puede intentar cerrar la conexión de Internet. La mayoría de los *applets* de Java con carácter malicioso implementan ataques para conseguir rechazos de servicio.

Estas amenazas existen con anterioridad al surgimiento de la popularidad del código móvil en general y los *applets* Java en particular. Ya que cuando un usuario descarga un programa de una página web o inserta un diskette en su equipo y lo ejecuta, el programa puede acceder a recursos críticos y, por lo tanto, se presentan las amenazas antes mencionadas.

Sin embargo, vale la pena estudiar estas amenazas ya que con la difusión de uso de código móvil resulta notable la gran escala en la que aparecen. Las diversas aplicaciones de código móvil

prometen constituir un inmenso número de programas móviles que ejecutarán grandes cantidades de tareas en una amplia variedad de tipos de consumidores de código (o *hosts*); por otro lado, habrán sido creados por una amplia diversidad de programadores, particularmente con diversos niveles de competencia y honestidad. De esta manera desaparece la apariencia de seguridad, algunas veces falsa, proporcionada por los paquetes de software o los sitios web "de confianza". Dado que los consumidores recibirán programas de fuentes distintas y desconocidas, ellos deberán poseer un mecanismo que les permita decidir si confiar o no en lo que reciben.

2.1. Enfoques para Tratar los Problemas del Código Móvil

Si no se adopta alguna medida de seguridad, el código no confiable es un peligro potencial. Por lo tanto, algunos mecanismos de seguridad son necesarios para poner los recursos del consumidor a salvo del código invasivo que pudiera ser ejecutado en él. Las técnicas propuestas para resolver este problema pueden clasificarse en tres:

- Autoría Personal,
- Verificación en Tiempo de Ejecución,
- Seguridad basada en el Lenguaje Fuente.

2.1.1. Autoría Personal

No se hace ningún intento para fortalecer la seguridad; el código móvil es firmado por una persona o compañía para afirmar que es seguro. El esquema de la firma digital puede usarse para autenticar el componente firmado. Este garantiza que el código ha sido escrito por un productor conocido y confiable pero, no garantiza que esté libre de errores o sea malicioso. Este esquema sólo garantiza la identidad del productor de código.

En este punto, se puede considerar también el uso de criptografía como medio de autenticación del productor de código, protegiendo así el envío de código por un medio inseguro. Pero, no hay que dejar de considerar que los protocolos criptográficos pueden ser descifrados por posibles interceptores con suficiente poder de cómputo.

Para asegurar confiabilidad no basta con la autoría personal ni criptografía por sí solos. Es necesario considerar enfoques que tengan en cuenta propiedades intrínsecas del código, independientemente de quién fue el productor o de cómo fue producido. Sin embargo, la autoría personal podría ser un componente importante de una política de seguridad para la ejecución de código móvil y, en tales casos, puede ser usada en combinación con otras técnicas.

2.1.2. Verificación en Tiempo de Ejecución

Componentes no confiables son monitoreados en tiempo de ejecución para asegurar que sus interacciones con otros componentes están estrictamente limitadas. Técnicas típicas incluyen *Kernel as Reference Monitor* y *Code Instrumentation* [42]. El costo de obtener seguridad con estos métodos imponen serias pérdidas en la performance. Además, a menudo, existe una gran brecha semántica entre las propiedades de bajo nivel que son garantizadas por el chequeo (por ejemplo, *address space isolation*) y las propiedades de alto nivel que son requeridas (por ejemplo, *black box abstraction* [43]).

Kernel as Reference Monitor es, quizás, el primer mecanismo de seguridad usado en sistemas de software. El *Kernel* es un programa con privilegios que puede acceder a los recursos críticos y a los datos directamente. Todo otro proceso debe acceder por intermedio del *Kernel* (usándolo como un *proxi*). Esto no solo previene que el código malicioso dañe al sistema, sino también permite al *Kernel* monitorear los accesos al sistema y realizar autenticaciones.

Code Instrumentation consiste en modificar el código insertando verificaciones dinámicas en el código objeto (o en su defecto en el código de máquina) para que operaciones críticas sean monitoreadas durante la ejecución. Un ejemplo de esta técnica es *Software Fault Isolation* (o *sandboxing*)

[75]. Básicamente, el proceso es implementado de tal manera que: (1) las verificaciones insertadas no deben alterar la funcionalidad del código original y deben probar que no viola las restricciones de seguridad; (2) si el código original viola las restricciones de seguridad en algún punto del programa entonces alguna verificación insertada lo debe detectar y debe retornar el control al sistema.

Una desventaja de esta técnica es que requiere hardware especial y un sistema operativo relativamente complejo que lo soporte, características que podrían no estar disponibles en ambientes más "pobres" como *smart cards* o sistemas integrados.

2.1.3. Seguridad basada en el Lenguaje Fuente

Seguridad basada en el Lenguaje Fuente (*language-based security*) consiste en un conjunto de técnicas basadas en la teoría de los lenguajes de programación y en su implementación, esto incluye semántica, tipos, optimización y verificación todo centrado en aspectos de seguridad [73].

Los compiladores de lenguajes de alto nivel generan mucha información durante el proceso de compilación: información de tipos, restricciones de los valores, información sobre la estructura y el flujo de los datos. Esta información se obtiene durante el *parsing* y/o realizando distintos análisis del programa. Ella puede ser utilizada para realizar optimizaciones o verificaciones de corrección. Pero, los compiladores tradicionales, descartan dicha información despúes de la compilación.

Sin embargo, esta información puede ser importante si se desean realizar posteriormente verificaciones sobre propiedades del código compilado. Por ejemplo, programas escritos en lenguajes tipados seguros deben cumplir con las restricciones de tipos impuestas cuando son compilados. Y, asumiendo que el compilador es correcto, el código compilado debería cumplir con las mismas restricciones de tipos. Si el ambiente o entorno de ejecución del consumidor de código tiene acceso a la información extra generada por el compilador (cuando compiló el programa) puede verificar con menor esfuerzo si el código es seguro de ejecutar [42].

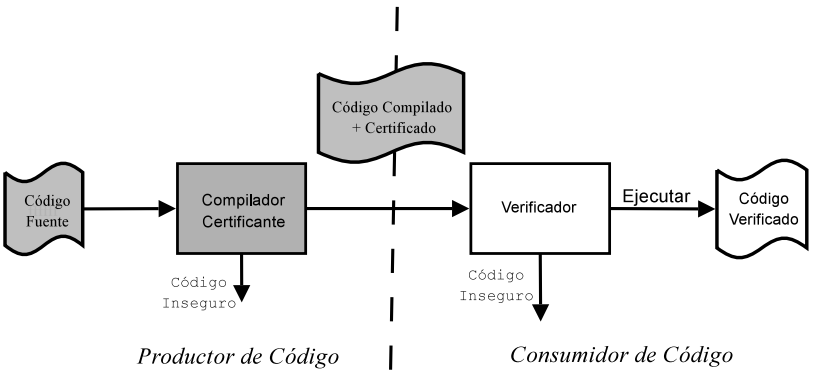

Figura 2.1: Vista Simplificada de Seguridad basada en Lenguajes.

La idea 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 [42].

En los próximos apartados se presentan las principales técnicas que siguen estos lineamientos: *Proof-Carrying Code* (PCC) [7, 17, 21, 52, 54, 55], *Type Assembly Language* (TAL) [47, 48, 72, 76], *Efficient Code Certification* (ECC) [41, 42] y *Information Flow* (JFlow) [51, 68]. En la siguiente

sección (sección 2.2) se presenta el framework propuesto basado en PCC y análisis estático de flujo de control y de datos.

Java

El lenguaje de programación Java es, quizás, la primer aplicación de éste enfoque que busca garantizar seguridad basándose en el lenguaje fuente. El compilador Java produce código independiente de la plataforma (*bytecode*) que puede ser verificado por el consumidor antes de su ejecución. El verificador de *bytecode* [28, 45] garantiza propiedades de seguridad básicas de memoria, flujo de control y tipos. La verificación de *bytecode* es ejecutado sobre el código de cada método no abstracto de cada clase. Esto consiste en una ejecución abstracta del código del método, ejecutado a nivel de tipos en lugar de valores como ocurre durante una ejecución normal.

Si el *bytecode* pasa el proceso de verificación entonces es interpretado por la máquina virtual (VM) de Java o puede ser compilado a código objeto por un compilador *just-in-time* (JIT). Este último paso es la principal desventaja de este enfoque debido a que causa un aumento en el tiempo de ejecución del código.

Proof-Carrying Code

La técnica *Proof-Carrying Code* (PCC) [55] exige a un productor de software proveer su programa conjuntamente con una prueba formal de su seguridad. La política de seguridad del destinatario se formaliza mediante un sistema de axiomas y reglas de inferencia, sobre el cual debe basarse la demostración construida por el productor. El consumidor, por su parte, verifica que la prueba sea válida y que el código recibido corresponda a la demostración, y sólo ejecuta el código en caso de que ambas respuestas sean positivas.

PCC no requiere autenticación del productor, ya que el programa sólo correrá si localmente se ha demostrado su seguridad, minimizando así la cantidad de confiabilidad externa requerida. Tampoco se requiere verificación dinámica, con lo cual no se introduce ninguna merma en el tiempo de ejecución, ya que la prueba se efectúa con antelación. Este enfoque estático de PCC es una ventaja respecto a los enfoques dinámicos que controlan la seguridad operación a operación mientras el código se ejecuta, tales como el implementado por Java.

El proceso de creación y uso de PCC está centrado alrededor de la política de seguridad, la cual es definida y hecha pública por el consumidor de código. A través de esta política, el consumidor de código especifica precisamente bajo qué condiciones considera que es segura la ejecución de un programa remoto.

La principal ventaja de PCC está en su expresividad y en su capacidad de permitir optimizaciones en el código producido. En principio, cualquier política de seguridad que pueda ser expresada en una lógica podría ser considerada. La principal desventaja reside en la complejidad de construir la prueba de seguridad. Además, y no menos importante, el tamaño de las pruebas generadas es demasiado grande con respecto al tamaño de los programas. Esto permite afirmar que el uso de PCC puede ser apropiado para aplicaciones que requieren código seguro y optimizado, pero que será ejecutado gran cantidad de veces (por ejemplo, un *kernel* extensible). Pero, su uso es poco atractivo para aplicaciones que serán ejecutadas una sola vez (por ejemplo, *applets*) [42].

Un análisis más detallado de este enfoque y las variantes que han sido desarrolladas son presentadas en el apéndice A.

Type Assembly Language

Otro conjunto interesante de trabajos son aquellos relativos a *Lenguajes Assembly Tipados* (TAL). Estos lenguajes proveen una sólida base para obtener verificadores de propiedades totalmente automáticos. Además, los sistemas de tipos brindan el soporte para forzar a utilizar abstracciones en lenguajes de alto nivel, entonces un programa bien tipado no puede violar estas abstracciones. La información de tipos del lenguaje de alto nivel es mantenida durante todo el proceso de compilación e insertada en el lenguaje *assembly* generado. El resultado consiste en a-

notaciones de tipos en el código objeto, que pueden ser chequeadas por un simple verificador de tipos.

Si se considera que las anotaciones de tipos son esencialmente pruebas de seguridad de tipos y que el verificador de tipos es un verificador de pruebas entonces TAL puede ser visto como una variante de PCC.

La técnica basada en TAL no es más expresiva que PCC, pero permite garantizar cualquier propiedad de seguridad que pueda ser expresada en términos de sistemas de tipos. Esto incluye memoria, flujo de control, seguridad de tipos y algunas otras propiedades.

TALx86 es un lenguaje assembly tipado para INTEL x86, desarrollado por Morriset et al. [48], que ha tenido una amplia difusión dentro de las técnicas basadas en TAL. TALx86 comenzó con funciones de alto orden y polimorfismo, considerando a System F como un lenguaje de alto nivel. De este lenguaje surgieron algunas extensiones como STAL [72] (el cual modela el *stack* con polimorfismo), DTAL [76] (usa tipos dependientes, permitiendo entre otras cosas hacer algunas optimizaciones en la verificación de accesos a arreglos) y Alias TAL [47] (permite reusar áreas de memoria , cosa que TALx86 prohibe, permitiendo *aliasing*).

Efficient Code Certification

Efficient Code Certification (ECC) [41, 42] es un enfoque simple y eficiente para la certificación de código compilado proveniente de una fuente no confiable. Este enfoque puede garantizar propiedades básicas, pero no triviales, de seguridad incluyendo seguridad de flujo de control, seguridad de memoria y seguridad de stack. ECC es menos expresivo que PCC y TAL pero sus certificados son relativamente más compactos y fáciles de producir y verificar. Tanto la generación del certificado como su verificación son automáticas e invisibles al productor y al consumidor.

ECC fue concebido para ser utilizado en aquellos casos en el cual el proceso de verificación no puede ser ser amortizado por la cantidad de veces que se ejecutará el código. Por ejemplo, si se deseara verificar la seguridad de un *applet* de una página *web*. Por lo cual, es necesario que los certificados sean concisos y fáciles de verificar [42].

Aunque está inspirado en PCC es inadecuado llamar pruebas a los certificados. Estos no son pruebas en un sistema formal. El certificado consiste de anotaciones que proveen información de la estructura y funcionalidad del código incluyendo información básica de tipos. Esta información se obtiene a partir del programa en tiempo de compilación. El verificador utiliza esta información para chequear estáticamente un conjunto de condiciones simples que implican inductivamente las propiedades de seguridad deseadas.

Los certificados ECC tienen un tamaño menor que sus equivalentes en PCC y TAL. La causa reside en el uso, por parte de ECC, de convenciones de compilación. Esto tiene la ventaja de evitar información que debe ser incluida explícitamente en los certificados PCC y TAL. Por ejemplo, si las funciones retornan siempre el resultado en un mismo registro no es necesario explicitarlo en el certificado. Pero tiene la desventaja de hacer que el verificador dependa fuertemente del compilador.

Information Flow

Control estático de flujo de información es un nuevo enfoque, muy prometedor, para proteger confidencialidad e integridad de la información manipulada por programas. El compilador traza una correspondencia entre los datos y la política de flujo de la información que restringe su uso [51, 68]. Esto es similar a las formas de seguridad antes expuestas excepto que la política de seguridad está basada en un modelo de flujo de información.

El usuario introduce anotaciones en el código fuente de alto nivel que especifican como puede ser el flujo de información en un programa y entre programas. Estos programas anotados son verificados en tiempo de compilación para asegurar que cumplen con las reglas de flujo.

JFlow [51] es un prototipo que extiende Java con primitivas de flujo de información. El compilador JFLow verifica la información de flujo y luego la descarta y genera código *bytecode* estándar. Si la información de flujo fuera adjuntada al código generado entonces se podría verificar su seguridad antes de ejecutarlo siguiendo un esquema similar al de TAL o ECC [42].

2.2. El Framework Propuesto

El entorno de ejecución de código móvil propuesto, denominado *Proof-Carrying Code based on Static Analysis* (PCC-SA), está centrado en poder brindar una solución en aquellos casos en los cuales la política de seguridad no puede ser verificada eficientemente por un sistema de tipos formal, como es el caso de verificar inicialización de variables y accesos válidos a arreglos. Esta falta de eficiencia se refiere tanto a nivel de razonamiento sobre propiedades del código como a nivel de performance. V. Haldar, C Stork y M. Franz [29] 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, por ejemplo un *grafo de flujo de control* o un *árbol sintáctico abstracto* anotados con información de tipos. Además, PCC-SA utiliza la técnica de análisis estático para generar y verificar esta información. La representación intermedia utilizada permite realizar diversos análisis estáticos para generar y verificar la información necesaria y para realizar optimizaciones al código. La principal ventaja de esta técnica es que el tamaño de las pruebas generadas es lineal respecto al tamaño de los programas además de que es posible aplicar varias técnicas de optimización de código sobre la representación intermedia.

El análisis estático de flujo de control y de datos es una técnica muy conocida en la implementación de compiladores que en los últimos años ha despertado interés en distintas áreas tales como verificación y re-ingeniería de software. La verificación de propiedades de seguridad puede ser realizada por medio de análisis estático debido a que este permite obtener una aproximación concreta del comportamiento dinámico de un programa antes de ser ejecutado. Existe una gran cantidad de técnicas de análisis estático que pueden ser usados para realizar verificaciones de seguridad (por ejemplo, *array-bound checking* o *escape analysis*) [25].

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

Es conveniente aclarar que si bien el análisis estático es una técnica importante para garantizar seguridad, no soluciona todos los problemas asociados con seguridad. En muchos casos no puede reemplazar controles en tiempo de ejecución, testeo sistemático o verificación formal. Sin embargo, no se avizora ninguna técnica que pueda eliminar todos los riesgos de seguridad. Con lo cual, el análisis estático debería ser parte del proceso de desarrollo de aplicaciones seguras [25] y combinarse con otras técnicas, por ejemplo, con PCC tradicional. PCC-SA, en cambio, combina los lineamientos de PCC y compiladores certificantes utilizando análisis estático de flujo de control y de datos.

El framework PCC-SA permite insertar verificaciones dinámicas (chequeos en tiempo de ejecución) a fin de ampliar el rango de los programas seguros aceptados, incluyendo aquellos sobre los cuales el análisis estático no puede garantizar la seguridad del código. Por ejemplo cuando un análisis estático no puede acotar un ciclo, se utilizan chequeos en tiempo de ejecución para garantizar la seguridad. Es decir, el mecanismo usado para verificar que el código es seguro puede ser una combinación de verificaciones estáticas (en tiempo de compilación) y de verificaciones dinámicas (en tiempo de ejecución). Cabe resaltar que, en muchos casos, es necesario insertar estos chequeos en tiempo de ejecución no sólo por las limitaciones de un análisis estático particular sino porque los problemas a resolver son no computables. Por ejemplo, el problema de garantizar la seguridad al eliminar la totalidad de *array-bound checking* no es computable, este problema en su caso general es equivalente al problema de la parada.

La información necesaria para garantizar distintos aspectos de seguridad puede ser generada mediante análisis estático de flujo de control y de datos de los programas [25, 29]. Combinando el esquema PCC y el análisis estático se pueden aprovechar los beneficios de ambos y proveer un sistema seguro y más eficiente. Utilizando estas dos técnicas se define la arquitectura del entorno de ejecución de código móvil seguro PCC-SA.

2.2.1. Descripción del Framework

La figura 2.2 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.

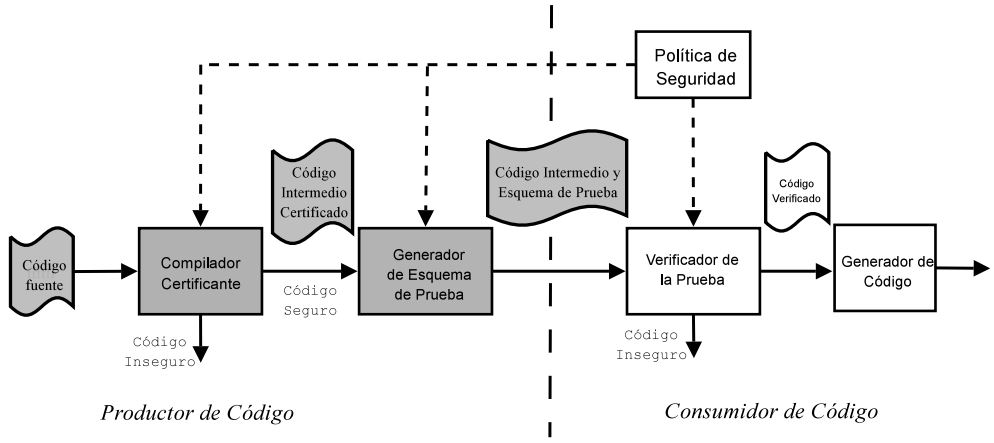

Figura 2.2: Vista Global de PCC-SA.

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.

2.2.2. Ventajas y Desventajas de PCC-SA

Proof-Carrying Code based on Static Analysis preserva las siguientes ventajas, ya presentes en *Proof-Carrying Code*:

1. La infraestructura del consumidor es automática y de bajo riesgo. Por lo cual, la base confiable requerida para el consumidor es pequeña.

2. El esfuerzo reside en el productor: el consumidor de código solamente tiene que ejecutar un proceso de verificación de pruebas rápido y simple.

3. El consumidor no debe confiar en el productor.

4. Es flexible, dado que puede utilizarse con un amplio rango de lenguajes y de políticas de seguridad.

5. No sólo se puede usar para seguridad.

6. La generación de código puede ser automatizada.

7. El código se verifica estáticamente.

8. Permite detectar cualquier modificación (accidental o maliciosa) del programa y/o su prueba, mientras que sigue garantizando la seguridad.

9. Puede usarse en combinación con otras técnicas.

Debido a que PCC-SA está basado en análisis estático, se pueden avizorar las siguientes características propias a la combinación de las técnicas antes mencionadas:

1. El tamaño de las pruebas es lineal al programa producido. En la mayoría de casos, el tamaño de las pruebas es menor al tamaño de los programas.

2. Permite ampliar el rango de las propiedades de seguridad que pueden ser demostradas automáticamente.

3. Permite eliminar una mayor cantidad de chequeos en tiempo de ejecución que en PCC.

4. Permite realizar diversas optimizaciones al código generado.

5. Manipular una representación equivalente al código fuente permite independencia de plataforma.

La idea de PCC-SA es fácil de comprender, pero su implementación eficiente presenta serias dificultades y su uso algunas desventajas. A continuación se enumeran las principales desventajas del uso de PCC-SA, algunas propias de este entorno y otras heredadas de PCC:

1. Al igual que 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.

5. Traducir una propiedad de seguridad al análisis estático que la verificará no es trivial.

2.3. Discusión

La *seguridad basada en lenguajes* es un área de investigación con un gran contenido intelectual y un campo de aplicación muy significante pero aún quedan muchos problemas por resolver y preguntas por responder. Se puede apreciar que hay una gran comunidad que esta investigando y desarrollando alternativas dentro de esta línea.

Los enfoques presentados (PCC, TAL, ECC, JFlow y PCC-SA) difieren en expresividad, flexibilidad y eficiencia pero todos tienen un punto común: usar la información generada durante el proceso de compilación para que el consumidor pueda verificar la seguridad del código eficientemente.

El certificado involucrado en seguridad basada en lenguajes puede tomar 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.

La idea de usar lenguajes y compiladores para garantizar seguridad no es nueva. El sistema Burroughs B-5000 requería aplicaciones escritas en un lenguaje de alto nivel (Algol). El sistema Berkeley SDS-940 usaba reescritura de código. El sistema operativo SPIN [18] utiliza tecnología basada en lenguajes para protejer el sistema base de un conjunto de ataques cuando se desea extenderlo.

Lo nuevo en *seguridad basada en lenguajes* es el uso de mecanismos que trabajan con lenguajes de bajo y alto nivel y que se pueden aplicar a un gran clase de propiedades de seguridad. Además de la integración de diversas técnicas para obtener sistemas más potentes y eficientes.

El uso de sistemas de tipos, compiladores certificantes, verificadores estáticos y monitores en tiempo de ejecución son enfoques prometedores para garantizar seguridad en los sistemas. Estos permiten contar con una base confiable pequeña. Pero, la clave de la técnica *seguridad basada en lenguajes* está en combinar compiladores certificantes, sistemas de tipos y monitores en tiempo de ejecución conjuntamente con sistemas que verifiquen estáticamente la información generada.

Recientemente se están desarrollando trabajos que tienden a usar *code instrumentation* en conjunción con *seguridad basada en lenguajes* para mejorar la performance [24, 26, 79] (en el apéndice B se dará más información de estos trabajos). La ventaja de *code instrumentation* consiste en que no requiere información extra acerca del código para garantizar seguridad pero tiene un incremento en el tiempo de ejecución debido a las verificaciones dinámicas introducidas. Muchas de estas verificaciones pueden ser eliminadas efectuando análisis que determinen si son verificaciones redundantes. Pero, la mayoría de estos análisis son necesariamente incompletos porque generalmente la satisfacción de propiedades de seguridad es generalmente indecidible.

Capítulo 3

Especificación del Prototipo de PCC-SA

En este capítulo se presenta la especificación informal del prototipo de PCC-SA. Se hace especial hincapie en la interface entre el productor de código y el consumidor de código. Primero, en la sección 3.1, se presenta el lenguaje fuente Mini, el lenguaje intermedio (sección 3.2) y la política de seguridad (sección 3.3). Luego, se presenta el diseño los análisis estáticos para verificar las propiedades de seguridad especificadas por la política de seguridad (sección 3.4).

3.1. El lenguaje Mini

En esta sección se define la estructura y el significado de un programa Mini. Este lenguaje fuente tiene una sintaxis similar al lenguaje C. Debido a que se busca analizar el comportamiento de los accesos a arreglos en los programas, se definió un lenguaje de programación simple cuya característica principal es la manipulación de arreglos y cuyos programas pueden analizarse sin preocuparse sobre los detalles irrelevantes (para el problema a analizar) de un lenguaje completo.

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 usados (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 estáticas de datos más complejas, tales como matrices.

Mini permite 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 $0 \leq \texttt{a} \leq 10$. Con esta extensión se pretende incrementar la cantidad de variables y expresiones acotadas que intervienen en los programas. Es decir, permite generalizar el proceso a funciones que reciben datos acotados por el programa invocante.

El lenguaje incluye sentencias condicionales `if` y `if-else`, la sentencia iterativa (`while`), y la sentencia `return`. A diferencia de C, la asignación es una sentencia y no una expresión. Esta decisión de diseño fue tomada para simplificar la semántica de Mini. Las sentencias se separan con `;`, al igual que la declaración de variables. Los bloques de sentencias son delimitados por { y }. En cada bloque, al inicio, pueden ser declaradas e inicializadas variables. Las reglas de visibilidad y tiempo de vida de las variables son las mismas que en C.

Los nombres de los identificadores siguen las reglas usuales, como así también los literales que denotan constantes numéricas y lógicas (`true` y `false`). Entre los operadores se encuentran los aritméticos (+, −, ∗, and /), lógicos (|| and &&) y relacionales (==, >, <, >=, <=, and ! =) con su significado y reglas de precedencia habituales.

Los comentarios están delimitados por /∗ y ∗/, pudiendo estar anidados.

3.1.1. Gramática del Lenguaje Mini

En la figura 3.1 se muestra la sintaxis del lenguaje Mini utilizando una gramática (usando la notación BNF [4]). Utilizando una gramática se describe de manera clara y sin ambigüedades la estructura de un programa Mini.

Program	::=	BasicType **Ident** **(** ParamDecl **)** Compound
TypeSpecifier	::=	BasicType | StructType
BasicType	::=	**int** | **boolean**
StructType	::=	BasicType [**intdenot**]
ParamDecl	::=	BasicType **Ident** | **int Ident (intdenot, intdenot)**
		| BasicType **Ident ,** ParamDecl
		| **int Ident (intdenot, intdenot) ,** ParamDecl
Compound	::=	**{ }** | **{** Statement **}** | **{** VariableDecl Statement **}**
VariableDecl	::=	VariableDecl VariableDecl | TypeSpecifier ListaIdent ;
ListaIdent	::=	**Ident** | **Ident =** Exp | ListaIdent , ListaIdent
Statement	::=	Statement Statement | Compound | **if (** BExp **)** Statement
		| **if (** BExp **)** Statement **else** Statement | **while (** BExp **)** Statement
		| Assign ; | **return** Exp ; | ;
Assign	::=	**Ident =** Exp | **Ident** [IExp] **=** Exp
Exp	::=	IExp | BExp
IExp	::=	**Ident** | **intdenot** | **Ident** [IExp]
		| IExp **op** IExp | **(** IExp **)**
BExp	::=	**Ident** | **true** | **false** | BExp **bop** BExp
		| **not** BExp | Exp **rop** Exp | **(** BExp **)**

Donde **Ident** son los Identificadores; **intdenot** son los números enteros; **op** son los operadores +, -, * y /; **bop** son los operadores || y &&, y **rop** los operadores $==, >, <, >=, <=, y\ !=$.

Figura 3.1: Gramática del Lenguaje Mini.

3.2. El Código Intermedio: Árbol Sintáctico Abstracto

El código intermedio que genera el compilador certificante es un árbol sintáctico abstracto (ASA). Éste es una representación abstracta del código fuente (descripto en la sección 3.1) 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.

La estructura de los árboles sintácticos abstractos utilizados es similar a la de los ASAs tradicionales. Los ASAs utilizados extienden a los ASAs tradicionales ya que permiten incluir anotaciones del estado del programa. Estas anotaciones reflejan el estado de los objetos que intervienen en el programa, conteniendo, por ejemplo, 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. 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.

En la figura 3.2 se presenta el ASA para el siguiente ejemplo:

```
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;
}
```

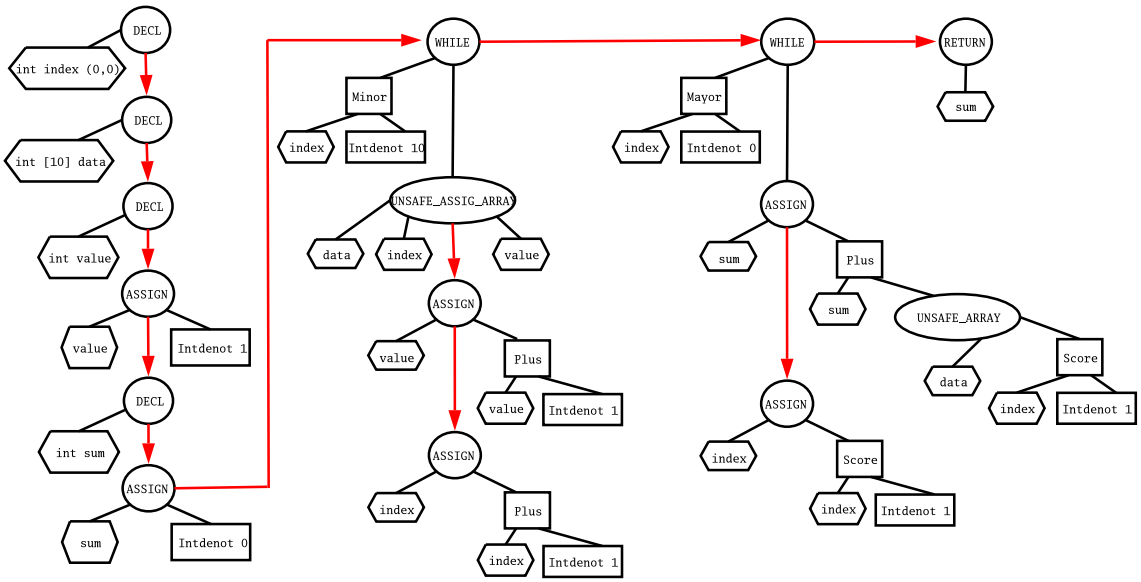

Figura 3.2: ASA del Programa.

En la figura 3.2 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 WHILE y RETURN. Por ejemplo, podemos observar que el primer ciclo esta constituido 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 $value + 1$.

3.3. La Política de Seguridad Abordada

Una política de seguridad es un conjunto de reglas que definen si un programa se considera seguro de ejecutar o no. Si consideramos a un programa como la codificación de un conjunto de posibles ejecuciones, se dice que un programa satisface la política de seguridad si el predicado de seguridad se mantiene verdadero para todo el conjunto de posibles ejecuciones del programa [73].

La política de seguridad que se eligió garantiza seguridad de tipos, además de garantizar que no se leen variables no inicializadas y que no se accede a posiciones fuera de rango sobre los arreglos. La posibilidad de violar esta última condición, ha permitido realizar famosas violaciones de seguridad recibiendo esta estrategia de violación el nombre de *buffer-overflow*. En la siguiente subsección se presenta una formalización de dicha política de seguridad.

3.3.1. Formalización de la Política de Seguridad

La política de seguridad se formalizó por medio de un sistema de tipos a fin de evitar toda ambigüedad y definirla claramente. Existen varias ventajas en utilizar un sistema de tipos para formalizar la política de seguridad. Primero, los sistemas de tipos son flexibles y fáciles de configurar. Una vez que los tipos son seleccionados, muchas propiedades interesantes de seguridad pueden ser obtenidas simplemente declarando los tipos de los valores. Además, diferentes políticas de seguridad pueden ser obtenidas variando el sistema de tipos. Otra ventaja importante es que utilizando ciertos sistemas de tipos los invariantes de ciclos pueden ser generados automáticamente por un compilador certificante.

Expressions:	e ::=	$me \mid be$
MathExpressions:	me ::=	$x \mid me_1\ op\ me_2 \mid -me$
MathOperations:	op ::=	$op_{asm} \mid op_{div}$
AddSubMulOperations:	op_{asm} ::=	$+ \mid - \mid *$
DivisionOperations:	op_{div} ::=	$/$
BoolExpressions:	be ::=	$b \mid be_1\ bop\ be_2 \mid not\ be \mid me_1\ rop\ me_2$
BoolOperations:	bop ::=	&& | ||
RelationalOperations:	rop ::=	$> \mid < \mid \geq \mid \leq \mid == \mid \neq$
Types:	τ ::=	bt | array (bt , me)
BasicTypes:	bt ::=	int (Min, Max) | boolean
Predicates	P ::=	$P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x P_x \mid me_1 \leq me_2$
		$\mid e : \tau \mid SaferRead(x) \mid SaferWrite(x_1, x_2)$

Figura 3.3: Sintaxis Abstracta de las Expresiones de Tipos.

Las figuras 3.3 y 3.4 establecen formalmente la política de seguridad por medio de la definición de las expresiones, tipos, predicados y reglas de tipado para un subconjunto extendido del lenguaje C (denominado Mini). La categoría sintáctica *Expressions* define todas las expresiones matemáticas y booleanas. Las expresiones matemáticas son definidas por *MathExpressions* y las expresiones booleanas por *BoolExpressions*. Por su parte, *Types* define los tipos clasificándolos en dos: los tipos básicos (enteros y booleanos) y el tipo arreglo. El tipo entero, además del valor, tiene dos atributos que representan el rango de valores que puede llegar a tomar en un determinado estado (el mínimo y el máximo). Los elementos de un arreglo pueden ser de tipo booleano o entero. Los predicados $SaferRead(x)$ y $SaferWrite(x_1, x_2)$ definen cuándo es seguro leer una variable y cuándo es seguro escribir en una variable, respectivamente. El predicado $e : \tau$ significa que la expresión e es de tipo τ.

Además de la descripción de la sintaxis, el sistema de tipos es definido por un conjunto de reglas de inferencia. En las reglas de la figura 3.4 el supraíndice de los tipos básicos indica si la variable está inicializada (+), no está inicializada (−) o si su estado no es relevante para la regla en que es usada (?). Las reglas **op_int**, **op_boolean**, **op_rel_boolean**, **minus_int** y **not_boolean** definen el tipo de las operaciones booleanas, matemáticas y relacionales. Las reglas **read_int** y **read_boolean** definen cuándo es seguro leer una variable, lo cual sucede cuando la variable está inicializada. La regla **read_array** define cuándo es seguro leer una posición de un arreglo. Esta regla indica que es seguro leer del arreglo a en la posición i cuando la variable i es un entero inicializado, cuyo valor se encuentra entre 0 y la longitud del arreglo menos uno ($0 \leq i < length(a)$).

Las reglas **write_int** y **write_boolean** indican cuándo es seguro cambiar el valor de una variable y establecen que la variable que cambia su valor queda inicializada. Además, la regla

$$\frac{x_1 : int^+ \qquad x_2 : int^+}{x_1 \ op \ x_2 : int^+} \ \textbf{op\_int} \qquad \frac{x_1 : int^+}{- \ x_1 : int^+} \ \textbf{minus\_int}$$

$$\frac{x_1 : boolean^+ \qquad x_2 : boolean^+}{x_1 \ bop \ x_2 : boolean^+} \ \textbf{op\_boolean} \qquad \frac{x_1 : boolean^+}{not \ x_1 : boolean^+} \ \textbf{not\_boolean}$$

$$\frac{x_1 : int^+ \qquad x_2 : int^+}{x_1 \ rop \ x_2 : boolean^+} \ \textbf{op\_rel\_boolean}$$

$$\frac{x : int^+}{SaferRead(x)} \ \textbf{read\_int} \qquad \frac{b : boolean^+}{SaferRead(b)} \ \textbf{read\_boolean}$$

$$\frac{a : array(bt, length) \qquad i : int^+ \qquad 0 \leq i.Min \qquad i.Max < length}{SaferRead(a[i]) \qquad a[i] : bt^+} \ \textbf{read\_array}$$

$$\frac{x_1 : boolean^? \qquad x_2 : boolean^+ \qquad x_1 = x_2}{SaferWrite(x_1, x_2) \qquad x_1 : boolean^+} \ \textbf{write\_boolean}$$

$$\frac{x_1 : int^? \qquad x_2 : int^+ \qquad x_1 = x_2}{SaferWrite(x_1, x_2) \qquad x_1 : int^+ \qquad x_1.Min = x_2.Min \qquad x_1.Max = x_2.Max} \ \textbf{write\_int}$$

$$\frac{a : array(bt, length) \qquad i : int^+ \qquad 0 \leq i.Min \qquad i.Max < length \qquad x : bt^+}{SaferWrite(a[i], x)} \ \textbf{write\_array}$$

$$\frac{x_1, x_2 : int^+ \qquad x_3 = x_1 \ op_{asm} \ x_2 \qquad x_3 : int^?}{x_3.Min = Min(x_1.Min \ op_{asm} \ x_2.Min, \ x_1.Min \ op_{asm} \ x_2.Max, \ x_1.Max \ op_{asm} \ x_2.Min, \ x_1.Max \ op_{asm} \ x_2.Max)} \ \textbf{var\_minor}$$

$$\frac{x_1, x_2 : int^+ \qquad x_3 = x_1 \ op_{asm} \ x_2 \qquad x_3 : int^?}{x_3.Max = Max(x_1.Min \ op_{asm} \ x_2.Min, \ x_1.Min \ op_{asm} \ x_2.Max, \ x_1.Max \ op_{asm} \ x_2.Min, \ x_1.Max \ op_{asm} \ x_2.Max)} \ \textbf{var\_mayor}$$

$$\frac{x_1, x_2 : int^+ \qquad x_3 = x_1 \ op_{div} \ x_2 \qquad x_3 : int^? \qquad (x_2.Min > 0) \ or \ (x_2.Max < 0)}{x_3.Min = Min(x_1.Min \ op_{div} \ x_2.Min, \ x_1.Min \ op_{div} \ x_2.Max, \ x_1.Max \ op_{div} \ x_2.Min, \ x_1.Max \ op_{div} \ x_2.Max)} \ \textbf{div\_minor}$$

$$\frac{x_1, x_2 : int^+ \qquad x_3 = x_1 \ op_{div} \ x_2 \qquad x_3 : int^? \qquad (x_2.Min > 0) \ or \ (x_2.Max < 0)}{x_3.Max = Max(x_1.Min \ op_{div} \ x_2.Min, \ x_1.Min \ op_{div} \ x_2.Max, \ x_1.Max \ op_{div} \ x_2.Min, \ x_1.Max \ op_{div} \ x_2.Max)} \ \textbf{div\_mayor}$$

Figura 3.4: Reglas del Sistema de Tipos.

write_int también actualiza el valor del rango de la variable. La regla **write_array** garantiza que es seguro escribir el valor x en la posición i del arreglo a si x es del mismo tipo que los elementos del arreglo e i es un entero definido, cuyos posibles valores se mueven entre cero y la longitud de a menos uno ($0 \leq i < length(a)$).

La modificación del rango de una variable entera al ser aplicado algún operador matemático se expresa por medio de las reglas **var_minor**, **var_mayor**, **div_minor** y **div_mayor**, las cuales indican los nuevos valores que toman el mínimo y el máximo, respectivamente. Las reglas para el operador / (**div_minor** y **div_mayor**) incluyen la condición extra de que el rango del divisor no incluya el 0 (el mínimo y el máximo son simultaneamente mayores o menores a 0) ya que sino, el algún caso, podría llegar a dividir por 0 Para el caso en que no se cumpla esta restricción, se insertan chequeos en tiempo de ejecución.

Las reglas **div_minor** y **div_mayor** pueden ser complementadas con las reglas **div_minor_gral** y **div_mayor_gral** (figura 3.5). Estas reglas contempla el caso en que el rango del divisor incluya al 0. Para el caso de la división por 0 el procesador ejecutará una excepción y el programa seguirá siendo seguro.

3.4. Diseño de los Análisis Estáticos

Los análisis estáticos realizados para verificar la política de seguridad están conformados por la identificación de variables inicializadas y por la identificación de los rangos de las variables. A continuación se presentan los análisis estáticos diseñados.

$$\frac{x_1,x_2{:}int^+ \qquad x_3{=}x_1\ op_{div}\ x_2 \qquad x_3{:}int^? \qquad (x_2.Min{<}0\ and\ x_2.Max{>}0)}{x_3.Min{=}Min(x_1.Min\ op_{div}\ 1,\ x_1.Min\ op_{div}\ {-}1,\ x_1.Max\ op_{div}\ 1,\ x_1.Max\ op_{div}\ {-}1)}\ \textbf{div\_minor\_gral}$$

$$\frac{x_1,x_2{:}int^+ \qquad x_3{=}x_1\ op_{div}\ x_2 \qquad x_3{:}int^? \qquad (x_2.Min{<}0\ and\ x_2.Max{>}0)}{x_3.Max{=}Max(x_1.Min\ op_{div}\ 1,\ x_1.Min\ op_{div}\ {-}1,\ x_1.Max\ op_{div}\ 1,\ x_1.Max\ op_{div}\ {-}1)}\ \textbf{div\_mayor\_gral}$$

Figura 3.5: Extensión del Sistema de Tipos para el Tratamiento de los Rangos que Contienen al 0 en la División.

3.4.1. Identificación de Variables Inicializadas

A fin de generar la información necesaria para verificar que toda variable se encuentre inicializada al momento en que se la referencia por primera vez es necesario analizar todas las trazas de ejecución posibles del programa.

Utilizando el árbol sintáctico abstracto presentado en la sección 3.2 se analiza estáticamente el flujo de cada programa. Las variables inicializadas se identifican siguiendo las siguientes reglas:

1. Al comenzar, sólo los parámetros se encuentran inicializados.
2. Las constantes son valores inicializados.
3. Cuando una variable es declarada se la considera no inicializada.
4. Una expresión se dice inicializada si todos sus operadores se encuentran inicializados.
5. Cuando a una variable se le asigna una expresión inicializada se la considera inicializada.
6. En el caso de un ciclo, se verifica que las variables utilizadas en su cuerpo se encuentren inicializadas de antemano o sean inicializadas en su cuerpo (sin embargo, note que las variables que se inicializan en el cuerpo de un ciclo no son consideradas como inicializadas para el resto del programa, ya que puede que no se cumpla nunca la condición del ciclo y por lo tanto el cuerpo nunca se ejecute).
7. En el caso de una sentencia condicional, se verifica que las variables utilizadas en sus dos ramas se encuentren inicializadas de antemano o sean inicializadas en cada una de ellas. Se consideran inicializadas para el resto del programa sólo aquellas variables que fueron inicializadas en ambas ramas.

3.4.2. Identificación de Rangos de las Variables

A fin de determinar si es seguro acceder a una posición i de un arreglo, primero se debe determinar el valor de la expresión i. Obviamente, en muchos casos una expresión puede tomar diferentes valores dependiendo del flujo del programa. Por ejemplo, considere el siguiente bloque de sentencias:

```
{
    int i;
    if (b) {
        i=2;
    }
    else {
        i=4;
    }
    (*)
}
```

En este caso, el valor de la variable i en el punto `(*)` puede ser 2 ó 4. Pero para nuestro análisis, si un índice de un arreglo es válido para 2 ó 4 entonces podemos afirmar que es válido para $2 \leq i \leq 4$. Es por eso que diremos que la variable i se encuentra en el rango $2 \leq i \leq 4$. Por lo tanto, para determinar si el acceso a un arreglo es seguro, primero se determinan los rangos de las variables enteras o, lo que es lo mismo, se determinan las cotas, si es posible, de las variables enteras.

Sin embargo, la determinación de las cotas de las variables no es computable en general. Por lo tanto sólo se resuelven algunos casos de identificación de rangos, los cuales son clasificados en tres categorías:

1. Identificación de rangos de variables que no son modificadas dentro de un ciclo.
2. Identificación de rangos de variables que son modificadas por valores cuyas cotas son constante dentro de un ciclo.
3. Identificación de rangos de variables inductivas que son modificadas dentro de un ciclo.

Variables que no son Modificadas Dentro de un Ciclo

Utilizando el árbol sintáctico abstracto se identifican los rangos de las variables que no son modificadas en un ciclo, siguiendo las reglas que se presentan a continuación:

1. Al comenzar sólo están acotados los parámetros de tipo entero con rangos declarados.
2. Las constantes enteras son valores con rango definido y están acotadas por su valor.
3. Cuando una variable es declarada se considera que su rango no está definido (no está acotada).
4. Una expresión e_1 *op* e_2 está acotada si todos sus operandos están acotados. En cuyo caso su cota está definida por medio de las reglas **var_minor**, **var_mayor**, **div_minor** y **div_mayor** definidas en la figura 3.4.
5. Cuando a una variable se le asigna una expresión acotada, el rango de la variable queda acotado por el rango de la expresión. En el caso en que la expresión no este acotada, el rango de la variable queda indefinido.
6. En el caso de una sentencia condicional se identifican los rangos de las variables en las dos ramas (la rama del cuerpo del `then` y la rama del cuerpo del `else`). Las cotas de las variables para el resto de programa están definidas por la unión de las cotas de las variables en ambas ramas. La unión se realiza de la siguiente manera para cada variable: si en las dos ramas la variable está acotada, entonces el rango resultante estará conformado por el valor mínimo y el valor máximo de las cotas originales. En el caso en que la variable tenga un rango indefinido en alguna de las dos ramas, entonces el resultado es un rango indefinido.

El proceso explicado anteriormente puede aplicarse en el siguiente programa:

```
int programa (int x(2,10), int y, int z, boolean b)
{
    if (b) {
        y=3;
        x=x+1;
     }
     else {
          y=5;
          x=0;
    }
     z=y+x;
     return z;
}
```

En este caso se obtienen los siguientes rangos para la última ocurrencia de las variables: $0 \leq x \leq 11$, $3 \leq y \leq 5$ y $3 \leq z \leq 16$.

Variables que son Modificadas por Valores Cuyas Cotas son Constante Dentro de un Ciclo

Sin necesidad de calcular la cantidad de veces que se ejecuta el ciclo, se pueden calcular los rangos de las variables cuyos valores son constantes dentro del ciclo. Por ejemplo, en el siguiente segmento de programa el rango de las variables x e y al finalizar el ciclo serán $1 \leq x \leq 5$ y $3 \leq y \leq 5$, respectivamente.

```
x=1;
y=5;
while (b) {
    x=y;
    y=3;
}
```

Se analiza estáticamente el flujo del cuerpo de la sentencia `while` y se identifican los rangos de las variables que son modificadas por valores acotados en el ciclo, siguiendo el algoritmo presentado en la sección anterior. Finalmente, los rangos calculados se propagan de acuerdo a las dependencias de las variables.

Variables Inductivas

Cuando se utilizan arreglos en un programa, la forma más común de acceder a ellos es mediante variables inductivas. Una variable es *inductiva* si dentro de un ciclo su valor es modificado sólo por una constante en cada iteración del ciclo. A una variable se la denomina *inductiva lineal* si su valor es incrementado o decrementado por una constante en cada iteración del ciclo. Por ejemplo, en el siguiente programa, que inicializa un arreglo en cero, la variable i es una variable inductiva lineal.

```
int [20] a;
int i=0;
while (i<20) {
    a[i]=0;
    i=i+1;
}
```

Si la variable que interviene en la condición de un ciclo es una variable inductiva lineal y la condición del ciclo es de la forma

Variable Relación Variable_Acotada o

Variable_Acotada Relación Variable,

en donde *Variable* está inicializada y tiene un rango constante (es decir el máximo y el mínimo son iguales) y la *Variable_Acotada* es una variable que está acotada, es decir tiene un rango[1], entonces se puede calcular la cantidad de veces que itera el ciclo, de la siguiente manera:

[1]Note que las constantes son consideradas como variables acotadas donde el máximo y el mínimo son iguales.

Supongamos que la variable i es inductiva lineal y está inicializada en i_{init} y k es una expresión de valor constante (rango $[k.min, k.max]$ donde $k.min = k.max$).

1. Si i es una variable inductiva lineal creciente (es decir de la forma $i = i + incremento$, e $incremento > 0$) y la condición del ciclo es de la forma $i \leq k$ o $i < k$.

$$Sean\ k_m = \begin{cases} k.max + 1 & \text{si la condición del ciclo es de la forma } i \leq k \\ k.max & \text{si la condición del ciclo es de la forma } i < k \end{cases}$$

$$l = \begin{cases} 1 & \text{si } mod(k_m - i_{init}, incremento) \neq 0 \\ 0 & \text{si } mod(k_m - i_{init}, incremento) = 0 \end{cases}$$

   Entonces la cantidad de iteraciones del ciclo ($cant\_itera$) está dado por el cociente entero:

$$cant\_itera = \frac{k_m - i_{init}}{incremento} + l$$

2. Si i es una variable inductiva lineal decreciente (es decir de la forma $i = i - incremento$, e $incremento > 0$) y la condición del ciclo es de la forma $i > k$ o $i \geq k$.

$$Sean\ k_m = \begin{cases} k.min - 1 & \text{si la condición del ciclo es de la forma } i \geq k \\ k.min & \text{si la condición del ciclo es de la forma } i > k \end{cases}$$

$$l = \begin{cases} 1 & \text{si } mod(k_m - i_{init}, incremento * (-1)) \neq 0 \\ 0 & \text{si } mod(k_m - i_{init}, incremento * (-1)) = 0 \end{cases}$$

   Entonces la cantidad de iteraciones del ciclo ($cant\_itera$) está dado por el cociente entero:

$$cant\_itera = \frac{i_{init} - k_m}{incremento * (-1)} + l$$

Suponiendo que la variable i es inductiva lineal, está inicializada en i_{init} y el número de veces que itera el ciclo ($cant\_itera$) se pudo calcular utilizando el algoritmo anterior, entonces se pueden calcular los rangos de las variables inductivas de la siguiente manera:

1. Para cada variable inductiva lineal creciente (es decir de la forma $v = v + incremento$) el invariante está determinado por:

$$v.min = i_{init} \qquad v.max = (incremento * (cant\_itera - 1)) + i_{init}$$

   Mientras que la postcondición es:

$$v.min = (incremento * cant\_itera) + i_{init} \qquad v.max = (incremento * cant\_itera) + i_{init}$$

2. Para cada variable inductiva lineal decreciente (es decir de la forma $v = v - incremento$) el invariante está determinado por:

$$v.min = i_{init} \qquad v.max = i_{init} - (incremento * (cant\_itera - 1))$$

   Mientras que la postcondición es:

$$v.min = i_{init} - (incremento * (cant\_itera)) \qquad v.max = i_{init} - (incremento * (cant\_itera))$$

Capítulo 4

El Compilador Certificante CCMini

En este capítulo se presenta el primer prototipo desarrollado de CCMini, las experiencias llevadas a cabo con él, el análisis de las experiencias y del prototipo y las conclusiones que se obtuvieron. Este capítulo está estructurado de la siguiente manera: en la sección 4.1 se presenta la arquitectura general del compilador certificante desarrollado. En la sección 4.2 se describen los componentes más importantes del prototipo. Un ejemplo de las tareas realizadas por el compilador se encuentran en la sección 4.3. En la sección 4.4 se muestran los resultados de las experiencias realizadas con el prototipo. En la sección 4.5 y se demuestra que la complejidad del proceso de certificación realizado por el prototipo es de orden lineal con respecto a la longitud del programa de entrada.En la sección 4.6 se analiza el producto obtenido. Por último en la sección 4.7 se presenta la inclusión de CCMini en el framework de PCC-SA.

4.1. El Compilador Certificante CCMini

La figura 4.1 muestra los módulos que conforman el compilador certificante y su interacción. Los cuadrados ondulados representan código y los rectangulares representan componentes que manipulan dicho código. El compilador toma como entrada *código fuente* y el *Generador de Código Intermedio* produce un representación abstracta de dicho código (el *código intermedio*). El *código intermedio* puede ser utilizado independientemente del lenguaje fuente y de la política de seguridad. Sobre este *código intermedio*, el *Generador de Anotaciones*, realiza diversos análisis estáticos para generar la información necesaria para determinar el estado del programa en cada punto. Estas anotaciones del código intermedio pueden variar de acuerdo a la política de seguridad seleccionada. El *Generador de Optimizaciones* y el *Verificador* basándose en las anotaciones optimizan el código intermedio y realizan la verificación de seguridad, respectivamente.

Es importante resaltar que 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. Si se encuentra algún punto del programa (en alguna ejecución) en el que forzosamente se viola la política de seguridad, el programa es rechazado.

Si el código pasa el proceso de verificación entonces es seguro. El último proceso llevado a cabo por el *productor de código* varía de acuerdo al ambiente en el que este inmerso. 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 como el 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.

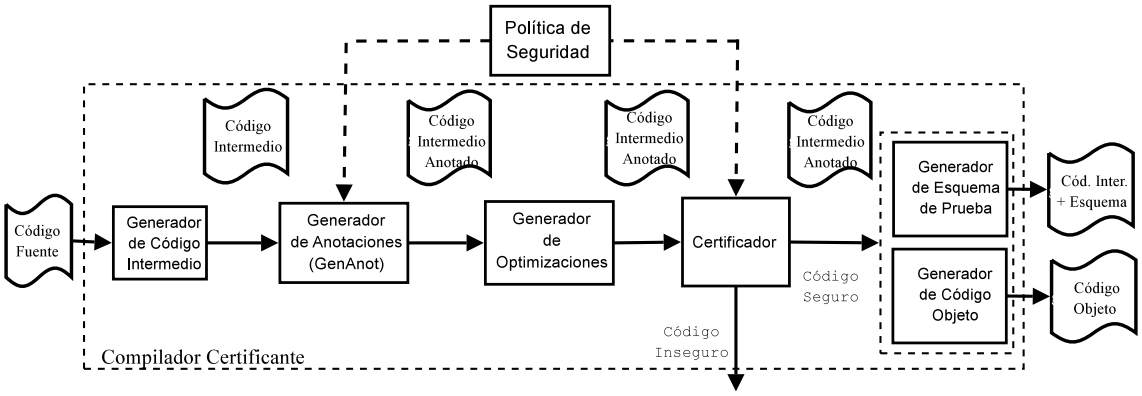

Figura 4.1: Vista global del Compilador Certificante CCMini.

En cambio el segundo caso se inscribe en la problemática de seguridad del código móvil.

En el primer caso, se considera que si el código pasa la etapa de verificación entonces es seguro de ejecutar por lo tanto el *Generador de Código* genera el *código objeto*. En el segundo caso, la última etapa es realizada 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 almacenada en el código intermedio. El esquema de prueba puede ser verificado automáticamente por un consumidor de código foráneo.

4.2. El Prototipo Desarrollado

Con el objetivo de experimentar y analizar lo propuesto se desarrolló un prototipo del *Compilador Certificante* CCMini descripto en la sección 4.1. Como primera medida se definieron los aspectos de seguridad que se deseaban garantizar, es decir, la política de seguridad. Luego se definió el lenguaje fuente del compilador, denominado Mini. El próximo paso consistió en establecer el código intermedio que se utilizaría, un árbol sintáctico abstracto (ASA). Como se comentó en el capítulo 3 esto se definió dentro del marco del framework de PCC-SA y se describió en las secciones 3.3, 3.1 y 3.2, respectivamente. Finalmente se implementaron los módulos correspondientes al *Generador de Código Intermedio*, el *Generador de Anotaciones*, el *Certificador* y el *Generador del Esquema de Prueba*. Estos módulos se describen en las siguientes subsecciones.

Con esta primera implementación se pretende:

- Analizar la efectividad y eficiencia de CCMini.
- Analizar la factibilidad de su inclusión dentro del framework de PCC-SA.
- Analizar la factibilidad del framework PCC-SA.

Posteriormente se introducirá el módulo *Generador de Optimizaciones* y el *Generador de Código Objeto*. En el capítulo 5 se presentan el diseño de las optimizaciones y las extensiones ha introducir en el prototipo.

Este prototipo de compilador certificante posee las siguientes características: seguridad, independencia de la plataforma, generación de código eficiente, verificaciones de seguridad simples. Además, solo descarta aquellos programas que se ha probado fehacientemente que son inseguros. Es decir, si el compilador certificó el código entonces puede ser ejecutado de manera segura. Además, como se mencionó anteriormente, se espera que el prototipo del compilador pueda ser utilizado independientemente del framework de PCC-SA.

El prototipo fue implementado usando el lenguaje de programación Java.

En la sección 4.3 se puede apreciar un ejemplo del proceso realizado por CCMini.

4.2.1. El Generador de Código Intermedio

Esté módulo toma como entrada un archivo con un programa Mini (ver subsección 3.1) y genera un *árbol sintáctico abstracto* (ASA) que lo representa (ver subsección 3.2). El tipo ASA usado es una representación abstracta de alto nivel del lenguaje fuente.

Para realizar esto, primero, se efectúa los análisis léxico y sintáctico del código fuente. Con estos análisis se verifica que el código fuente es efectivamente Mini Paralelamente al *parsing* del código fuente se efectúa un análisis semántico simple. Este último análisis, verifica que no existan ocurrencias de variables no definidas, redefinición de variables y errores de tipos. Entre los errores de tipos podemos nombrar: errores en las expresiones (por ejemplo, $true + 4$), error en el tipo de las condiciones de las sentencias (deben ser siempre de tipo lógico), errores en el tipo de retorno de la función (debe coincidir con el tipo declarado).

A medida que progresan estos análisis el *Generador del Código Intermedio* produce el ASA correspondiente al código Mini provisto. Cabe resaltar que las tareas realizadas por este módulo son análogas a las realizadas por las primeras etapas de un compilador tradicional [4, 6].

Se utilizaron las herramientas JLex [16] y Cup [37] para realizar el análisis léxico y el parser del código fuente. Estas herramientas generan automáticamente los analizadores necesarios para cada etapa a partir de una especificación. JLex toma, básicamente, una especificación de expresiones regulares (similar a la tomada por Lex [44]) y Cup una especificación de una gramática libre de contexto LALR (similar a la tomada por YACC [40, 44]).

4.2.2. El Generador de Anotaciones

El *Generador de Anotaciones* aplica varios análisis estáticos de flujo de control y de datos sobre el ASA y genera un *árbol sintáctico abstracto anotado*. Estas anotaciones incluyen invariantes de ciclos, cotas de cada variable e información de inicialización de variables. En algunos casos se obtienen las pre y postcondiciones. Las cotas de las variables son utilizadas para determinar si los accesos a los arreglos son válidos.

Para lograr realizar estos análisis de manera más eficiente y escalable a programas de gran longitud solo se aplican al cuerpo de funciones. Sin embargo, el *Generador de Anotaciones* implementa un punto medio entre *flow-sensitive analysis*, que considera todos los posibles caminos de un programa, y *flow-insensitive analysis*, que no considera el flujo del programa. Este módulo analiza el flujo del programa, pero en algunos casos solo reconoce ciertos patrones en el código (por ejemplo, cuando analiza los ciclos).

Para generar la información requerida, se aplican los siguientes procesos:

- *Identificación de variables inicializadas.* Analizando todos las posibles trazas de ejecución de un programa se puede detectar referencias a variables no inicializadas. En tal caso, el programa es rechazado.
- *Identificación de rangos de las variables que no son modificadas en los ciclos.* Para esto es necesario considerar las cotas de los parámetros y luego analizar estáticamente el flujo de ejecución de los programas (sin considerar los ciclos) para poder determinar la cota de todas las variables. Analizando las operaciones y sentencias se determinan las posibles cotas de las variables en cada punto de los programas. Este análisis indefine las cotas de aquellas variables que son modificadas en el cuerpo de algún ciclo.
- *Identificación de rangos de variables inductivas.* Una variable se la denomina inductiva lineal si su valor es incrementado o decrementado por una única constante en cada iteración del ciclo. Si la condición de un ciclo está determinada por una variable inductiva lineal es factible determinar la cantidad de iteraciones de dicho ciclo, y por lo tanto, se puede determinar el rango de las variables inductivas de dicho ciclo y el valor de salida (del ciclo) de dichas variables.
- *Identificación de accesos válidos a los elementos de los arreglos.* Con la información obtenida en los pasos anteriores es posible, en muchos casos, determinar si los accesos a arreglos son válidos o no.

Estos procesos fueron implementados siguiendo los algoritmos presentados en la sección 3.4.

4.2.3. El Generador de Optimizaciones

Este módulo toma un ASA anotado y utiliza esta información para realizar las optimizaciones deseadas. Las optimizaciones de código ha realizar son:

1. Propagación de constantes.
2. Eliminación de código muerto.
3. Eliminar asignaciones constantes en los ciclos (código invariante).
4. Eliminación de expresiones redundantes.

Optimizando el código generado se obtiene:

- 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.
- 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, [4, 6, 49, 50]) por lo cual no se considera importante redundar en detalles.

Cabe resaltar que estas optimizaciones pueden redundar, en algunos casos, en el refinamiento de las anotaciones originales. Lo que puede mejorar la efectividad del entorno.

4.2.4. El Certificador

El *Certificador* recorre el ASA cotejando que la información contenida por las anotaciones en cada punto del programa cumple con lo establecido por la política de seguridad. Como por ejemplo, verifica que en cada acceso a un arreglo las cotas de la expresión que determina el valor del índice estén incluídas entre los límites del arreglo. Es decir, que la cota inferior del índice sea mayor o igual al límite inferior del arreglo y que la cota superior sea menor o igual al límite superior del arreglo. Si puede verificar esto para un determinado punto de un programa entonces puede certificar que en todas las posibles ejecuciones del programa ese acceso a el arreglo es seguro. Por lo tanto, elimina la indicación de introducir una verificación dinámica que existe en ese acceso al arreglo. Esto consiste en cambiar la etiqueta `unsafe` del nodo por una etiqueta `safe`.

Si en algún punto logra determinar que existe un acceso a un arreglo inseguro (fuera de los límites) entonces rechaza el programa. Si esto pasa, puede haber sucedido que todos los valores comprendidos dentro de las cotas del índice estaban fuera de los límites del arreglo; o solo un subconjunto de valores pero esto es suficiente para determinar que en alguna de las posibles ejecuciones el acceso será inseguro.

Como se aclaro oportunamente, los análisis estáticos que realiza el *Generador de Anotaciones* no son completos ya que el problema de acotar variables es un problema no computable. Esto puede ocasionar que el *Certificador* no cuente con información sobre el estado de algún punto del programa. En estos casos el *Certificador* no realiza ninguna tarea dejando ese punto como inseguro, es decir, con una indicación de introducir una verificación dinámica.

4.2.5. El Generador del Esquema de Prueba

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 y solo contempla los accesos a arreglos declarados como seguros. Accesos que se desean verificar estáticamente ya que los accesos de seguridad incierta son verificados dinámicamente.

El esquema de prueba generado por el prototipo considera todas las asignaciones a las variables críticas como puntos a verificar (puntos críticos). Entonces el esquema hace referencia a asignaciones que no determinan el valor de los índices con que se accede a un arreglo. En la figura 4.2 se muestra un fragmento de un programa Mini en el cual se puede apreciar esto. 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 el valor de i en otros puntos del programa depende de las variables j y k. Es decir, el esquema de prueba generado por el prototipo para el programa del ejemplo incluye todos los puntos del programa. El ejemplo presentado en la sección 4.3 muestra un esquema de prueba que solo cubre una parte 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 4.2: Ejemplo de un Programa Mini.

4.2.6. El Generador de Código Objeto

Luego de que se verifica la seguridad del código el *Generador de Código Objeto* toma como entrada el árbol sintáctico abstracto anotado y genera código objeto. La ventaja de tener un módulo independiente brinda la posibilidad de generar código objeto utilizando distintos lenguajes ensambladores o incluso generar directamente código binario. Más aún, este módulo podría fácilmente ser reemplazado por un intérprete.

4.3. Ejemplo del Proceso Realizado por el Prototipo

En la figura 4.3 se puede ver un programa fuente Mini. El mismo es una función que luego de inicializar un arreglo de números enteros (*data*) retorna la suma de sus elementos. Se puede notar que la función *ArraySum* toma como parámetro un entero (*index*) que esta acotado por el rango $[0, 0]$ y que la función retorna un valor entero.

Si CCMini toma como entrada el programa de la figura 4.3, el *Generador de Código Intermedio* luego de las verificaciones de rigor genera el ASA correspondiente. En la figura 4.4 se puede ver el ASA generado para el programa de la figura 4.3.

En la figura 4.4 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 constituido por una condición (la expresión $index < 10$) y el cuerpo del mismo. El cuerpo del ciclo

```
int ArraySum ( int index(0,0) )
{      /* Precondicion: index=0 */
    int [10] data;          /* Define el arreglo data de 10 elementos enteros */
    int value=1;            /* variable auxiliar para inicializar el arreglo */
    int sum=0;              /* variable para calcular la sumatoria de arreglo */
    while (index<10) {      /* Inicializa el arreglo */
                    /* Invariante: 0<=index<10  */
        data[index]=value;
        value=value+1;
        index=index+1;
    }
    while (index>0) {       /* Calcula la sumatoria */
                    /* Invariante: 0<index<=10 y
        sum = sumatoria de i=(index-1..9) de data[i]  */
        sum=sum+data[index-1];
        index=index-1;
    }
    return sum;
}
```

Figura 4.3: Ejemplo de un Programa Mini.

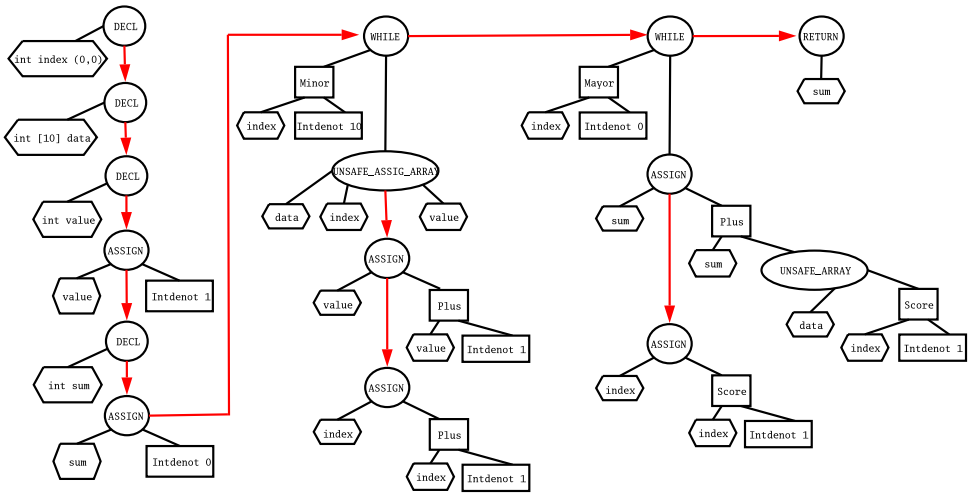

Figura 4.4: ASA del Programa Mini de la Figura 4.3.

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 $value + 1$. Y en la tercera, a *index* se le asigna $index + 1$.

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. Inicialmente todos los accesos son considerados inseguros.

El *Generador de Anotaciones* toma este ASA y realiza los análisis estáticos de flujo de control y de datos que se mencionaron anteriormente. Con estos determina el estado de las variables y sus cotas en cada punto del programa. Con esta información genera las anotaciones y las inserta en el ASA. En la figura 4.5 se muestran algunas de las anotaciones generadas en este punto. Solo se grafican algunas para no sobrecargar la imagen con información y entorpecer su visualización. Los rectángulos grises representan anotaciones. La anotación $index(0, 9)$ significa que el valor de la variable *index* se encuentra acotado 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 punto del ciclo.

Para este ejemplo, el *Generador de Optimizaciones*, no realiza ninguna optimización al ASA. Por lo cual, el *Certificador* toma el ASA anotado de la figura 4.5 y lo recorre cotejando que los accesos al arreglo *data* sean correctos. Es decir, que la variable *index* este acotada por el rango [0.,9] cuando se accede al arreglo *data*. Como esto se cumple el código es seguro, por lo tanto, es pasado al siguiente módulo. En este caso se considero que CCMini se encuentra en un ambiente de código móvil.

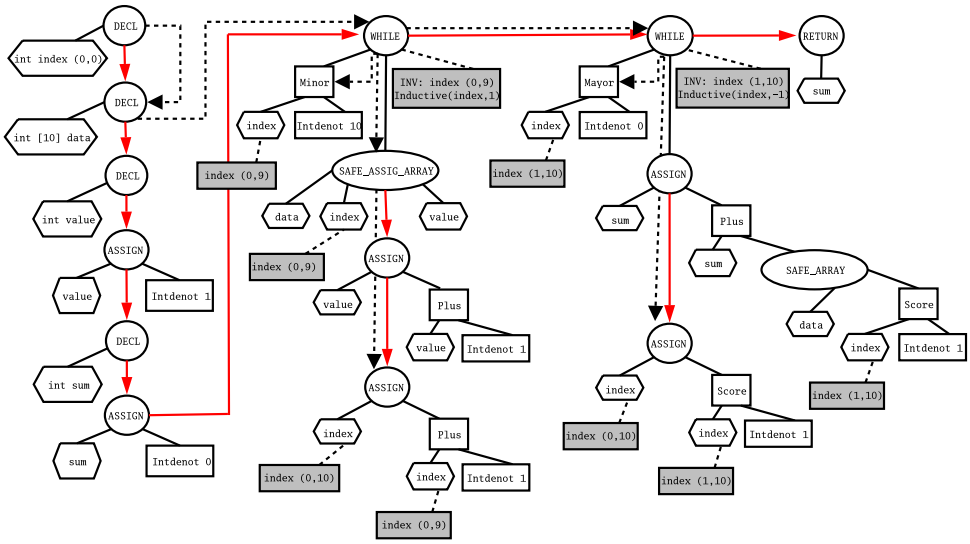

Figura 4.5: Esquema de Prueba del ASA Anotado para el Programa Mini 4.3.

Además, el *Certificador*, si puede determinar que los accesos a los arreglos son seguros modifica el ASA para eliminar la verificación dinámica que tenía en este punto. Esta tarea consiste simplemente en cambiar la etiqueta del nodo UNSAFE por una etiqueta SAFE. Nótese que las figuras 4.4 y 4.5 difieren en las etiquetas de los nodos correspondientes a accesos a arreglos. En este caso se pudo determinar que los accesos son seguros.

Por último, 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 4.5 se muestra el esquema de prueba producido para el programa de la figura 4.3. Las líneas de puntos representan el camino de prueba generado.

4.4. Experiencias

Para analizar la eficiencia y eficacia de CCMini se realizó una serie de pequeños experimentos. Estos experimentos consistieron en utilizar como entrada de CCMini 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 CCMini y 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 las figuras 4.6, 4.7 y 4.8, se presentan los resultados de los análisis de los

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 a CCMini realizar la generación del código y la certificación de la política de seguridad.

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

Figura 4.6: Experiencias con Algoritmos de Ordenación.

Los algoritmos de ordenación (figura 4.6) 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. CCMini 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.*
Sequential Search	1735 bytes	1344 bytes (77.46 %)	391 bytes (22.54 %)	100 %	0.363 seg.
Binary Search	2028 bytes	1593 bytes (78.55 %)	435 bytes (21.45 %)	100 %	0.368 seg.

Figura 4.7: Experiencias con Algoritmos de Busqueda.

Programa	*Prog. y Prueba*	*T. Prog.*	*T. Prueba*	*Arreglos sin RTC*	*Tpo. de Comp.*
Merge Array	3824 bytes	3235 bytes (84.60 %)	589 bytes (15.40 %)	100 %	0.639 seg.
Factorial	881 bytes	719 bytes (81.61 %)	162 bytes (18.39 %)	100 %	0.371 seg.
All	20563 bytes	17696 bytes (86.06 %)	2867 bytes (13.94 %)	96 %	1.947 seg.

Figura 4.8: Experiencias con otros Algoritmos.

Las experiencias con los algoritmos de búsqueda (figura 4.7), *Merge Array* y *Factorial* (figura 4.8) son más simples que los algoritmos de ordenación. Pero los resultados obtenidos permiten verificar que tanto los tiempos de verificación como el tamaño de la prueba se mantienen entre los

mismos rangos de valores.

La última experiencia realizada consistió en incluir todos los algoritmos mencionados en un solo programa (denominado *All*). Con este programa *All* se pretende mostrar que el compilador tiene un comportamiento similar con programas de mayor tamaño. Como se puede ver en la figura 4.8 el tamaño de la prueba se mantiene por debajo del tamaño del programa (en un porcentaje similar a los anteriores). Además, el tiempo de compilación obtenido permite suponer que este tiempo crece linealmente con respecto al tamaño de los programas usados.

Estos experimentos permiten suponer que si bien el costo de compilación es mayor que el de un compilador tradicional CCMini 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 resultó lineal al tamaño de los programas usados.
- 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.

4.4.1. 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 4.9 se pueden observar algunos de los resultados obtenidos de la observación realizada.

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 4.9: Resultados de la Observación Informal de Aplicaciones `C`.

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 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.

4.5. Análisis de la Complejidad Temporal de la Certificación de Código

En esta sección se verá que para una amplia familia de programas (que parecería abarcar a los que se encuentran en la práctica) el proceso de certificación de código realizado por CCMini es lineal respecto de la longitud de los programas fuente. Primero, se caracterizará una familia de ASAs para los cuales se demostrará que el tiempo de certificación es lineal respecto de su longitud. Luego, en la sección 4.5.1 se verá que todos los programas de la muestra analizada en la sección 4.4.1 pertenecen a la familia anteriormente introducida. Finalmente, en la sección 4.5.2 se harán consideraciones sobre el peor caso de comportamiento de CCMini siendo que es a lo sumo cuadrático.

Teorema 1 *La cantidad de nodos de un ASA esta acotada por una función lineal que depende de la longitud del código de entrada.*

Demostración 1

A- La cantidad de pasos realizados por un parser LR es lineal con respecto a la cadena de entrada. Esto está demostrado en A. Aho y J. Ullman ([2], p. 395).

B- La cantidad de nodos de un árbol de parsing LR está acotada por una función lineal que depende de la longitud del programa de entrada. Esta afirmación es válida puesto que, por la afirmación anterior, la cantidad de pasos de un parser LR es lineal con respecto a la cadena de entrada y que en cada paso de un parser LR se corresponde con a lo sumo la introducción de un nodo en el árbol de derivación (árbol de parsing).

C- El ASA utilizado por CCMini se construye a partir del árbol de parsing eliminando los nodos requeridos por la sintaxis concreta y dejando sólo aquellos nodos que corresponden a la sintaxis abstracta.

Por C- se puede afirmar que el ASA es el resultado de una poda del árbol de parsing y por B- se puede afirmar que la cantidad de nodos del ASA esta acotada por la función lineal que acota la cantidad de nodos del árbol sintáctico.

Pudiéndose concluir que la cantidad de nodos de un ASA esta acotada por una función lineal que depende de la longitud del programa de entrada. □

Definición 1 *Diremos que un ASA A es de anotación acotada por $t = (c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}, V, M) \in \mathbb{N}^8$ si A satisface las siguientes condiciones:*

- *c_l es una cota superior de la cantidad de computaciones elementales necesarias para generar las anotaciones para los nodos de A de tipo l ($l \in \{EXP, EXP\_OP, ASSING, RETURN, IF, WHILE\}$).*
- *V es una cota superior del número de veces que se visita cada nodo de A.*
- *M es una cota superior de la cantidad de variables que son utilizadas en A.*

Donde EXP representa a los nodos hojas del ASA (identificadores y constantes), EXP_OP a los nodos de las expresiones, ASSING a las asignaciones, RETURN representa a la sentencia de retorno, IF a la sentencia condicional y WHILE representa a los nodos que corresponden a las sentencias iterativas.

Definición 2 *Dada una tupla $t \in \mathbb{N}^8$, llamaremos $\mathcal{F}(t)$ a la familia de ASAs de anotación acotada por t.*

$$\mathcal{F}(t) = \{A : ASA \mid A \text{ es de anotación acotada por } t\}$$

Teorema 2 *Dada* $t = (c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}, V, M)$*, para todos los* ASA *de* $\mathcal{F}(t)$ *el proceso de generación de las anotaciones tiene un comportamiento temporal lineal con respecto a la cantidad de nodos del* ASA*.*

Demostración 2 *Sea* Γ *la cantidad de computaciones elementales que insume el proceso de generación de anotaciones en cada visita a un nodo.*

*Sea A:*ASA *tal que* $A \in \mathcal{F}(t)$*, si* $\{nodo_1, ..., nodo_n\}$ *es el conjunto de nodos de A y* v_i *la cantidad de ciclos en que se visita al nodo* `nodo_i` *durante el proceso de generación de las anotaciones entonces la cantidad de computaciones elementales que necesita el proceso será:*

$$\sum_{i=1}^{n} v_i * \Gamma(nodo_i)$$

Sea $k = max\{c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}\}$ *resulta que:*

$$\sum_{i=1}^{n} v_i * \Gamma(nodo_i) \leq \sum_{i=1}^{n} V * k,$$

Además se cumple que:

$$\sum_{i=1}^{n} V * k = n * (V * k), \quad donde(V * k) \in \mathtt{N}$$

Entonces la complejidad temporal para generar las anotaciones tiene un orden $\mathrm{O}(n{*}V{*}k){=}\mathrm{O}(n)$ *como se quería demostrar.* □

Teorema 3 *Para todo programa cuyo* ASAA *(generado por* CCMini*) sea de anotación acotada por* $t \in \mathtt{N}^8$ *el proceso de generación de las anotaciones tiene un comportamiento temporal lineal con respecto a la longitud del código de entrada.*

Demostración 3 *La demostración es trivial dado que por Teorema 2 se puede afirmar que la generación de anotaciones es* O(n) *con respecto a la cantidad de nodos de A y por Teorema 1 se sabe que la cantidad de nodos de A es de orden lineal con respecto a la longitud del programa de entrada.* □

Teorema 4 *Para todo programa cuyo* ASAA *(generado por* CCMini*) sea de anotación acotada por* $t \in \mathtt{N}^8$ *el proceso de certificación del código tiene un comportamiento temporal de orden lineal con respecto a la longitud del programa de entrada.*

Demostración 4 *El proceso de certificación de A consta de dos pasos: (1) la generación de anotaciones, proceso cuya complejidad temporal es lineal con respecto a la entrada (afirmación válida por Teorema 1); y (2) la certificación de las anotaciones, proceso que consiste en realizar un recorrido exhaustivo de A verificando que las anotaciones generadas cumplen con la política de seguridad. En este proceso se visita una sola vez cada nodo de A por lo tanto su complejidad es lineal con respecto a la longitud del programa de entrada.*

Entonces se puede afirmar que la complejidad temporal del proceso de certificación es de O(n) *con respecto a la longitud del código de entrada.* □

4.5.1. Programas de Anotación Acotada

En esta sección se muestra que los `ASAs` correspondientes a programas usados en la práctica (que parecería abarcar a la gran mayoría) son de *anotación acotada* por una tupla $t \in \mathtt{N}^8$. En particular, se determina una tupla $t = (c_{exp}, c_{exp\_op}, c_{assing}, c_{return}, c_{if}, c_{while}, V, M)$ tal que los programas de la muestra de la sección 4.4.1 pertenezcan a $\mathcal{F}(t)$.

A continuación se establecen los componentes de t:

M El análisis informal de funciones `C` de la sección 4.4.1 muestra que la cantidad de variables utilizadas en una función no supera la cantidad de 70. Entonces es posible definir, para los programas de la muestra, un valor constante para M ($M = 70$) que sea cota de la cantidad máxima de variables utilizadas en una función.

V Los nodos de un ASA son visitados una vez durante el proceso de generación de las anotaciones salvo en el caso de aquellos nodos que forman parte del cuerpo de un ciclo (que pueden ser visitados hasta dos veces). Pero, en el caso de existir varios niveles de anidamiento los nodos de cada ciclo (anidado) serán visitados dos veces más una visita por cada uno de los ciclos que lo contengan. Es decir, si tenemos W ciclos anidados (nivel 1, 2, ..., W de anidamiento) entonces los nodos del ciclo W_i (i=1...W) serán visitados $2 + (i - 1)$ veces.

El análisis informal de funciones `C` mencionado anteriormente permite observar que en raras ocasiones existen más de seis niveles de anidamiento de ciclos. Por lo tanto se puede definir una constante A ($A = 6$) cota del máximo nivel de anidamiento. Y se puede definir la constante $V = 2 + (6 - 1) = 7$ como cota máxima de la cantidad de veces que puede ser visitado un nodo.

c_{exp} esta constante está dada por la cantidad de computaciones elementales necesarias para determinar el tipo del objeto que contiene la hoja (1 computación elemental) y si el mismo está inicializado o no (1 computación elemental). Si es de tipo entero hay que sumar los pasos necesarios para recuperar su cota. Para recuperar la cota se pueden dar dos casos: (1) que el nodo corresponda a un valor constante por lo tanto recuperar su cota consiste en recuperar ese mismo valor (1 computación elemental); o (2) que el nodo corresponda a una variable por lo cual el costo esta determinado por el costo de recuperar la información de la variable de la tabla de símbolos. Se debe recordar que esta tabla de símbolos esta implementada con una tabla de hash.

Si la tabla de hash está dividida en B posiciones y hay M elementos almacenados en la tabla entonces cada posición tiene, en promedio, M/B elementos y se puede esperar que la operación para recuperar un elemento lleve un tiempo $O(1 + M/B)$. La constante 1 representa la computación elemental necesaria para hallar la posición, y M/B las computaciones para buscar en ella. Si puede elegirse B casi igual a M, este tiempo se convierte en una constante. Por lo tanto, el tiempo promedio para recuperar un elemento de la tabla de hash, suponiendo que un elemento cualquiera tiene la misma probabilidad de ser dispersado en cualquier posición [1], es una constante que no depende de M [3]. La tabla de hash utilizada en el prototipo cuenta con 50 posiciones por lo tanto el análisis anterior es válido para la implementación presentada en este trabajo. En el peor caso, en el cual todos los identificadores se encuentren almacenados en la misma posición de la tabla, el costo se corresponde con una busqueda secuencial. Por lo tanto, la cantidad máxima de computaciones elementales es de $70 + 1$.

Es decir, analizar una hoja del ASA tiene un costo acotado por la constante $c_{exp} = (70+1)+2$.

c_{exp_op} esta constante está dada por la cantidad máxima de computaciones elementales necesarias para analizar si ambos operandos de la expresión están inicializados (2 computaciones elementales). Además, si la expresión es de tipo entero y esta inicializada se deben considerar los pasos necesarios para calcular la cota de la expresión (4 computaciones elementales). Por lo tanto, $c_{exp\_op} = 6$.

c_{assing} esta constante está dada por la cantidad de computaciones elementales necesarias para recuperar la información de la variable destino más los pasos necesarios para actualizar su información de inicialización y cotas con la información correspondiente a la expresión que se le asigna. Como se comentó anteriormente el costo de recuperar la información de una variable de la tabla de hash está acotada por una constante, al igual que el costo de actualizar la información de su cota (1 computación elemental). Por lo tanto el costo para analizar una asignación está acotado por un valor constante, $c_{assing} = c_{exp} + 1$.

[1] A. Aho, J. Hopcroft y J. Ullman [3] presentan *"una función de dispersión para cadenas de caracteres que es muy buena, aunque no perfecta"*. Dicha función es la que se utilizó en la implementación de la tabla.

c_{return} esta constante está dada por la cantidad de computaciones elementales necesarias para actualizar la información de inicialización y de cotas correspondiente a los valores de retorno de la función involucrada. Esta información se obtiene simplemente consultando la información almacenada en la expresión a retornar (1 computación elemental), es decir, $c_{return} = 1$.

c_{if} esta constante está dada por la cantidad de computaciones elementales necesarias para generar una copia de la tabla de símbolos (una instancia de la tabla se utiliza como entrada para analizar el cuerpo del `then` y la copia como entrada para analizar el cuerpo del `else`). Se deben adicionar computaciones elementales necesarias para unir las cotas de las variables que se encuentran en ambas tablas (la copia y la original) al finalizar el análisis de ambas ramas.

La constante c_{if} se puede determinar si se considera el resultado del análisis informal de funciones `C` mencionado anteriormente. Considerando este resultado se puede encontrar M cota de la cantidad máxima de variables utilizadas. Si la tabla tiene B posiciones, M cota de la cantidad de variables y en el peor caso M elementos en una posición entonces las operaciones de copia y unión de las tablas tienen un costo máximo de $B + M$. Por lo tanto, c_{if} es un valor constante definido por $2 * (B + M) = 2 * (50 * 70)$.

c_{while} esta constante está dada por la cantidad de computaciones elementales necesarias para generar una copia de la tabla de símbolos (la copia se utiliza como entrada para analizar el cuerpo del `while`). Se debe adicionar los pasos necesarios para unir las cotas de las variables que se encuentran en ambas tablas (la copia y la original) al finalizar el análisis de cuerpo del ciclo. Lo cual tiene un costo acotado por una constante y se desprende de un análisis similar al realizado en el punto anterior.

Al analizar los ciclos se debe recordar que primero se realiza un análisis (una pasada por el cuerpo del `while`) para determinar: (1) inicialización de las variables; (2) asignaciones constantes a variables (su valor no esta dado por la cantidad de iteraciones); y (3) variables inductivas. Si se determina la existencia de variables inductivas se debe determinar la cantidad de iteraciones (si es posible). Por lo cual, hay que sumar a c_{while} la cantidad de pasos necesarios para determinar el número de iteraciones del ciclo que se esta analizando (2 computaciones elementales más). Es decir, $(c_{while} = 2 * (B + M) + 2 = 2 * (50 + 70) + 2)$

En el caso de poder acotar el número de iteraciones del ciclo se debe realizar otra pasada que analice el cuerpo del `while` utilizando la información obtenida en los pasos anteriores.

4.5.2. Análisis de la Complejidad en el Peor Caso

Un peor caso a considerar es aquel en el cual los ciclos anidados pueden conformar todo el programa entonces cada nodo será visitado un máximo de veces

$$2 * (n - 1) \quad = \quad 2 * n - 2$$

donde n es la cantidad de nodos. Se debe notar que en este peor caso cada ciclo anidado tendrá solo un nodo menos que el ciclo que lo contiene. En el Teorema 2 se demostró que la complejidad temporal es $O(n * V * k)$.

Entonces dado $V = 2 * n - 2$, número máximo de veces que puede ser visitado un nodo, la complejidad temporal es $O(n * (2 * n - 2) * k) = O(n^2)$. Es decir, en este peor caso, la complejidad es polinomial de orden 2. Lo cual sigue siendo mucho menor a la complejidad exponencial de los procesos de certificación basados en demostradores de teoremas (tal como `Touchstone`).

Otro peor caso a considerar es aquel en el cual k (cota de máxima cantidad de pasos elementales para analizar cado nodo) dependa de la cantidad de nodos. Por ejemplo, porque la cantidad de variables es igual a la cantidad de nodos (n) lo que trae aparejado que el costo de recuperar un elemento de la tabla de símbolos depende de n. Entonces $k = n + c$ por lo tanto en este caso la complejidad temporal es $O(n * V * k) = O(n * V * n + c) = O(n^2)$ (con c y V valores constantes).

4.6. Análisis del Prototipo

Este prototipo, CCMini, provee las siguientes características: 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 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. Además la generación de la certificación también es lineal con respecto al tamaño de los programas. No hay que dejar de notar que estas características son fundamentales en un ambiente de código móvil.

El prototipo sólo descarta 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 muestra que generalmente los accesos a arreglos se realizan mediante variables inductivas. Esto sugiere que se puede determinar en la mayoría de los casos prácticos la validez de los accesos evitando la necesidad de insertar chequeos dinámicos.

CCMini, puede ser utilizado independientemente del prototipo PCC-SA. También puede utilizarse como un asistente de depuración de programas. CCMini informa en tiempo de compilación posibles errores durante la ejecución evitando posteriores revisiones de código y/o la realización de testing.

4.7. Integración de CCMini al framework de PCC-SA

Dado que la interfaz entre el productor y el consumidor, del framework de PCC-SA, fue claramente diseñada este paso fue trivial y sin complicaciones. El framework 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.

De las experiencias realizadas con el framework se desprendieron resultados alentadores que permitieron corroborar la factibilidad del mismo en el mundo real. Se pueden certificar propiedades realmente interesantes desde el punto de vista de la seguridad de manera eficiente y generando pruebas de tamaño lineal con respecto al tamaño de los programas. Si bien las tareas a realizar por el consumidor de código son similares a las del productor el esfuerzo que debe realizar es significativamente menor y utilizando una cantidad menor de recursos.

Futuros cambios o extensiones de CCMini solo influenciarán cuando la representación del ASA o del *esquema de prueba* es modificada. En cuyo caso, solo el *Generador del Esquema de Prueba* deberá ser modificado para soportar los cambios y cumplir con su rol de interfaz entre productor-consumidor. Extensiones del lenguaje y la política de seguridad deben contemplar la introducción de nuevos análisis estáticos, por lo cual, no solo se debe extender el *Generador de Anotaciones* y el *Certificador* sino también el *Verificador* perteneciente al consumidor. Estas modificaciones pueden realizarse por medio de módulos independientes, por lo cual, no debería existir la necesidad de modificar el código existente.

Capítulo 5

Optimización y Extensión de CCMini

El prototipo de CCMini aquí presentado es un primer paso a la obtención de un entorno de generación de código móvil seguro que pueda ser usado industrialmente. Por lo cual es necesario continuar su desarrollo. En este capítulo se plantean una serie de optimizaciones y extensiones tendientes a obtener un compilador que sea una alternativa potencial para ser usada en ambientes con código móvil inseguro. Las optimizaciones tienen como fin proporcionar un ambiente más eficiente y que utilice menos recursos. Las extensiones tienden a proporcionar un ambiente más flexible y potente.

En la sección 5.1 se presenta la optimización propuesta para el *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 5.2 se presentan las extensiones propuestas.

5.1. Optimización del Generador de Anotaciones

El *Generador de Anotaciones*, como se mencionó en el capítulo 4, realiza diversos análisis de flujo de control y de datos sobre el árbol sintáctico abstracto (ASA). Estos análisis determinan la inicialización y las cotas de las variables utilizadas. Esto último permite asegurar si un valor es válido como índice de un arreglo. Además, en ciertos casos se determinan los 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* que almacena información de las variables. 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 una tabla de *hash* como estructura de datos auxiliar para realizar el análisis de cotas se propone utilizar como estructura de datos una *pila de contextos* de cada variable. Cada elemento de estas *pilas de contextos* contiene información del estado de cada variable en determinado contexto. Donde por *contexto* se referiere a cada bifurcación del flujo del programa.

Cada ocurrencia de una variable referencia al mismo objeto. Este objeto contiene información de tipo, estado de inicialización y una pila de contextos (en la que cada elemento tiene información del estado de la variable en cada contexto). Por ejemplo, variables de tipo entero mantienen una pila de contexto donde cada elemento de la pila contiene información de cotas de un determinado contexto (por ejemplo, las cotas correspondientes a cada rama de una sentencia condicional).

En la figura 5.1 se presenta el algoritmo para calcular cotas con esta nueva estructura. Se considera que:

- Las constantes enteras son valores con rango definido y están acotados por su valor.
- Inicialmente el rango de las variables no está definido, salvo en el caso de los parámetros acotados.
- 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.

---

1. **nivel** es igual a cero y **variables** está vacío.
2. Una expresión e_1 *op* e_2 se dice que está acotada por e_{Min}, e_{Min} si todos sus operandos están acotados y su cota está definida por medio de las reglas **var_minor**, **var_mayor**, **div_minor** y **div_mayor** definidas en la figura 3.4.
3. Recorrer todo el árbol:
   - *a*) En cada acceso a arreglo **A[exp]**: Determinar (exp_{Min}, exp_{Max}) cota de **exp**. Por cada acceso seguro a arreglo en **exp** hacer $3a$).
   - *b*) En cada **return exp**: Determinar (exp_{Min}, exp_{Max}) cota de **exp**. Por cada acceso seguro a arreglo en **exp** hacer $3a$).
   - *c*) En cada asignación **x = exp**: Determinar (exp_{Min}, exp_{Max}) cota de **exp**. Por cada acceso seguro a arreglo en **exp** hacer $3a$). Además,
     1) Si **x** es un acceso a arreglo hacer $3a$.
     2) En otro caso: Si **nivel** es distinto de cero y **x** no está en **variables[i]**: insertar **x** en **variables[i]**, donde $0 \leq i \leq nivel$.
        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**:
     1) Analizar la expresión **cond** para: determinar cotas y encontrar accesos a arreglos (en caso de encontrar algún acceso a arreglo hacer $3a$).
     2) Incrementar **nivel** en uno. Procesar el cuerpo del **then**.
     3) Incrementar **nivel** en uno. Procesar el cuerpo del **else**.
     4) 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**) y almacenar el resultado en el tope.
     5) Decrementar nivel en dos.
   - *e*) En cada iteración **WHILE cond cuerpo**:
     1) Analizar la expresión **cond** para: determinar cotas y encontrar accesos seguros a arreglos (en caso de encontrar algún acceso a arreglo hacer $3a$).
     2) 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**) y almacenar el resultado en el tope.
     3) Decrementar nivel en uno.

Figura 5.1: Algoritmo para Generar las Anotaciones

El algoritmo recorre el ASA definiendo en cada asignación la cota de la variable receptora. La *pila de contextos* tiene su utilidad cuando el flujo del programa analizado se divide en ramas. Por ejemplo en el caso de un `if` se identifican las cotas de las variables en las dos ramas (la rama del cuerpo del `then` y la rama del cuerpo del `else`). Al iniciarse el proceso en cada rama la cota de cada variable es igual a su cota antes de la sentencia condicional. Las cotas de las variables para el resto de programa está definido por la unión de las cotas determinada en la rama del cuerpo del `then` y en la rama del cuerpo del `else`. Esta unión de cotas es un proceso que define las cotas de las variables teniendo en cuenta los dos contextos anteriores. Para cada variable: si en las dos ramas la variable está acotada, la cota resultante es la conformada por el valor mínimo y el valor máximo de las dos cotas consideradas. En el caso de que alguna de las dos cotas este indefinida el resultado es una cota indefinida.

Esta optimización redunda en una mejora en el tiempo de ejecución del Generador de Anotaciones. 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.

Otro mejora reside en la eliminación de la necesidad de copiar, unir e intersecar tablas de hash. Este proceso lo realiza el prototipo al procesar sentencias **if** y **while**. En cambio, con este algoritmo solo se necesita contar con referencias 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 puede resumir en:

- En un menor tiempo de ejecución para acceder a la información de variables porque se eliminó la necesidad de recuperar la información de las mismas de la tabla de hash. En el nuevo algoritmo esta recuperación de información es de costo constante (solo se debe acceder al objeto a referencido por el nodo del ASA).
- En un menor tiempo de ejecución porque se eliminó la necesidad de copiar la información mantenida en la tabla de *hash* y por lo tanto tampoco se deben realizar los procesos de unión e intersección de las tablas de *hash*.
- En un menor espacio de memoria utilizado, por la eliminación de la copia de la tabla 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).

No hay que dejar de mencionar que esta estructura de datos, la *pila de contextos* puede ser usada para realizar otros análisis. Por ejemplo, en la figura 5.5 se introduce un algoritmo de análisis de dependencias que utiliza esta estructura. Además, la estructura de datos utilizada por el Generador de Anotaciones es usada por los demás módulos del compilador. Por lo tanto las mejoras introducidas son globales al compilador.

5.1.1. Ejemplo del Proceso Realizado por el Algoritmo que Genera la Anotaciones

En la figura 5.2 se muestra un fragmento de programa Mini. En este fragmento intervienen las variables enteras: a, b, x, z. Y se asume que a esta acotado por $(0, 5)$ y b por $(2, 8)$. La figura 5.2 presenta el ASA correspondiente a este fragmento de algoritmo con las pilas de contextos correspondientes a cada variable.

Cuando comienza el análisis (realizado por el algoritmo de la figura 5.1) la pila de contextos de cada variable contiene información sobre el contexto actual. Cuando el análisis llega a la línea `/*1*/` del fragmento de programa las cotas de x y de z no están definidas. No sucediendo lo mismo con las cotas de a y b. El resultado de procesar la línea `/*1*/` se puede ver en la información de las pilas de contexto graficadas en la figura 5.3(a).

Luego, se procesa el condicional de la línea `/*2*/`. El efecto de analizar el cuerpo del then se puede apreciar en la figura 5.3(b) y el resultado obtenido después de procesar el cuerpo del else en la figura 5.3(c).

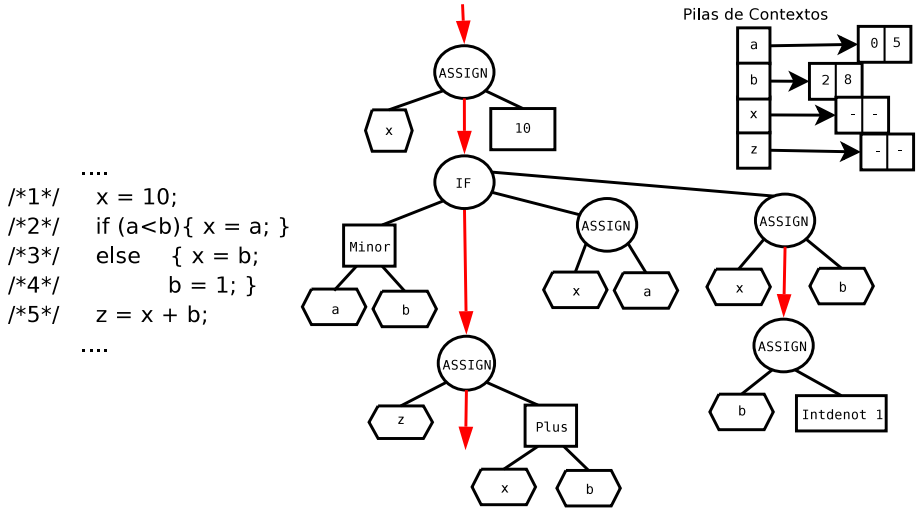

Figura 5.2: Fragmento de Programa Mini y su ASA Correspondiente.

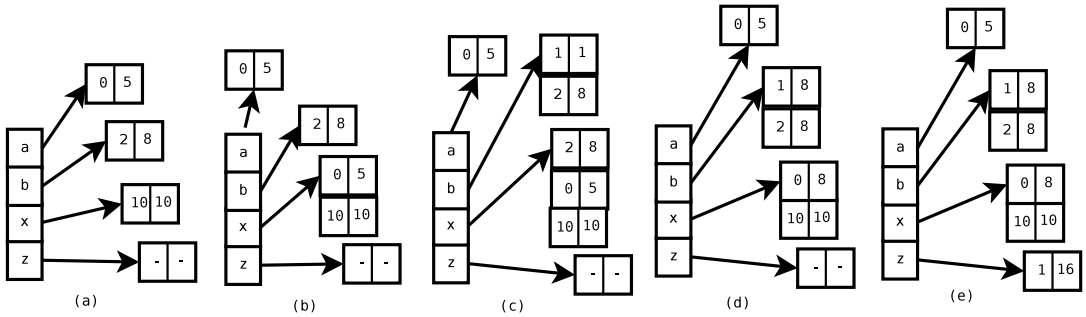

Figura 5.3: Pilas de Contextos.

Al terminar de procesar el condicional se realiza la unión de las cotas de cada variable modificada en cada una de las ramas de esta sentencia. Luego se procesa la asignación de la línea /*5*/. El efecto de ambas actividades se puede apreciar en las figuras 5.3(d) y (e), respectivamente.

5.2. Extensiones del Prototipo

Si bien el prototipo permitió extraer conclusiones interesantes en cuanto a la aplicación de análisis estático y PCC para verificar la seguridad de aplicaciones móviles es necesario que este pueda ser una buena alternativa para desarrollar aplicaciones en el mundo "real". Para lograr esto se debe pensar en un lenguaje "real" y por consiguiente en una política de seguridad que contemple todos los posibles puntos críticos de este lenguaje.

5.2.1. Extensiones del Lenguaje Fuente

El lenguaje se extendió incorporando los tipos básicos **float** (reales) y **char** (caracter). Se sobrecargaron los operadores matématicos y relacionales para estos tipos. Una característica nueva del lenguaje es permitir definir constantes de tipos básicos. Se incorporaron funciones de entrada/salida. **scanf** y **printf** toman datos de la entrada estándar y envían datos a la salida estándar respectivamente. Ambas funciones toman una cadena de formato y los argumentos y retornan un

entero (0 si se produjo un error de entrada/salida, un valor distinto de cero en otro caso). Los perfiles de estas funciones son:

int scanf (string format [, argumentos, ...]);
int printf (string format [, argumentos, ...]);

Donde *string format* es una cadena que describe el formato de cada uno de los argumentos a leer o imprimir. Este formato se describe de manera similar que en C con **%descriptor_de_tipo**. Estos descriptores de tipos son:

n	int
f	float
c	char

Además **scanf** permite, al igual que con los parámetros de las funciones, definir una precondición. Por ejemplo, $scanf(\text{" }\%n(0,5)\text{"}, x)$; significa que el valor de x debe estar acotado por los valores 0 y 5.

Se permite la definición de tipos estructurados (como **struct** de **C**) y arreglos de cualquier tipo.

Se incorporó el tipo puntero para permitir manipulación de memoria dinámica. Para esto se definieron las operaciones para obtener memoria (**malloc**) y liberarla (**free**). Se definió la constante NULL. Y se sobrecargaron los operadores de comparación = y ! =. Los punteros pueden ser de cualquier tipo básico pero su tipo debe ser declarado.

Se definió el operador de *casting* al estilo C. Por ejemplo, si tenemos la variable x de tipo real y queremos asignarle un entero el *casting* se realiza anteponiendo al entero (int). Por ejemplo, $x = (int)4$;.

También se permite declarar variables y constantes globales (al estilo **C**). Se permiten sentencias **repeat**, **for** y **switch**. Y las sentencias **break** y **continue**. Tanto los parámetros como el tipo retornado puede ser de cualquier tipo. Se definieron dos tipos de pasajes de parámetros: por valor y por referencia. Para este último se debe anteponer **ref** a la declaración del parámetro formal. Además, para los parámetros de tipo puntero se puede definir su precondición de forma similar a los parámetros enteros. Estas precondiciones expresan que el puntero es **NULL**, no es **NULL** o no esta inicializado. Por ejemplo, $intf(int * p(NOT\_\_NULL))$ establece que el puntero p debe estar inicializado y apuntar a una dirección válida. Por último, el lenguaje se extendió para que permita llamadas a funciones.

Gramática del Lenguaje Extendido en la figura 5.4 se muestra la sintaxis del lenguaje Mini extendido utilizando una gramática (usando la notación BNF [4]).

5.2.2. Extensiones del Código Intermedio

La extensión del código intermedio tiene como fin soportar las extensiones del lenguaje fuente. Cada sentencia, tipo o función que extiende Mini debe poder ser representada en el árbol sintáctico abstracto. La extensión del ASA es necesaria (y se podría decir obligatoria) pero es una actividad simple y sin mayores complicaciones. Ya que, por cada una de las extensiones en Mini solo hay que definir un nuevo nodo. Y estos nodos son similares a los nodos definidos previamente. A modo de ejemplo, se puede mencionar que es trivial que el ASA que representa una sentencia **repeat** o **for** es muy parecido al de la sentencia **while**.

5.2.3. Extensiones de la Política de Seguridad

La política de seguridad original del prototipo garantiza que no se leen variables no inicializadas y que no se accede a posiciones no definidas sobre los arreglos. Esta política era acorde al lenguaje fuente que se había considerado. Al extender dicho lenguaje surgen nuevos puntos críticos que deben ser verificados. Especialmente con respecto al uso seguro de memoria dinámica. El lenguaje sólo

Program	::=	ConstDecl StructDecl FuncDecl
FuncDecl	::=	TypeSpecifier **Ident** **(** ParamDecl **)** Compound | FuncDecl FuncDecl
TypeSpecifier	::=	BasicType | StructType
BasicType	::=	**int** | **boolean** | **char** | **float**
StructType	::=	TypeSpecifier [**intdenot**] | BasicType ***** | **Ident**
ConstDecl	::=	ConstDecl ConstDecl | **Ident** = Exp;
StructDecl	::=	StructDecl StructDecl | **typedef struct Ident {** VariableDecl **};**
TypeParam	::=	λ | **ref**
ParamDecl	::=	TypeParam TypeSpecifier **Ident** | TypeParam **int Ident (intdenot, intdenot)** | TypeParam ***** TypeSpecifier **Ident (** PointerPrec **)** | ParamDecl **,** ParamDecl
PointerPrec	::=	**is_null** | **not_null** | **not_def**
Compound	::=	**{ }** | **{** Statement **}** | **{** VariableDecl Statement **}**
VariableDecl	::=	VariableDecl VariableDecl | TypeSpecifier ListIdent ;
ListIdent	::=	**Ident** | **Ident =** Exp | ListIdent , ListIdent
Statement	::=	Statement Statement | Compound | **if (** BExp **)** Statement | **if (** BExp **)** Statement **else** Statement | **while (** BExp **)** Statement | **repeat** Statement **until** BExp | **for (int Ident =** IExp ; Exp ; Exp **)** Statement | Assign ; | **return** Exp ; | **break** ; | **continue** ; | **Ident (** ListIdent **)** ; | **printf** ; | **scanf** ; | **free (ident)** ; | ;
Assign	::=	**Ident =** Exp | **Ident** [IExp] **=** Exp
Exp	::=	IExp | BExp | PExp
IExp	::=	**Ident** | **intdenot** | **floatdenot** | **chardenot** | **Ident** [IExp] | IExp **op** IExp | **(** IExp **)**
BExp	::=	**Ident** | **true** | **false** | BExp **bop** BExp | **not** BExp | Exp **rop** Exp | **(** BExp **)**
PExp	::=	**NULL** | **(**TypeSpecifier***) malloc (intdenot);**

Donde **Ident** son los Identificadores; **intdenot** son los números enteros; **op** son los operadores +, -, * y /; **bop** son los operadores || y &&, y **rop** los operadores ==, >, <, >=, <=, $y!$ =; λ es la cadena vacía.

Figura 5.4: Gramática del Lenguaje Mini.

considera punteros de tipos básicos por lo cual no se pueden construir estructuras arbitrarias. Sin embargo, los principales problemas con punteros se encuentran presentes en el lenguaje definido.

Los puntos críticos a considerar son:

- acceso a punteros no inicializados y punteros nulos.
- acceso a memoria no referenciada.

Además, se considera como puntos crítico la división por cero que, si bien estaba presente, no era tenido en cuenta en la política original.

Punteros No Inicializados y Punteros NULL En este punto se busca asegurar que cualquier referencia a un puntero solo será ejecutada cuando esté inicializada y no contenga una referencia a **NULL**. El análisis para detectar punteros no inicializados es el mismo que el presentado para detectar variables enteras (variables de cualquier tipo) no inicializadas. Con la salvedad de que hay que considerar la sentencia free, la cual, tiene como efecto definir como no inicializado el puntero al cual se aplica.

En cuanto al análisis para detectar punteros nulos está dividido en dos pasos:

1. Identificar los puntos en donde se inicializan punteros en **NULL**.
2. Determinar las dependencias entre variables. En el apartado *Análisis de Dependencias* se presenta el análisis para detectar dichas dependencias.
3. Determinar que no exista ninguna referencia a un puntero que tenga como posible valor NULL.

Tanto en el caso de punteros no inicializados como en el de referencias a punteros NULL se insertarán verificaciones dinámicas en aquellos puntos donde no se puede demostrar fehacientemente si es seguro o no la referencia al puntero en cuestión. Este paso es similar al realizado con respecto al acceso a arreglos.

Acceso a Memoria no Referenciada Cuando se utilizan punteros puede surgir el problema de acceder a memoria no referenciada. Esto se puede dar porque se accede a un puntero no inicializado, a un puntero nulo o porque se accede a un puntero que referencia a una posición no inicializada.

Esto último, es consecuencia de la posibilidad de que existan *alias*. Es decir, dos o más variables distintas apuntan a la misma dirección (son *alias*). Esto puede ocasionar errores en programas si existen *alias* no considerados entre parámetros, valores de retorno, variables globales y variables locales. Análisis de *alias* consiste en identificar aquellas variables que apuntan a la misma dirección de memoria. Con esto se puede evitar que se libere una dirección de memoria apuntada por más de una variable.

Para determinar la existencia de *alias* se utiliza el análisis de dependencias del apartado *Análisis de Dependencias* que se encuentra a continuación. Una vez detectadas las dependencias se establece que no se libere memoria que se encuentra referenciada por más de un puntero. Siguiendo el mismo criterio tomado en las demás verificaciones se insertarán verificaciones dinámicas en aquellos puntos de seguridad incierta.

Inserción de Verificaciones Dinámicas por División por Cero En aquellos casos en los cuales la cota determinada para el divisor no contenga al cero no se inserta la verificación dinámica correspondiente. Esto se debe a que la misma sería innecesaria. Pero en aquellos casos en los cuales la cota del divisor contenga al cero o dicha cota no pueda ser determinada por CCMini se inserta la verificación dinámica. Con esto se elimina un error en tiempo de ejecución que no se había contemplado originalmente. Este proceso no requiere casi ningún costo adicional ni la incorporación de ningún análisis nuevo en CCMini. La información requerida es proporcionada por el análisis de cotas existente. Solo se requiere generar una nueva etiqueta que determine si es seguro o no efectuar una división de acuerdo a la información de cotas existente.

Análisis de Dependencias

En esta sección se presenta el algoritmo para generar las dependencias entre variables. Este algoritmo detecta las asignaciones relevantes de las variables. Es decir, el conjunto de variables que determinan el valor de cada variable en un punto determinado del programa. También determina los puntos del programa en que ocurren dichas asignaciones.

Este algoritmo está basado en las *pilas de contextos* introducidas en la sección 5.1. Solo se extendió la información de cada contexto para que referencie a los contextos de las variables de las cuales depende el contexto actual.

El algoritmo de la figura 5.5 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. El algoritmo presentado solo contempla el análisis de dependencias para punteros. El tratamiento para variables de otros tipos es similar.

5.2.4. Extensiones de la Efectividad del Entorno

Extender la efectividad del entorno consiste en aumentar la cantidad de casos en los cuales se determina con certeza la seguridad del código. Si bien el prototipo respondió satisfactoriamente con los casos de prueba con estas extensiones se pretende determinar el estado del programa en una cantidad mayor de casos. Para esto se busca incrementar el análisis de variables inductivas, propagar condiciones, discriminar entre verificaciones dinámicas por límite inferior y superior en los arreglos e insertar verificaciones dinámicas por división por cero.

Incremento del Análisis de Variables Inductivas El prototipo original determina las variables inductivas lineales, es decir, aquellas que se incrementan (o decrementan) en una constante. Con esto solo se consideran aquellos casos en que las variables son de la forma:

$$variable = variable + constante.$$

1. Al comenzar todas las dependencias están indefinidas. **nivel** es igual a cero. **variables** está vacío.
2. Recorrer todo el árbol:

   a) En cada **return ptro**, determinar como postcondición si puede o no ser NULL o puede no estar inicializado **return ptro**.

   b) En cada asignación **ptro1 = ptro2**:
   Si **nivel** es distinto de cero y **ptro1** no está en **variables[nivel]**: insertarla.
   Insertar un elemento en la pila de **ptro1** que referencie al contexto actual de ptro2. Con esto obtenemos la lista de variables de las cuales depende el valor de **ptro1** en este punto del programa.
   Asociar los nodos del ASA que contienen a **ptro1** y a **ptro2** con sus contextos actuales.

   c) En cada condicional **IF cond then else**:

      1) Analizar la expresión **cond** buscando referencias a punteros (para asociar los nodos del ASA que contienen dichos punteros a su contexto corriente).
      2) Incrementar **nivel** en uno. Analizar el cuerpo del **then**.
      3) Incrementar **nivel** en uno. Analizar el cuerpo del **else**.
      4) Por cada variable **ptro** que este en **variables[nivel]** o en **variables[nivel-1]**:
      Insertarle a **ptro** un elemento nuevo en su pila que referencie los dos contextos anteriores (los elementos que eran tope y subtope). Luego eliminar **x** de **variables[nivel]** y de **variables[nivel-1]** según corresponda.
      5) Decrementar nivel en dos.

   d) En cada iteración **WHILE cond cuerpo**:

      1) Analizar la expresión **cond** buscando referencias a punteros (para asociar los nodos del ASA que contienen dichos punteros a su contexto corriente).
      2) Incrementar **nivel** en uno. Analizar el cuerpo del ciclo, **cuerpo**.
      3) Por cada variable **x** que este en **variables[nivel]**: Insertarle a **x** un elemento nuevo en su pila que referencie el contexto anterior (el elemento que era tope). Luego eliminar **x** de **variables[nivel]**.
      4) Decrementar nivel en uno.

Figura 5.5: Algoritmo para Generar las Dependencias de Variables

Donde *constante* esta acotada por el rango (min, max) con $min = max$.

Para incrementar la cantidad de casos se incluyó en el algoritmo que determina las variables inductivas análisis que consideran incrementos en rangos (constantes en cada ciclo), condiciones de ciclo compuestas y variables co-inductivas.

Considerar incrementos en rangos consiste simplemente en extender el algoritmo que determina si una variable es inductiva para que considere expresiones de la forma

$$variable = variable + expresion.$$

Donde *expresion* esta acotada por el rango (min, max) con $min \leq max$. Notar que en este punto la cota de *expresion* debe ser constante dentro del ciclo, es decir, no esta determinada por el número de iteraciones. Por supuesto, luego hay que extender el análisis que se realiza para determinar la cantidad de iteraciones que realiza el ciclo. Ya que hay que determinar cual es la mayor cantidad de iteraciones posibles. Para esto se considera como posibles valores iniciales min y max para realizar con ambos valores el mismo análisis que se realizaba en el prototipo original. El resultado es aquel que determina una mayor cantidad de iteraciones.

Originalmente solo se contemplaron condiciones simples de los ciclos. Para determinar la cantidad máxima de iteraciones que realizará un ciclo se tuvo en cuenta variables inductivas y condiciones simples de la forma

$$var_{inductiva}\ op_{relacional}\ expresion.$$

Esto se extendió para considerar condiciones compuestas de la forma

$$(var_{inductiva}\ op_{relacional}\ expresion) op_{logico} (var'_{inductiva}\ op_{relacional}\ expresion).$$

Es decir, condiciones simples con una variable inductiva relacionadas por && (and) o || (or). En estos casos se realiza el análisis mencionado anteriormente para cada condición simple y luego dependiendo del operador lógico se determina el número de iteraciones.Si el operador es && se toma el número mínimo de iteraciones, en cambio, si el operador es || se toma el número máximo de iteraciones.

Variables co-inductivas son aquellas variables cuyo valor esta determinado por variables inductivas. La variable k es una variable co-inductiva en un ciclo C si:

1. Hay solo una definición de k en C, de la forma $k = j + d$ o $k = j * d$ donde j es una variable inductiva y d es constante en el ciclo.

2. Si j es una variable co-inductiva de la familia de i entonces existe una sola definición de j que define a k y no existe ninguna definición de i entre j y k.

Por ejemplo, si i es una variable inductiva y $i = i + 1$ y k es una variable co-inductiva de la familia de i, tal que, $k = i * 4$, entonces se puede afirmar que k esta acotada por $(i_min * 4, i_max * 4)$. Ahora, si además existe $j = k + 2$ entonces j también es una variable co-inductiva de la familia de i. Y j está acota por $(i_min * 4 + 2, i_max * 4 + 2)$.

En la figura 5.6 se pueden apreciar ejemplos de variables inductivas, co-inductivas y variables que no lo son. Por supuesto, se debe considerar que las variables solo son modificadas en los puntos que se muestran. La variable `ind` es inductiva, solo se modifica en un punto del ciclo y es incrementada por la constante `1`. `co_ind` es una variable co-inductiva ya que su valor depende de una variable inductiva. Ocurre una situación similar con `co_ind_2` ya que su valor depende de una variable co-inductiva. En cambio, la variable `no_ind` no es inductiva ya que su valor depende de una variable (`aux`) para la cual sus cotas no son necesariamente constantes.

```
...
while (index<10) {
    ...
    ind = ind + 1;
    ...
    co_ind = ind * 2;
    ...
    co_ind_2 = co_ind + 1;
    ...
    no_ind = aux + 1;
    ...
}
...
```

Figura 5.6: Ejemplo de Variables inductivas y Co-inductivas.

Propagación de Condiciones Propagando condiciones de sentencias condicionales e iterativas se busca saber con mayor exactitud el estado de los programas en cada punto de ejecución. Especialmente en cada rama de ejecución posible. Básicamente este proceso consiste en acotar las variables que intervienen en alguna condición en concordancia con el predicado del cual forman parte.

Las reglas son simples, para una condición *expresion* que contiene a la variable x:

1. Si *expresion* es de la forma $x < exp$, la cota máxima de x es la cota máxima de exp menos 1.

2. Si *expresion* es de la forma $x > exp$, la cota mínima de x es la cota mínima de exp más 1.

3. Si *expresion* es de la forma $x \leq exp$, la cota máxima de x es la cota máxima de exp.

4. Si *expresion* es de la forma $x \geq exp$, la cota mínima de x es la cota mínima de exp.

5. Si *expresion* es de la forma exp_1 && exp_2, con x en exp_1 y exp_2, se intersecan las cotas dadas por cada uno de los conyunjendos.

6. Si *expresion* es de la forma exp_1 || exp_2, con x en exp_1 y exp_2, se unen las cotas dadas por cada uno de los conyunjendos.

Intersecar cotas consiste simplemente en realizar la intersección de los conjuntos de valores definidos por cada una de las cotas que se quieren intersecar. En el caso de la unión de cotas se realiza la unión de dichos conjuntos.

Cabe resaltar que las modificaciones en las cotas de las variables que intervienen en las condiciones son efectivas cuando se comienza a analizar el cuerpo de la sentencia a la cual pertenece la condición. No se modifican durante el análisis de la condición porque puede modificar las cotas de otras variables que intervengan.

Para incorporar este análisis al algoritmo de la figura 5.1 no es necesario ningún cambio. Solo hay que definir como *analizar condiciones para determinar cotas* al algoritmo mencionado anteriormente.

En este punto también se identifican aquellas condiciones de ciclos y condicionales que siempre se cumplen. Con esto se determina exactamente el valor de las variables al terminar la sentencia involucrada. En este caso se debe modificar el algoritmo de la figura 5.1 para que determine como cotas de las variables involucradas a aquellas determinadas por la rama que siempre se ejecuta. Eliminando en estos casos, solo cuando se cumple siempre una condición, la unión de los contextos anteriores. El análisis requerido consiste en verificar si las cotas de las variables que intervienen en la condición están dentro de los valores especificados. Las reglas son simples, para una condición *expresion* que contiene a la variable x la condición siempre se cumple si:

1. *expresion* es de la forma $x < exp$ y la cota máxima de x es menor que la cota máxima de *exp*.

2. *expresion* es de la forma $x > exp$ y la cota mínima de x es mayor que la cota mínima de *exp*.

3. *expresion* es de la forma $x \leq exp$ y la cota máxima de x es menor o igual que la cota máxima de *exp*.

4. *expresion* es de la forma $x \geq exp$ y la cota mínima de x es mayor o igual que la cota mínima de *exp*.

5. *expresion* es de la forma exp_1 && exp_2 y se cumplen ambos conyunjendos.

6. *expresion* es de la forma exp_1 || exp_2 y se cumple al menos uno de los conyunjendos.

Cabe resaltar que se están analizando las reglas y una posible optimización en el caso de las ramas del **else**.

Discriminación de Verificaciones Dinámicas El prototipo original solo contempla verificaciones dinámicas de límites de arreglos por ambos extremos. Es decir, si se inserta una verificación dinámica esta siempre verificará que el índice no sea menor al límite inferior y que no sea mayor al límite superior del arreglo en cuestión.

La información generada en el punto anterior (propagación de condiciones) puede ser utilizada, en ciertos casos, para eliminar verificaciones dinámicas. Solo se mantendrá la verificación dinámica para aquel límite que no se tiene información. También se puede dar el caso que con esta información se detecten violaciones que antes no eran detectadas. Al ser mayor la cantidad de información disponible es mayor la certeza con que se puede decidir.

En este punto hay que mencionar que es necesario definir nuevos nodos para el ASA. Estos nodos deben permitir expresar si es totalmente seguro el acceso al arreglo, totalmente inseguro, es seguro por el límite inferior o solamente es seguro el acceso por el límite superior. Es decir, se deben definir los nodos UNSAFE_ARRAY_INF y UNSAFE_ARRAY_SUP para representar verificaciones dinámicas por el límite inferior y el límite superior respectivamente.

Capítulo 6

Conclusiones y Trabajos Futuros

En este capítulo se resumen las principales contribuciones y conclusiones de este trabajo como así también se presentan los trabajos futuros.

6.1. Contribuciones

CCMini. Se diseñó y desarrolló un prototipo de compilador certificante basado en análisis estático de flujo de control y de datos. Este compilador permite certificar propiedades de seguridad de gran importancia. Por ejemplo, certifica la ausencia de accesos inválidos a los arreglos. CCMini toma como entrada un subconjunto extendido de C y genera un *árbol sintáctico abstracto* anotado con información del estado del programa en cada punto.

CCMini 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 dos 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.

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.

Este prototipo, CCMini, 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 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. Además la generación de la certificación también es lineal con respecto al tamaño de los programas. No hay que dejar de notar que estas características son fundamentales en un ambiente de código móvil.

Una ventaja importante de CCMini sobre otros compiladores certificantes, tal como Touchstone, reside en que la complejidad temporal de los algoritmos de CCMini que generan la certificación son lineales con respecto al tamaño de los programas. Si se lo compara con javac, CCMini 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.

Las extensiones presentadas permiten especular que el prototipo puede ser usado para desarrollar aplicaciones en el mundo "real". Se obtuvo el diseño de un prototipo más eficiente y efectivo que permite certificar propiedades de gran relevancia para la seguridad. Se debe resaltar la inclusión de punteros con la certificación de ausencias de referencias a NULL y de *aliasing*. Además, se comprobó que análisis estático de flujo de control y de datos es una técnica muy interesante para verificar propiedades de seguridad. Estos análisis balancean el costo entre el esfuerzo de diseño y la complejidad requerida.

Cabe mencionar que dichas optimizaciones y extensiones se están implementando con el fin de obtener una nueva versión de CCMini.

PCC-SA. Se desarrolló un framework para la ejecución segura de código móvil denominado *Proof-Carrying Code based-on Static Analysis* (PCC-SA) [62]. Este framework, está basado en PCC y técnicas de análisis estático de flujo de control y de datos. El framework posee las mismas características que CCMini. Además el consumidor de código cuenta con una infraestructura pequeña, confiable y automática con la cual verifica el código estáticamente.

Una de las características más importantes del framework 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. Además, y no menos importante, el proceso de verificación del esquema de prueba tiene complejidad temporal lineal con respecto al tamaño de los programas de entrada.

La arquitectura de PCC-SA es similar a la de PCC. Tanto en PCC-SA como en PCC la infraestructura confiable es pequeña y automática. Las pruebas PCC son realizadas sobre un lenguaje *assembly* tipado mientras que en PCC-SA son realizadas sobre un código intermedio de más alto nivel (un ASA anotado). PCC-SA mejora uno de los principales problemas de PCC: el tamaño de las pruebas. En la mayoría de los casos el tamaño de las pruebas PCC son exponenciales respecto a la longitud del código, mientras que en PCC-SA el tamaño de los esquemas de prueba son lineales respecto a la longitud del código. La desventaja es que el consumidor de código debe reconstruir las pruebas para los puntos críticos, no sucede lo mismo en el caso de PCC.

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.

Resumiendo lo anterior, se puede afirmar que la experimentación con el prototipo de PCC-SA permitió corroborar que el framework posee las características enunciadas en la sección 2.2.2:

1. El tamaño de las pruebas es lineal al programa producido. En la mayoría de casos, el tamaño de las pruebas es menor al tamaño de los programas.

2. Permite eliminar una mayor cantidad de chequeos en tiempo de ejecución que en PCC.

3. Manipular una representación equivalente al código fuente permite independencia de plataforma.

4. La infraestructura del consumidor es automática y de bajo riesgo. Por lo cual, la base confiable de consumidor es reducida.

5. El esfuerzo reside en el productor: el consumidor de código solamente tiene que ejecutar un proceso de verificación de pruebas rápido y simple.

6. No se requieren relaciones productor/consumidor confiables.

7. Es flexible, dado que puede utilizarse con un amplio rango de lenguajes y de políticas de seguridad.

8. No sólo se puede usar para seguridad sino también para realizar certificaciones sobre la funcionalidad del código.

9. El código se verifica estáticamente.

10. Permite detectar cualquier modificación (accidental o maliciosa) del programa y/o su prueba, mientras que sigue garantizando la seguridad.

11. Permite ampliar el rango de las propiedades de seguridad que pueden ser demostradas automáticamente.

12. Permite realizar diversas optimizaciones al código generado.

13. Puede usarse en combinación con otras técnicas.

La experimentación también permitió corroborar que el framework posee las siguientes desventajas:

1. 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.

5. Traducir una propiedad de seguridad al análisis estático que la verificará no es trivial.

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.

Experiencias. Un estudio utilizando un pequeño conjunto de programas mostró que la mayoría de los accesos a arreglos utilizados es mediante variables inductivas. Este resultado sugiere que el enfoque propuesto resuelve la seguridad de accesos a arreglos en la mayoría de las situaciones sin utilizar chequeos en tiempo de ejecución. También, el estudio de los resultados obtenidos usando estos programas permitió determinar que en el peor de los casos, el tamaño de la prueba es cercano al 25 % del tamaño total del programa y prueba. Se pudo determinar que el crecimiento del tamaño de la prueba es lineal con respecto al tamaño de los programas utilizados.

Sin embargo, a fin de obtener una muestra significativa, habría que contar con bibliotecas numerosas, cosa que sólo puede lograrse utilizando un lenguaje estándar en vez del actual. En este sentido, sería muy útil contar con una estadística, computada sobre una muestra real y numerosa de programas, de los resultados de aplicar las técnicas de análisis estático usadas para garantizar la seguridad de acceso a arreglos. Por estos motivos se esta estudiando la posibilidad de utilizar las bibliotecas de C. En este punto no hay que dejar de mencionar la complejidad de esto por la gran cantidad de variantes que tiene C para acceder a un arreglo.

Sin embargo, la observación informal de algunas aplicaciones escritas en el lenguaje C (`Mozilla`, `gcc`, `AbiWord` y el `kernel` de `linux`) permitió extraer información que permitió corroborar las afirmaciones realizadas en este trabajo. Además, la observación de estos programas generó cierta evidencia de que los programas que se encuentran en la práctica pertenecen a la familia (caracterizada en la sección 4.5) para los cuales el proceso de certificación de código realizado por CCMini es lineal respecto de la longitud de los programas fuente.

6.2. Trabajos Futuros

Se está trabajando en la extensión del prototipo 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 este 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 varios sub-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.

También 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.

Sería interesante introducir análisis de punto fijo para acotar los ciclos y analizar el costo y la eficiencia de utilizar los resultados de este análisis. Con este tipo de análisis se lograría acotar la mayoría de (en muchos casos todos) los ciclos. Para reducir el costo del consumidor se podría incluir un candidato de punto fijo en el esquema de prueba. La verificación de que un candidato es realmente un punto fijo es un proceso que se puede realizar en una "sola pasada".

Consideramos de mucho interés indagar en tres direcciones: análisis abstracto, lenguajes ensambladores tipados 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 ensamblador tipado 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 [73] 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.

Apéndice A

Proof-Carrying Code (PCC)

El surgimiento de *Proof-Carrying Code* (PCC) está ligado a los problemas que genera la utilización del código móvil. PCC nace, entonces, como un enfoque alternativo a las soluciones tradicionales, planteadas para tratar dichos inconvenientes, y puede clasificarse dentro de las técnicas englobadas en *seguridad basada en lenguajes*.

El propósito de PCC es permitir a un consumidor de código determinar fehacientemente si un programa se comportará de manera segura antes de ejecutarlo. Esto requiere que el productor de código provea la evidencia de que la ejecución del programa no violará las normas de seguridad impuestas. Esta evidencia debe verificarse eficientemente y, como el nombre de la técnica lo implica, consistirá de una prueba formal de que el programa cumple con la especificación de seguridad.

A.1. La Arquitectura de Proof-Carrying Code

Proof-Carrying Code (PCC) es una propuesta para garantizar código móvil seguro, introducida por Necula y Lee [54, 52], que ha generado una activa línea de investigación con una importante producción [55, 21, 7, 8, 9, 69, 60, 61, 66, 17], que aún presenta muchos problemas abiertos. *PCC* requiere tanto la construcción de pruebas matemáticas de propiedades de programas que sean eficientemente verificables como de la especificación formal de propiedades de seguridad.

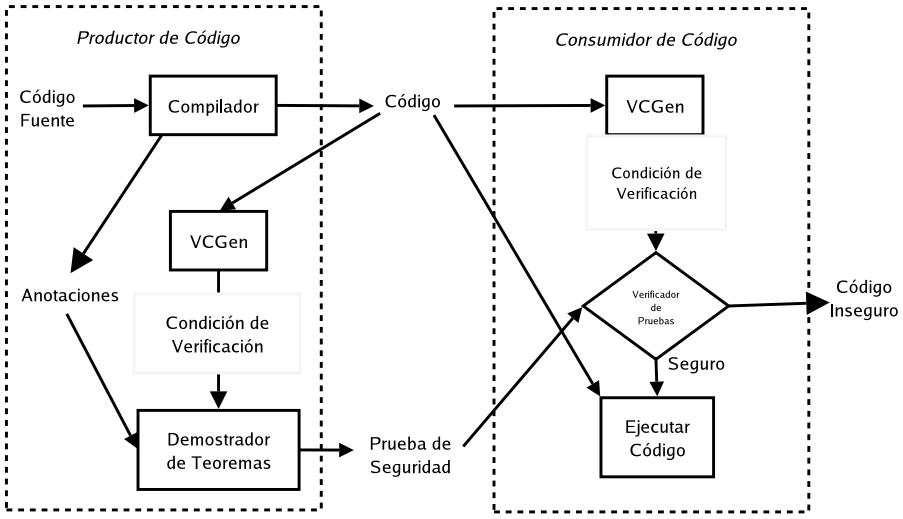

Figura A.1: Visión Global de una Arquitectura de Proof-Carrying Code.

La técnica PCC, cuya arquitectura general se muestra en la Figura A.1, exige a un productor

de software proveer su programa conjuntamente con la prueba formal de su seguridad. La política de seguridad del destinatario se formaliza mediante un sistema de axiomas y reglas de inferencia, sobre el cual debe basarse la demostración construida por el productor. El consumidor, por su parte, verifica que la prueba sea válida y que el código recibido corresponda a la demostración, y sólo ejecuta el código en caso de que ambas respuestas sean positivas.

PCC no requiere autenticación del productor, ya que el programa sólo correrá si localmente se ha demostrado su seguridad, minimizando así la cantidad de confiabilidad externa requerida. Tampoco se requiere verificación dinámica, con lo cual no se introduce ninguna merma en el tiempo de ejecución, ya que la prueba se efectúa con antelación. Este enfoque estático de PCC es una ventaja respecto a los enfoques dinámicos que controlan la seguridad operación a operación mientras el código se ejecuta, tales como el implementado por Java, uno de los primeros intentos para garantizar seguridad en código móvil.

El proceso de creación y uso de PCC está centrado alrededor de la política de seguridad, la cual es definida y hecha pública por el consumidor de código. A través de esta política, el consumidor de código especifica precisamente bajo qué condiciones considera que es segura la ejecución de un programa remoto.

La política de seguridad consiste en dos componentes principales: las reglas de seguridad y la interfaz. Las reglas de seguridad describen todas las operaciones autorizadas y sus precondiciones de seguridad asociadas. La interfaz describe las convenciones de llamada entre el consumidor de código y el programa remoto, éstas son las invariantes que mantiene el consumidor cuando invoca el código remoto y las invariantes que el código remoto debe establecer antes de llamar a las funciones proporcionadas por el consumidor. En analogía con la teoría de tipos, las reglas de seguridad son las reglas de tipado y la interface es el perfil (signatura) que el módulo remoto debe implementar.

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

1. Certificación, el productor de código compila (o ensambla) 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, que junto con los componentes del código nativo conforman el PCC binario. El productor de código puede almacenar el PCC binario 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 parte de la prueba de un PCC binario dado y carga el componente del código nativo para la ejecución. La validación y aprobación es rápida y dirigida por un algoritmo seguro. El consumidor debe confiar solamente en la implementación de este algoritmo simple, 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 en código máquina, posiblemente, muchas veces. Esta fase puede proceder sin que se realicen chequeos en tiempo de ejecución adicionales porque la etapa de validación anterior asegura que el código obedece a la política de seguridad.

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, consume mucho tiempo o requiere interacción con el usuario.

De todo lo anterior puede verse que una implementación típica de PCC requiere poseer los siguientes cinco elementos:

1. Lenguaje de especificación formal usado para expresar la política de seguridad,

2. Semántica formal del lenguaje usado para el código no confiable,

3. Lenguaje usado para expresar la prueba,
4. Método para generar la prueba de seguridad,
5. Algoritmo para validar la prueba.

A.1.1. Descripción de los Componentes del Entorno

En la figura A.1 el módulo que contiene al compilador certificante, el generador de condiciones de verificación (**VCGen**, Verification Condition Generator) y el demostrador de teoremas, permite generar código seguro para un sistema PCC. El *compilador certificante* genera el código intermedio (usualmente como *bytecode* o en un lenguaje assembly tipado) con sus respectivas anotaciones. Esta salida es procesada por el **VCGen** que produce un predicado de seguridad para cada función en el código, de manera tal que dichos predicados puedan ser demostrados si y sólo si el programa en lenguaje intermedio es seguro de acuerdo a su especificación tipada. Gracias a las anotaciones del código y a las especificaciones tipadas, el **VCGen** puede ser implementado como un programa de una sola pasada (*single pass*). Luego, los predicados de seguridad forman la condición de verificación (VC, por sus siglas en inglés), la cual se suministra a un demostrador de teoremas, que produce una prueba formal para el código basada en la política de seguridad provista. Finalmente, los tres (la condición de verificación, la prueba y el código ejecutable) son provistos al consumidor.

El Compilador Certificante

Un compilador certificante toma como entrada el código fuente y obtiene como resultado el código, las anotaciones y las especificaciones tipadas. Usualmente las únicas anotaciones de código que deben ser provistas por el productor son las anotaciones de invariantes de ciclos y las especificaciones tipadas de las funciones.

La especificación tipada de una función es un par: precondición y postcondición. La *precondición* es una descripción del tipo de cada registro argumento usado por la función, mientras que la *postcondición* es una descripción similar de los tipos de los registros que se obtienen como resultado de la invocación a dicha función.

Existen varias ventajas en utilizar compiladores certificantes en lugar de un compilador tradicional: las certificaciones realizadas por dicho compilador pueden ser usadas para generar optimizaciones sobre el código, mientras que las verificaciones realizadas sobre estas certificaciones son de gran utilidad para depurar modificaciones y extensiones del compilador.

El Generador de Condición de Verificación (VCGen)

El **VCGen** es un analizador simbólico para lenguajes intermedios, por ejemplo, lenguajes assembly tipados. Su función es examinar el código en busca de operaciones potencialmente inseguras y emitir un predicado de seguridad que debe ser verificado por cada una de dichas operaciones. Estos predicados conforman la condición de verificación (VC). Por ejemplo, para una política de seguridad que busca garantizar seguridad de tipos y de memoria, siempre que el **VCGen** encuentra una operación de memoria emite un predicado que establece bajo qué condiciones la operación de memoria es considerada segura.

El Demostrador de Teoremas

Para demostrar formalmente los predicados de seguridad producidos por el **VCGen** se necesita un *demostrador de teoremas* que genere la prueba o demostración. En la actualidad existen muchos demostradores de teoremas (se puede mencionar a Coq e Isabelle) [19, 23, 27, 64] que pueden ser utilizados para este propósito, pero ninguno produce aún pruebas que puedan ser verificadas automáticamente. Estos demostradores de teoremas son sistemas lógicamente incompletos que requieren la interacción humana en muchas instancias, mientras que los sistemas requeridos para que la técnica de PCC pueda aplicarse industrialmente deben ser completamente automáticos. Para poder utilizarlos en arquitecturas PCC los demostradores no sólo deben demostrar la condición de

verificación, sino también generar el esquema de la prueba, el cual será enviado al consumidor de código.

El Verificador de Pruebas

El rol del *verificador de pruebas* es doble: verificar que cada paso del esquema de la prueba es válido y además, que la prueba provista demuestra la condición de verificación requerida y no otro predicado. Por ejemplo, el verificador de pruebas puede ser implementado usando un cálculo lambda tipado sobre un *Logical Framework* (LF), tal como se propone en [55, 65]. Para esto, se codifican las pruebas como expresiones LF y el predicado de seguridad como un tipo LF. Entonces la validación de la prueba consistirá simplemente en verificar el tipo de la expresión que la representa.

A.1.2. Los primeros trabajos relativos a PCC

Una de las primeras implementaciones de PCC [53, 52] se realizo con el fin de garantizar seguridad en filtros de paquetes de red (*network packet filter*). En este caso se usó un prototipo especial de assembly basado en el assembly DEC Alpha. Las verificaciones de seguridad se realizan sobre el *Edinburgh Logical Framework* (LF), el cual es básicamente un cálculo lambda tipado, garantizando seguridad de memoria. En este experimento solo se requiere de 1 a 3 milisegundos para realizar la verificación de los paquetes recibidos.

G. Necula y P. Lee definen un *logical framework* derivado del *Edinburgh Logical Framework* (LF) [56, 57] que puede ser usado para obtener representaciones compactas de las pruebas y verificadores de pruebas eficientes. Este framework es denominado LFi, el cual hereda de LF la capacidad de codificar varias lógicas de manera natural. Además, el LFi permite representar pruebas sin la redundancia característica de LF.

Otro aporte importante de G. Necula y P. Lee [55, 58] es el concepto de Compilador Certificante. Estos compiladores, cuya nueva característica es que contienen un certificador (un probador de teoremas) incorporado, no solo compilan el código fuente sino que también verifican que cumpla con determinadas propiedades generando una prueba de seguridad. El primer compilador certificante desarrollado [54], llamado Touchstone, fue desarrollado en el marco de obtener una arquitectura PCC totalmente automática.

A.2. Ventajas y Desventajas de PCC

PCC posee varias características que, en combinación, proporcionan ventajas sobre otros enfoques para la ejecución segura de código remoto. Las ventajas fundamentales que brinda la utilización de PCC son:

1. La infraestructura del consumidor es automática y de bajo riesgo,
2. El esfuerzo reside en el productor: el consumidor de código solamente tiene que ejecutar un proceso de verificación de pruebas rápido y simple,
3. PCC no requiere relaciones productor/consumidor confiables,
4. PCC es flexible: puede utilizarse con un amplio rango de lenguajes y de políticas de seguridad,
5. PCC no solo se puede usar para seguridad,
6. La generación de PCC puede ser automatizada,
7. El código PCC se verifica estáticamente,
8. Los programas PCC permiten detectar cualquier modificación (accidental o maliciosa) del programa y/o su prueba, mientras siguen garantizando la seguridad,
9. PCC puede usarse en combinación con otras técnicas.

La idea de PCC es fácil de comprender, pero su implementación eficiente presenta serias dificultades y su uso algunas desventajas. A continuación se enumeran las principales desventajas que surgen del uso de PCC:

1. La arquitectura 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. La codificaciones triviales de pruebas de propiedades de programas son muy largas,
4. La generación de predicados acerca de propiedades del código es una tarea dificultosa,
5. Probar la condición de verificación es una tarea complicada y costosa,
6. Verificar la prueba no es una tarea fácil si lo que se quiere es una prueba concisa y que el chequeador sea pequeño, rápido e independiente de la política de seguridad,
7. Establecer la política de seguridad es una tarea costosa,
8. Garantizar la correctitud de la arquitectura es un procedimiento muy costoso.

Comparando la lista de beneficios con la de dificultades, se hace clara la estrategia de diseño de PCC: la carga de la seguridad debe estar en el productor y no en el consumidor de código. Esto habilita a PCC para trabajar con una infraestructura pequeña, confiable y automática, por cuanto el "trabajo difícil" es realizado por el productor de código, que se halla en una mejor posición para entender el código o para usar herramientas interactivas para este propósito.

A.3. Avances más Relevantes en PCC

En los últimos años se han logrado importantes avances en el desarrollo de PCC. Estos aportes tienden a buscar soluciones a las desventajas antes mencionadas y a obtener un sistema PCC automático y flexible. A continuación presentamos los nuevos enfoques que consideramos más importantes.

A.3.1. Foundational Proof-Carrying Code

A. Appel [8] señala que la arquitectura PCC original está especializada en un sistema de tipos en particular, denominándola *type-specialized PCC*, dado que en los primeros prototipos de PCC las pruebas fueron construídas sobre una lógica extendida con un sistema de tipos particular. Formalmente podemos decir que los constructores de tipos aparecen como primitivas de la lógica y algunos lemas son parte del sistema de verificación. Además, recalca A. Appel, un sistema PCC está basado en un sistema de tipos y en un lenguaje máquina en particular.

En pos de obtener una arquitectura PCC no especializada en un sistema de tipos en particular A. Appel definió el enfoque denominado *Foundational Proof-Carrying Code* (FPCC) [7, 8], en el cual la arquitectura PCC no depende de un sistema de tipos ni necesita usar un Generador de la Condición de Verificación. De esta manera, los únicos componentes que se deben considerar como confiables son el verificador de pruebas y la política de seguridad. Esta reducción en la infraestructura confiable trae aparejada una reducción en la complejidad y costo de la demostración de corrección de un sistema PCC.

En FPCC la semántica operacional del código de máquina es definida [46] en una lógica de alto orden con unos pocos axiomas aritméticos, la cual es suficientemente expresiva para definir también la política de seguridad [9].

Si bien A. Appel afirma que el FPCC es más flexible y más seguro que el PCC original, ya que se pueden usar distintos sistemas de tipos y la infraestructura confiable es más pequeña, no hay que dejar de mencionar que es más difícil de construir, ya que definir modelos semánticos de sistemas de tipos complejos es una tarea dificultosa. Trabajos posteriores como los de Necula y Schneck [69, 61] se reconoce que FPCC reduce el conjunto de componentes confiables pero señalan que el verificador de pruebas debe asumir, en FPCC, funciones análogas al **VCGen** eliminado.

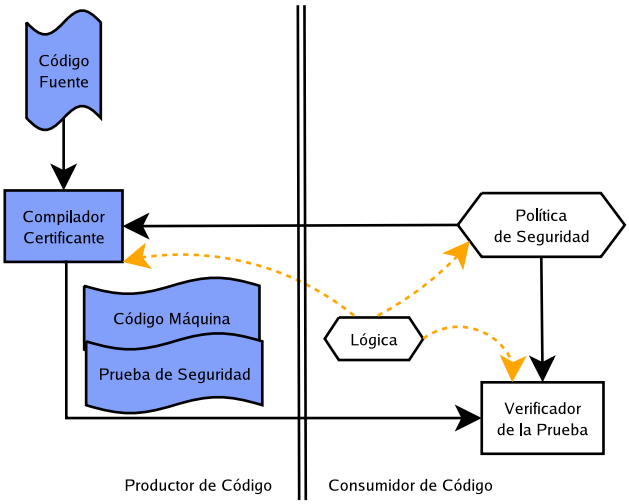

Figura A.2: Visión Global de una Arquitectura de Foundational Proof-Carrying Code.

A.3.2. Syntactic Foundational Proof-Carrying Code

En pos de solucionar las dificultades planteadas por FPCC Hamid et al. [32] presenta un enfoque sintáctico de FPCC (SFPCC) en el cual la prueba fundacional para un programa máquina tipado consiste en la derivación de tipos más la prueba de correctitud sintáctica (del sistema de tipos considerado). Cabe notar que la derivación de tipos puede ser obtenida por un verificador de tipos (*type checker*) mientras que la prueba de correctitud sintáctica es más fácil de construir que la prueba de correctitud semántica.

La implementación existente de SFPCC [32] se realizó en el asistente de pruebas Coq [38]. También se implementó una variante del lenguaje assembly tipado TAL [48] denominada FTAL.

A.3.3. Configurable Proof-Carrying Code

Mientras que el enfoque clásico de PCC [55] y el FPCC [8] constituyen dos extremos en el espectro de diseño de sistemas PCC, Necula y Schneck [60, 61] han propuesto ciertas modificaciones al sistema de PCC clásico a fin de acercarse a la meta ideal planteada en FPCC. Esta alternativa intermedia, denominada *Configurable Proof-Carrying Code* (CPCC), combina la eficiencia y la escalabilidad del enfoque de PCC tradicional con el incremento de confiabilidad y flexibilidad dado por FPCC.

En esencia este enfoque propone que el productor no sólo provea el código y la prueba sino también una gran parte del verificador de pruebas. En este caso el verificador está conformado por un núcleo confiable y local al consumidor, y por una parte enviada por el productor (que al ser foránea no es confiable). El diseño debe garantizar que los resultados del verificador sean de todos modos confiables.

La arquitectura de un sistema CPCC es similar a la de PCC clásico, sólo que el **VCGen** está formado por el **Configurador del VCGen**, provisto por el productor, el **Núcleo del VCGen** y un decodificador de instrucciones. Se puede describir la interacción de los componentes en dos fases: la de configuración y la de verificación. En la fase de *configuración* el productor de código provee el **Configurador del VCGen**. El proceso de verificación del código es similar al de un sistema de PCC clásico. Se debe notar que el **VCGen** es el componente más complejo que interviene en la fase de verificación y que la configuración se realiza una sola vez para cada política de seguridad.

Necula y Schneck afirman que CPCC es más confiable dado que la infraestructura confiable es más pequeña, y es más flexible porque el productor de código puede crear un **VCGen** que explote

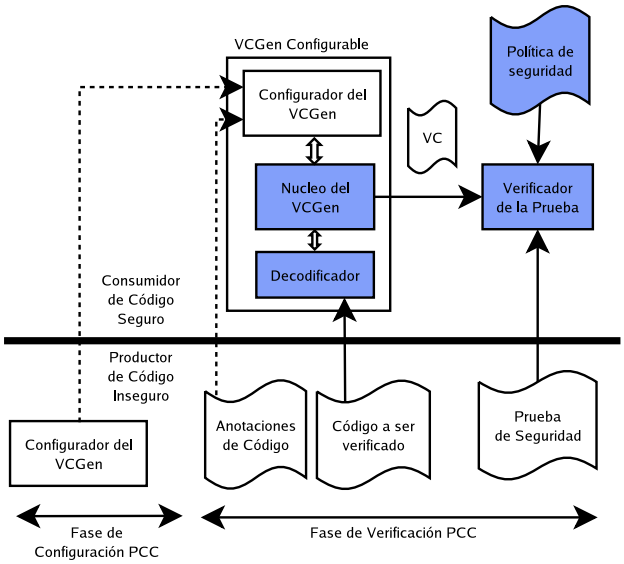

Figura A.3: Visión Global de una Arquitectura de Configurable Proof-Carrying Code.

las estructuras y convenciones particulares del código en pos de generar menos y más simples predicados a probar.

A.3.4. Temporal-Logic Proof-Carrying Code

Este enfoque propone utilizar lógica temporal para especificar las políticas de seguridad de un sistema de PCC. Este enfoque pretende ser "universal", en el sentido de que no necesita ser modificado para cada nueva política de seguridad, siempre y cuando dicha política pueda ser especificada en lógica temporal. Además, se puede remplazar una parte importante del verificador de pruebas con una especificación formal, con el costo de obtener pruebas más largas.

Bernard y Lee [17] argumentan que el uso de lógica temporal permite considerar al **VCGen** como un componente no confiable y separar la política de seguridad de los componentes de verificación, se reduce así la cantidad de software que debe considerarse confiable. En este punto se parece a FPCC, con la diferencia de que el productor y el consumidor de código deben aceptar una noción compartida de seguridad de tipos.

El mecanismo de verificación de *Temporal-Logic PCC* reconoce a un programa como un término formal y la semántica operacional de la máquina del consumidor es representada como un conjunto de reglas de inferencia. La política de seguridad se puede codificar directamente; la prueba de seguridad debe mostrar que al correr el programa con un conjunto de condiciones iniciales, la política de seguridad no es violada. Hay que notar que la política de seguridad es independiente del mecanismo de verificación, pero no se requiere ningún mecanismo adicional para especificarla.

El uso de lógica temporal no es un requerimiento fundamental para este enfoque, dado que, por ejemplo, la lógica temporal puede ser traducida a una lógica de primer orden con parámetros de tiempo explícitos y relaciones de transición de estados pueden imitar operadores temporales por clausura transitiva.

A.3.5. Lazy-Abstraction y Proof-Carrying Code

Henzinger, Necula y otros [34] mostraron que *lazy-abstraction* (un refinamiento de *model-checking*) puede ser usada de manera natural y eficiente para construir pruebas pequeñas para propiedades de seguridad temporal en un sistema de PCC. La generación de las pruebas está entrelazada con el proceso de *model-cheking*: las estructuras de datos producidas por *lazy-abstraction*

suplantan las anotaciones requeridas para la construcción de la prueba y proveen una descomposición de la demostración que permite guiar a un verificador de pruebas pequeño. En particular, usando predicados de abstracción donde la prueba debe ser necesariamente pequeña y usando el algoritmo de *model-checking* para guiar la generación de la prueba, se elimina la necesidad de *backtracking*, por ejemplo, en la prueba de disyunciones. La VC requerida se obtiene automáticamente de la estructura construida por el algoritmo de *lazy-abstraction*. La prueba de la VC se codifica en LF, la codificación es estándar [55], por lo que la verificación de la prueba se reduce a un chequeo de tipos en tiempo lineal.

A.3.6. Proof-Carrying Code con Reglas No Confiables

Necula y Schneck [69] proponen remover las reglas de prueba de la política de seguridad, dejándolas fuera de la base confiable y así obteniendo una arquitectura PCC en la cual la corrección de las reglas de prueba debe ser verificada formalmente. Esto incrementa la flexibilidad de PCC, ya que cualquier actor, ya sea el consumidor o el productor, puede introducir nuevas políticas de seguridad.

Un prototipo de este enfoque [69] produce pruebas de corrección verificables automáticamente para los axiomas de tipos de las políticas de seguridad. Esto permite confiar en las reglas de prueba de nuevas políticas de seguridad sin la necesidad de verificar nuevamente el sistema y sin ninguna reducción en la escalabilidad de PCC para programas largos. Una diferencia con otras implementaciones es que se usa como lenguaje de especificación el sistema Coq [38], el cual tiene dos ventajas sobre otras elecciones posibles: una es que produce un objeto de prueba (un término lambda en un cálculo lambda tipado) para todo teorema probado y la segunda es su fuerte manejo de tipos y predicados definidos inductivamente.

El prototipo actual está limitado, por el momento, a producir la corrección de las reglas de prueba, las cuales son sólo una parte de la política de seguridad. El resto de la política se implementa como una extensión del **VCGen**.

A.3.7. Proof-Carrying Code y Linear Logical Framework

Plesko y Pfenning [66], en pos de facilitar la demostración de corrección, formalizaron la arquitectura confiable de PCC en un *Linear Logical Framework* (LLF) [20]. Esta formalización constituye un importante primer paso hacia un ambiente para experimentar y verificar formalmente propiedades de políticas de seguridad y sus implementaciones en arquitecturas PCC. La lógica lineal permite describir de manera natural la semántica de los lenguajes de programación, especialmente aquellos de naturaleza imperativa. El LLF permite especificaciones de alto nivel de código assembly, mientras que las políticas y las pruebas de seguridad pueden ser expresadas con el mismo lenguaje. Otra razón por la que introdujeron linealidad a la arquitectura PCC es la reducción que se obtiene en el tamaño de las pruebas.

Se implementaron dos modelos de ejecución para código assembly seguro, un **VCGen** y las reglas de inferencia del cálculo de predicados en el LF, mientras que el LLF provee el verificador de la prueba. Los resultados señalan que la especificación en el LLF es concisa, tiene un alto grado de abstracción y puede ser ejecutada usando la interpretación de programación lógica del LLF. Las características de LLF, que hicieron posible esta codificación natural, incluyen operadores lineales para estados de máquina y tipos dependientes.

A.3.8. Interactive Proof-Carrying Code

En Interactive PCC- iPCC - [77] se extiende el mecanismo de certificación haciéndolo interactivo y probabilístico. Con esto se pretende reducir el tamaño de la prueba que se envía al consumidor y el tiempo que necesita para hacer la verificación. El consumidor de código interactuando con el productor debe ser convencido, con una gran probabilidad, de la existencia y validez de la prueba de seguridad. Actualmente la interacción consiste en preguntas del consumidor que el productor contesta.

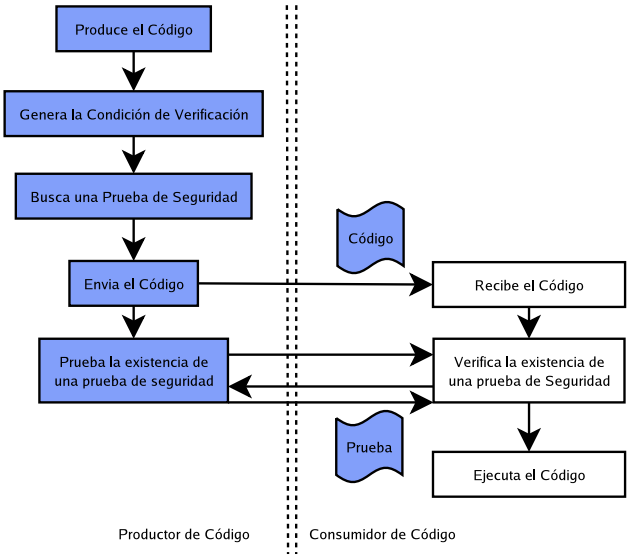

Figura A.4: Visión Global de una Arquitectura de Interactive Proof-Carrying Code.

iPCC puede ser extendido a zkPCC [77] (zero-knowledge PCC) en el cual la entidad encargada de realizar la prueba puede convencer al verificador de la existencia y validez de la prueba sin revelar información de la prueba en si misma.

Si bien estos enfoques, iPCC y zkPCC, reducen la complejidad de las pruebas son poco realistas al momento de implementarlos por la ineficiencia de la interacción necesaria.

A.4. Discusión

PCC constituye una línea de investigación activa, con una basta cantidad de interesantes problemas abiertos, que ha sumado los trabajos de destacados investigadores. La aplicación industrial de PCC tendría una importante repercusión económica al resolver el problema de la seguridad frente al código móvil, problema capital en la sociedad informatizada actual.

Para que PCC pueda ser aplicado industrialmente es necesario llegar a una implementación totalmente automatizada. Las soluciones que requieran intervención humana, son valiosas en el avance del conocimiento pero no puede pensarse en destinarlas al uso masivo.

Las dificultades que presenta el abordaje de PCC, tanto teórico como práctico, han llevado a utilizar una gran variedad de campos de conocimiento y teorías, abriéndose líneas que se insertan en ellos. Esta situación hace muy difícil poder abarcar a nivel de detalle las distintas corrientes de PCC siendo por lo tanto necesario para quien quiera trabajar en PCC optar por la especialización en una línea particular.

Como se comenta en el capítulo 2.2 la combinación de PCC, análisis estático de flujo de control y verificaciones dinámicas puede constituir una aproximación que permita implementaciones eficientes que solucione el problema del código móvil seguro, línea en la que se encuentra PCC-SA.

Apéndice B

Trabajos Relacionados

B.1. Compiladores Certificantes

En la actualidad se han desarrollado grandes avances en cuanto a la compilación certificante. Entre los más difundidos se encuentran Touchstone [55, 58], Special J [21], Cyclone [35, 39], TIL [74], FLINT/ML [71], Popcorn [48] y SPLINT [25]. Otro de los más conocidos es el compilador javac de Sun para Java que retorna Java bytecode. El propósito de javac es similar al de Special J, pero la salida del primero es mucho más simple que la del segundo porque el lenguaje de *bytecode* es mucho más abstracto que el generado por Special J. Por su parte, TIL y FLINT/ML son compiladores certificantes que mantienen información de tipos a través de todas las etapas de la compilación, descartándola solamente luego de la generación de código. Popcorn y Cyclone son compiladores certificantes cuyo lenguaje de salida es el lenguaje ensamblador tipado TAL. SPLINT es un compilador que verifica propiedades de seguridad utilizando diversas técnicas de análisis estático de flujo de control. A continuación se puede encontrar una breve descripción de cada uno de los compiladores antes mencionados.

Touchstone es un compilador certificante desarrollado por Necula y Lee [55, 58] que traduce programas escritos en un subconjunto tipado seguro de C a programas de lenguaje ensamblador del procesador **DEC Alpha** altamente optimizados. El compilador genera las invariantes de todos los ciclos y las especificaciones de todas las funciones locales. El lenguaje compilado por Touchstone tiene ciertas características no presentes en el lenguaje estándar C, tales como excepciones, un operador de longitud para arreglos y verificación de punteros nulos.

Uno de los objetivos de Touchstone fue demostrar que un compilador certificante puede generar código con calidad comparable a la de compiladores clásicos, como por ejemplo GNU/gcc. Touchstone tiene una desventaja en su performance sobre los compiladores antes mencionados debido a que la semántica del subconjunto seguro de C exige la verificación de límites de arreglos y de punteros nulos. El compilador intenta minimizar el costo de performance de estas verificaciones en tiempo de ejecución usando optimizaciones de condiciones redundantes. Pero, por supuesto, la performance no es todo: el código emitido por Touchstone no es sólo de diseño seguro sino también demostrablemente seguro utilizando un demostrador de teoremas.

Special J es un compilador certificante para Java sobre procesadores Intel x86, desarrollado por Colby, Lee, Necula y otros [21]. La principal diferencia entre un compilador estándar y Special J es que éste produce una prueba formal de que el código de máquina resultante es seguro respecto de tipos y de memoria.

El punto de partida del compilador Special J fue el compilador certificante Touchstone. La mayor ventaja de Special J sobre Touchstone es el alcance del lenguaje fuente compilado de éste, ya que necesita de mecanismos de tiempo de ejecución no triviales, tales como representación de objetos y métodos dinámicos, para despachar y capturar excepciones.

Cyclone es un compilador certificante para un subconjunto seguro de C [35, 39], que ha sido diseñado para prevenir violaciones de seguridad que son comunes en programas en C, tales como *buffer overflows*, *format string attacks* y errores de manejo de memoria. Cyclone brinda la misma garantía de seguridad que Java, de modo que ningún programa validado por el compilador puede cometer una violación de seguridad.

Las mayores diferencias entre Cyclone y C son relativas a la seguridad. El compilador Cyclone ejecuta un análisis estático sobre el código fuente e inserta chequeos de tiempo de ejecución en la salida compilada, en lugares donde el análisis no puede determinar que una operación es segura. El compilador puede no terminar de compilar el programa, ya sea porque el programa es verdaderamente inseguro o porque el análisis estático no es capaz de garantizar que el programa es seguro aún insertando chequeos en tiempo de ejecución.

Debe tenerse en cuenta que Cyclone rechaza algunos programas que un compilador C podría compilar perfectamente: esto incluye tanto todos los programas inseguros como algunos programas perfectamente seguros. Algunos programas seguros son rechazados porque Cyclone no puede implementar un análisis que separe los programas seguros de los inseguros.

Popcorn es un compilador certificante para un subconjunto seguro de C pero cuyas características inseguras, tales como aritmética de punteros, operador de dirección y *casts* de punteros, no son permitidas [48]. Compilar estas características de seguridad impondría una significante penalidad en la performance sobre el código de Popcorn. Por otra parte, Popcorn tiene varias características avanzadas que no están incluídas en el lenguaje C, tales como excepciones y polimorfismo paramétrico.

El compilador toma como entrada un código fuente similar a C y produce un programa en lenguaje ensamblador tipado TALx86. Un programa TAL contiene instrucciones de lenguaje ensamblador junto con anotaciones tipadas y pseudo-instrucciones TAL que son usadas por el verificador de tipos (*typechecker*) de TAL para verificar el programa.

TIL es un compilador optimizado dirigido por tipos para ML, que fuera desarrollado por Tarditi, Morrisett, Lee y otros [74] y está basado en cuatro tecnologías: polimorfismo intencional, recolección de basura (*garbage collection*), optimización del lenguaje funcional convencional y optimización de ciclos. TIL utiliza polimorfismo intencional y recolección de basura para proveer representaciones especializadas, incluso cuando SML (Standard ML) es un lenguaje polimórfico. La utilización de optimización del lenguaje funcional convencional permite reducir el costo de polimorfismo intencional, mientras que la utilización de optimización de ciclos permite generar buen código para funciones recursivas.

Una propiedad muy importante del compilador TIL es que todas las optimizaciones y las transformaciones son realizadas sobre un lenguaje intermedio tipado. Mantener información correcta de tipos durante las optimizaciones es necesario para soportar polimorfismo intencional y recolección de basura, ya que requieren información de tipos en tiempo de ejecución. TIL utiliza este lenguaje intermedio para garantizar que la información de tipos se mantiene en forma permanente, en lugar de confiar en invariantes *ad-hoc*.

FLINT/ML, desarrollado por Shao [71], toma como entrada código ML y retorna código intermedio tipado FLINT. El lenguaje FLINT es una representación de código móvil seguro, eficiente e independiente de la plataforma. Es utilizable con lenguajes orientados a objetos, funcionales e imperativos, y usa teoría de tipos y cálculo lambda de alto orden para codificar el sistema de tipos. Esto permite que el chequeo de tipos tome lugar en cualquier fase de la compilación.

La contribución más importante del compilador FLINT/ML consiste en su capacidad de compilar íntegramente el complejo lenguaje SML97 (extendido con módulos de alto orden) al lenguaje intermedio FLINT. Por otra parte, análisis de representaciones flexibles proveen una forma de compilar polimorfismos en FLINT sin restringir las representaciones de datos sobre código monomórfico.

SPLINT es un compilador que verifica propiedades de seguridad utilizando diversas técnicas de análisis estático de flujo de control [25]. Este compilador es asistido en el proceso de verificación

por anotaciones introducidas en el código fuente. SPLINT sacrifica efectividad por eficiencia y escalabilidad. Es decir, rechaza programas por ser inseguros cuando en realidad no lo son.

B.2. Código Móvil Seguro

Existe una gran variedad y cantidad de trabajos que se encuentran relacionados con PCC-SA. En especial aquellos enmarcados dentro del enfoque denominado seguridad basada en lenguajes y aquellos que utilizan análisis estático de flujo de control y de datos para verificar propiedades del código.

El desarrollo de PCC esta ligado intrínsecamente a otros áreas que brindan la base para su desarrollo, tales como teoría e implementación de *logical frameworks*, teoría y sistemas de tipos, verificación automática de sistemas, representación eficiente de pruebas, compilación y diseño de lenguajes. También existen líneas que son propias a PCC o están estrechamente relacionadas a él (como Compiladores Certificantes y Lenguajes Asembbly Tipados). También existen otras líneas que persiguen el mismo fin con enfoques independientes a PCC.

Sólo el desarrollo de compiladores certificantes podrá hacer que PCC pueda ser usado masivamente. Ellos permitirán compilar programas escritos en lenguajes de alto nivel a binarios PCC automáticamente. La conjunción de PCC y compilación certificante proverá una solución eficiente al problema de código móvil seguro. Un compilador certificante podrá producir además de código objeto seguro, las anotaciones e información de tipos requerida para una posterior verificación del receptor de su seguridad.

Efficient Code Certification (ECC) [41, 42] es un enfoque simple y eficiente para la certificación de código compilado proveniente de una fuente no confiable. Este enfoque puede garantizar propiedades básicas, pero no triviales, de seguridad incluyendo seguridad de control de flujo, seguridad de memoria y seguridad de stack. Esto lo lleva a cabo de una forma simple, eficiente y lo más importante es que se puede incorporar a los compiladores sin dificultades. ECC es menos expresivo que PCC y TAL pero sus certificados son relativamente más compactos y fáciles de producir y verificar.

Otro conjunto interesante de trabajos son aquellos relativos a Lenguajes Assembly Tipados (TAL). Estos lenguajes proveen una sólida base para obtener sistemas PCC totalmente automáticos, además los sistemas de tipos brindan el soporte para forzar a utilizar abstracciones en lenguajes de alto nivel, entonces un programa bien tipado no puede violar estas abstracciones. Morriset et al. [48] presenta el diseño de TALx86 un TAL para INTEL x86. TALx86 comenzó con funciones de alto orden y polimorfismo, considerando a System F como un lenguaje de alto nivel. De este lenguaje surgieron algunas extensiones como STAL [72] (el cual modela el stack con polimorfismo), DTAL [76] (usa tipos dependientes, permitiendo entre otras cosas hacer algunas optimizaciones en la verificación de accesos a arreglos) y Alias TAL [47] (permite reusar áreas de memoria que TALx86 prohibe, permitiendo *aliasing*). Si bien el enfoque basado en teoría de tipos, en el cual la seguridad es garantizada por medio de la verificación de tipos, debe ser considerado un enfoque diferente a PCC no hay que dejar de mencionar que la teoría de tipos es una de las bases de PCC.

Franz et al. [26] muestra una solución híbrida, para el problema del código móvil, entre PCC y maquinas virtuales (VM, por Virtual Machine). Su meta es conjugar por un lado alta performance de ejecución y compilación veloz "just-in-time"; y por otro lado pruebas eficientes y pequeñas de seguridad de tipos. Ellos diseñaron una VM que soporta PCC reduciendo así la complejidad de las pruebas requeridas a unas pocas obligaciones de prueba.

Otro enfoque, Model Carrying Code (MCC) [70], consiste en generar un modelo del comportamiento de los programas. Entonces se envía el modelo junto con el programa al consumidor. Estos modelos son mucho menos complejos que los programas y, por lo tanto, es posible que el consumidor verifique mecánicamente si el modelo cumple con la política de seguridad. Si bien el modelo se verifica estáticamente luego debe existir un monitor que controle, en tiempo de ejecución, que el programa se corresponde con el modelo. La necesidad del monitor hace perder las ventajas obtenidas con la verificación estática.

E. Albert et al. [5] introducen un enfoque con un esquema similar a PCC basado en interpretación abstracta [22]. El intérprete abstracto genera una tabla con información a partir del

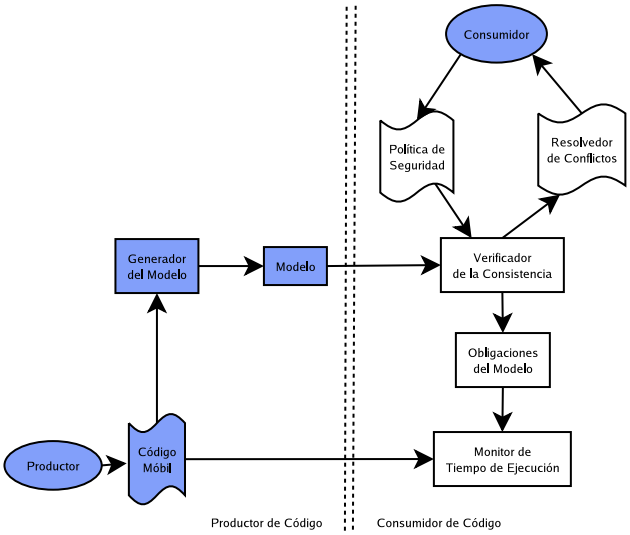

Figura B.1: Visión Global de una Arquitectura de Model Carrying Code.

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 [68] afirma que interpretación abstracta es una poderosa metodología para diseñar análisis estáticos correctos por construcción.

WELL (Wellformed Encoding at the Language-Level) [29] es un enfoque basado en transportar "árboles sintácticos comprimidos" (CASTs). Esta representación permite transportar de manera segura la semántica a nivel del código fuente, es decir, a un nivel más alto que el usado en PCC. Además se afirma que esta representación es segura por construcción; aquello que no satisface las restricciones de seguridad no se puede expresar con esta codificación. Esto reduce la base confiable necesaria pero los resultados que obtuvieron solo les permite afirmarlo para verificar *escape analysis*.

Control estático de flujo de información es un nuevo enfoque, muy prometedor, para proteger confidencialidad e integridad de la información manipulada por programas. El compilador traza una correspondencia entre los datos y la política de flujo de la información que restringe su uso [51, 68].

J. Bergeron et al. [15] 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.

Bibliografía

[1] J. Aguirre, R. Medel, M. Arroyo, N. Florio, F. Bavera, P. Caymes Scutari, D. Nordio. "Avances en Procesadores de Lenguajes y Proof-Carrying Code". En *V Workshop de Investigadores en Ciencias de la Computación (WICC 2003)*. Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil (Argentina). 22 y 23 de Mayo, 2003.

[2] A. Aho, J. Ullman. The Theory of Parsing, Translation, and Compiling. Volumen I: Parsing. Prentice-Hall. 1972.

[3] A. Aho, J. Hopcroft, J. Ullman. Data Structures and Algorithms. Adisson Wesley. 1983.

[4] A. Aho, R. Sethi, J. Ullman. Compilers: Principles, Techniques and Tools. Adisson Wesley. 1986.

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

[6] A. Appel. Modern Compiler Implementation in Java. Cambridge University Press. 1999.

[7] A. Appel, A. Felty. "A Semantic Model of Types and Machine Instructions for Proof-Carrying Code". En *Proceedings of the 27^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL'00), pp. 243–253, ACM Press, Boston, Massachusetts (EE.UU.). Enero 2000.

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

[9] A. Appel, E. Felten. "Models for Security Policies in Proof-Carrying Code". Princeton University Computer Science Technical Report TR-636-01. Marzo 2001.

[10] F. Bavera, M. Nordio, R. Medel, J. Aguirre, G. Baum, M. Arroyo. "Un Survey sobre Proof-Carrying Code". *5to Simposio Argentino de Computación, AST 2004*. Universidad de Córdoba (Argentina). Septiembre 2004.

[11] F. Bavera, M. Nordio, J. Aguirre, G. Baum, R. Medel. "Optimización del Prototipo del Entorno de Ejecución de PCC-SA". En *X Congreso Argentino de Ciencias de la Computación*. Universidad de la Matanza, Buenos Aires (Argentina). Octubre de 2004.

[12] F. Bavera, M. Nordio, J. Aguirre, M. Arroyo, G. Baum, R. Medel. "Grupo de Procesadores de Lenguajes - Línea: Código Mívil Seguro". En *VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004)*. Universidad Nacional del Comahue, Neuquén (Argentina). 20 y 21 de Mayo, 2004.

[13] F. Bavera, M. Nordio, J. Aguirre, G. Baum, R. Medel. "Desarrollo de un Prototipo de Compilador Certificante". En *VIII Jornadas de Informática e Investigación Operativa (JIIO03)*. Universidad de la República, Facultad de Ingeniería, INCO. Montevideo, Uruguay. 24 al 28 de Noviembre de 2003.

[14] F. Bavera, M. Nordio, J. Aguirre, G. Baum, R. Medel. "Avances en Proof-Carrying Code". En los resúmenes del *IX Congreso Argentino de Ciencias de la Computación, CACIC 2003*. Buenos Aires (Argentina). Octubre de 2003.

[15] 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.

[16] E. Berk. "JLex: A lexical analyzer generator for Java". Department of Computer Science, Princeton University. 1997.

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

[18] B. Bershad, S. Savage, et. al. "Extensibility, Safety and Performance in the SPIN Operating Sistem". En *Proc. 15th Symp. Operating System Principles*, pp. 267-284. ACM. Diciembre de 1995.

[19] R. Boyer, J. Moore. "A Computational Logic". Academic Press, 1979.

[20] I. Cervato, F. Pfenning. "A Linear Logical Framework". En *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pp. 264–275, IEEE Computer Society Press, New Brunswick, New Jersey (EE.UU.). Julio 1996.

[21] C. Colby, P. Lee, G.Ñecula, 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.

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

[23] D. Detlefs. "An orverview of the Extended Static Checking system". En *Proceedings of the First Formal Methods in Software Practice Workshop*, pp. 1–9, ACM SIGSOFT. Enero 1996.

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

[25] D. Evans y D. Larochelle. "Improving Security Using Extensible Lightweight Satatic Analisis". IEEE Software. Enero/Febrero 2002.

[26] M. Franz, D. Chandra, A. Gal, V. Haldar, F. Reig y N. Wang. "A Portable Virtual Machine Target for Proof-Carrying Code". Departament of Computer Science, University of California. 2002

[27] M. Gordon. "HOL: A Machine Oriented Formulation of Higher Order Logic". Tech. Rep. 85, University of Cambridge, Computer Laboratory. Julio 1985.

[28] J. Gosling. "Java intermediate bytecodes". En Proc. ACM SIGPLAN Workshop on Intermediate Representations, pages 111-118. ACM. 1995.

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

[30] V. Haldar, C. Stork, C. Krintz y M. Franz. "Tamper-Proof Annotations By Construction". Technical Report 02-10. Departament of Computer Science, University of California. 2002

[31] J. Halpern, V. Weissman. "Using first-order logic to reason about policies". En Proceedings of the 16th IEEE Computer Security Foundations Workshop, pp. 187-201. 2003.

[32] N. Hamid, Z. Shao,V. Trifonov,S. Monnier y Z. Ni. "A Syntactic Approach for Foundational Proof-Carrying Code". En *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pp. 89–100, Copenague, Dinamarca. 2002.

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

[34] T. Henziger, G. Necula, R. Jhala, R. Majumbar, G. Sutre, W. Weimer. "Temporal-Safety Proofs for Systems Code". En *Proceedings of the 14^{th} International Conference on Computer-Aided Verification*, pp. 525–538, Lecture Notes in Computer Science, Springer-Verlag. 2002.

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

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

[37] S. Hudson. "CUP User's Manual". Graphics Visualization and Usability Center Georgia Institute of Technology. 1999.

[38] G. Huet, G. Kahn, C. Paulin-Mohring. "The Coq proof assistant: a tutorial, version 6.1.". INRIA Technical Report RT 0204, Francia. Agosto 1997.

[39] 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.

[40] S. Johnson. "Yacc - yet another compiler compilers". Computing Science Technical Report. AT&T. 1975.

[41] D. Kozen. "Efficient Code Certification". Tech. Report 98-1661, Cornell Univ.. 1998.

[42] 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.

[43] X. Leroy. "Bytecode Verification on Java smart cards". En *Proceedings Software Practice and Experience*. 2002.

[44] J. Levine, T. Mason, D. Brown. Lex & Yacc. O'Reilly & Associates, Inc.. 1992.

[45] T. Lindholm, F. Yellin. "The Java Virtual Maquine Specification". The Java Series. Addison-Wesley, 1999. Second Edition.

[46] N. Michael, A. Appel. "Machine Instruction Syntax and Semantics in Higher Order Logic". En *17^{th} International Conference on Automated Deduction* (CADE-17), Springer-Verlag (Lecture Notes in Artificial Intelligence). Junio 2000.

[47] G. Morrisett, K. Crary, N. Glew, D. Walker. "Stack-Based Typed Assembly Language". En *Second International Workshop on Types in Compilation*, pp. 95–117, Kioto. 1998.

[48] 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.

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

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

[51] A. Myers. “JFlow: Practical Static Information Flow Control”. En *Proc. 26th Symp. Principles of programming Languages*. ACM. Enero de 1999.

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

[53] G. Necula y P. Lee. “Safe Kernel Extensions Without Run-Time Checking”. Second Symposium on Operating System Design and Implementation (OSDI’96), Seattle. 1996.

[54] G. Necula, P. Lee. “Safe, Untrusted Agents using Proof-Carring Code”. Carnegie Mellon University. 1997.

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

[56] G. Necula y P. Lee. “Efficient Representation and Validation of Proof”. En Proceedings of the 13th Annual symposium on Logic in Computer Science, Indianapolis. 1998.

[57] G. Necula y P. Lee. “Efficient Representation and Validation of Proof”, Reporte Técnico. 1998.

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

[59] G. Necula. “A Scalable Architecture for Proof-Carrying Code”. En *Fifth International Symposium on Functional and Logic Programming*, Tokio. 2001.

[60] G. Necula, R. Schneck. “Proof-Carrying Code with Unstrusted Proof Rules”, en *Proceedings of the 2^{nd} International Software Security Symposium*. Noviembre 2002.

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

[62] 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.

[63] M. Nordio, F. Bavera, R. Medel, J. Aguirre, G. Baum. “STALlion: A Simple Typed Assembly Language for Static Analysis”. En *X Congreso Argentino de Ciencias de la Computaciín*. Universidad de la Matanza, Buenos Aires (Argentina). Octubre de 2004.

[64] D. Oliva, J. Ramsdell, M. Wand. “The VLISP verified PreScheme Compiler”. *Lisp and Symbolic Computation*, 8(1-2), pp. 111–182. 1995.

[65] F. Pfenning. “Elf: A meta-language for deductive systems”. En *Proceedings of the 12^{th} International Conference on Automated Deduction*, pp. 811–815, Francia. 1994.

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

[67] D. Richardson. “TAOS: Testing with Analysis and Oracle Support”. En *ACM SIGSOFT Software Engineering Notes: Proceedings of the 1994 International Symposium on Software Testing and Analysis*. 1994, Seattle, Washington, EE.UU.

[68] A. Sabelfeld, A. Myers. “Language-Based Information-Flow Security”. IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security. 2003.

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

[70] R. Sekar, C. Ramakrishnan, I. Ramakrishnan y S. Smolka. “Model Carrying Code (MCC): A New Paradigm for Mobile-Code Security”. Departament of computer Science, SUNY at Stony Brook, New York. 2003.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.

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

[77] Y. Ysukada. “Interactive and Probabilistic Proof of Mobile Code”. En *Advances in Infraestructure for Electronic Business, Science and Education on the Internet* (SSGRR 2001).

[78] Joachim Wegener, Ines Fey. “Systematic Unit-Testing of Ada Programs”. En *Proc de Ada-Europe*. pp 64-75. 1997.

[79] David Walker. “'A Type System for Expressive Security Policies”. En *Proc. FLOC'99 Workshop in Real-Time Result Verification*. Julio de 1999.

[80] 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.