CCMini: Un Prototipo de Compilador Certificante*

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

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

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

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

Abstract

Certifying compilers use static information of a program to verify that it complies with certain security properties and to generate certified code. To do so, those compilers translate the source program into an annotated program written in some intermediate language. These annotations are used to verify the generated code. Given a source program, a certifying compiler will produce object code, annotations, and a proof that the code comply with the customer's security specifications. Thus, certifying compilers can automatically produce the security evidence required to establish a Proof-Carrying Code (PCC) setting.

In this work we present CCMini, a certifying compiler for a subset of the language C. This compiler guarantees that compiled programs do not read uninitialized variables and do not access to undefined array positions. The verification process is carried on abstract syntactic trees by using static analysis techniques; in particular, control analysis and data analysis are used.

Keywords: Code Verification, Static Analysis, Safe Mobile Code, Proof-Carrying Code, Programming Languages

Resumen

Los compiladores certificantes usan información estática del comportamiento de un programa para verificar que éste cumple con ciertas propiedades de seguridad y generar código certificado. Para ello, estos compiladores traducen el programa fuente a un programa con anotaciones, escrito en algún lenguaje intermedio. Estas anotaciones son utilizadas para verificar el código generado. Dado un programa fuente, un compilador certificante producirá el código objeto, las anotaciones y una prueba formal de que el código generado cumple con las especificaciones de seguridad provistas por el consumidor del código. Por lo tanto, los compiladores certificantes pueden producir automáticamente las evidencias de seguridad requeridas en un ambiente de *Proof-Carrying Code*.

En este trabajo presentamos **CCMini**, 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, utilizando técnicas de análisis estático. En particular, se utilizan las técnicas de análisis de control de flujo y análisis de datos.

Palabras claves: Verificación de Código, Análisis Estático, Código Móvil Seguro, Proof-Carrying Code, Lenguajes de Programación

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

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

1 Introducción

La certificación de código es una técnica desarrollada para garantizar y demostrar 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 realize 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*.

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. Si la verificación tuvo éxito, entonces es seguro ejecutar la aplicación final.

Existen ventajas adicionales a las corcenientes en seguridad en la utilización de compiladores certificantes en lugar de compiladores tradicionales, ya que la mayor cantidad de información generada por un compilación 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 [12, 13] 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 [5], Cyclone [10, 8], TIL [14], FLINT/ML [16] y Popcorn [11]. También es considerado 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 [11], por lo que incluyen información de tipos aún en el código objeto.

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 indices 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*.

En este trabajo presentamos **CCMini**, 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 utilizando técnicas de análisis estático. En particular, se utilizan las técnicas de análisis de control de flujo y análisis de datos.

Los compiladores certificantes antes mencionados utilizan lenguajes ensambladores tipados para expresar el código intermedio. En cambio, nuestro compilador genera un árbol sintáctico abstracto anotado con la información de tipos necesaria para verificar las propiedades de seguridad. Otra diferencia 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 un análisis estático sobre el código intermedio generado y prueba que no se viole la política de seguridad siguiendo el flujo y utilizando las reglas que expresan formalmente la política de seguridad (figura 2).

Este trabajo está estructurado de la siguiente manera: en la sección 2 se presenta la arquitectura general del compilador certificante desarrollado. En las secciones 3, 4 y 5 se comentan las características más relevantes del lenguaje fuente, la política de seguridad que utiliza el compilador para realizar las verificaciones y el código intermedio. En las secciones 6 y 7 se describen los componentes más importantes de nuestro compilador certificante. Finalmente, en la sección 9 se presentan las conclusiones extraídas de este trabajo

y se delinean futuros emprendimientos.

2 El Compilador Certificante CCMini

El compilador certificante CCMini está compuesto por un compilador tradicional y un generador y verificador de anotaciones. El **Compilador** toma como entrada *código fuente* en lenguaje "Mini" y genera como código intermedio un *árbol sintáctico abstracto* (*ASA*). El **Generador y Verificador de Anotaciones** (**GenAnot**) realiza diversos análisis estáticos de flujo de control y de flujo de datos sobre el *ASA*, generando un *árbol sintáctico abstracto* anotado. Estas anotaciones incluyen información sobre cotas de variables e invariantes de ciclos. La política de seguridad especifica las propiedades que el código debe poseer. **GenAnot** es el encargado de verificar que el código no viola la política de seguridad. Si el código viola la política de seguridad, es rechazado. En caso contrario, el código objeto puede ser generado de dos maneras distintas. Para la ejecución sobre un entorno confible, en este caso el **Generador de Código Objeto** toma el *ASA* y genera el código objeto. Y para la ejecución sobre un entorno no confiable, el **Generador de Código y Prueba** toma el *ASA* y genera el código objeto y una prueba de que el código cumple con la política de seguridad para ser enviados al consumidor de código. En aquellos puntos del programa donde no se puede determinar fehacientemente si se cumple o no la política de seguridad, **GenAnot** inserta código que realizará una verificación en tiempo de ejecución. Esto permite ampliar el rango de programas certificados sin poner en riesgo la seguridad del consumidor del código. La figura 1 muestra los componentes antes mencionados y su interacción.

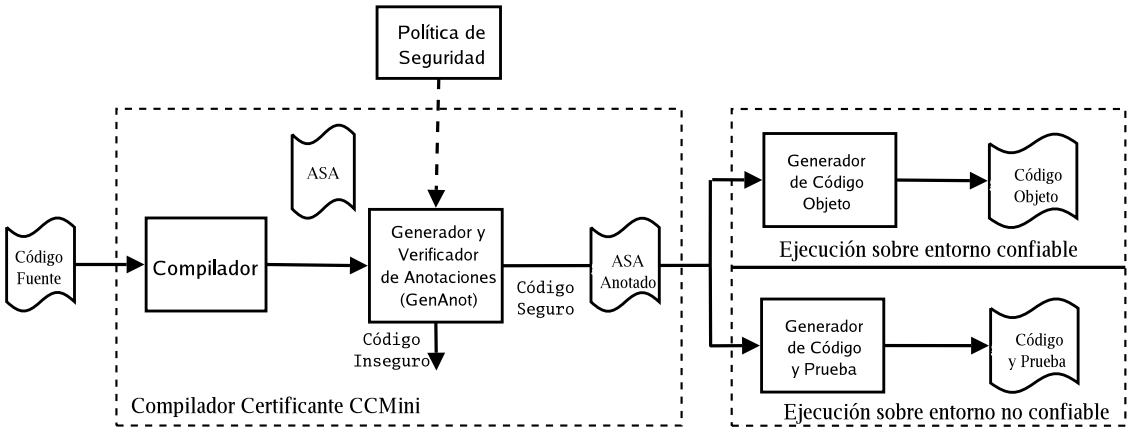

Figura 1: Esquema general del Compilador Certificante CCMini.

3 El Lenguaje "Mini"

En esta sección definimos la estructura y el significado de un programa "Mini". Este lenguaje fuente tiene una sintaxis similar a un subconjunto del lenguaje C. Básicamente, un programa "Mini" es una función que debe tomar al menos un parámetro y retornar un valor. Tanto los parámetros como los valores que retorna deben ser de uno de los tipos básicos (enteros y booleanos). A diferencia de C, en el cual cero significa falso y cualquier entero distinto de cero significa verdadero, "Mini" posee el tipo booleano. Además, cuenta con arreglos unidimensionales cuyos elementos pueden ser de un tipo básico. Cabe aclarar que la complejidad de utilizar arreglos unidimensionales es la misma que si se incluyen estructuras de datos más complejas, tales como matrices.

El lenguaje cuenta con sentencias condicionales (`if, if-else`), la sentencia iterativa (`while`) y la sentencia `return`. A diferencia de C, la asignación es una sentencia y no una expresión. Esta última decisión de diseño fue tomada a fin de simplificar la semántica del lenguaje fuente. "Mini" también permite escribir bloques de sentencias delimitados por paréntesis ({ y }). En cada bloque opcionalmente se pueden declarar e inicializar variables. Las reglas de visibilidad y tiempo de vida son las mismas que en C.

Además, los parámetros de tipo entero pueden ser acotados al ser declarados. Para esto, la declaración del parámetro va acompañada por un rango que indica el menor y el mayor valor que puede tomar el 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.

Los identificadores se representan de la manera usual, al igual que las constantes numéricas y las boleanas (`true` y `false`). Los operadores están clasificados en operadores matemáticos (+, -, * y /), operadores booleanos (|| y &&) y operadores relacionales (==, >, <, ≥, ≤, y ≠). Todos ellos con su significado habitual.

4 La Política de Seguridad

Una política de seguridad es un conjunto de reglas que definen si un programa es 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 [15].

La política de seguridad que definimos garantiza seguridad de tipos y de memoria, además de garantizar que no se leen variables no inicializadas y que no se accede a posiciones no definidas sobre los arreglos. En la siguiente subsección se presenta una formalización de dicha política de seguridad.

4.1 Fomalización 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 extremadamente 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 ::=	$+ \mid - \mid * \mid /$
BoolExpressions:	be ::=	$b \mid be_1\ bop\ be_2 \mid not\ be \mid me_1\ rop\ me_2$
BoolOperations:	bop ::=	$\&\& \mid \|\|$
RelationalOperations:	rop ::=	$> \mid < \mid \geq \mid \leq \mid == \mid \neq$
Types:	τ ::=	bt | array (bt , e)
BasicTypes:	bt ::=	int (Min, Max) | boolean
Predicates	P ::=	$P_1 \wedge P_2 \mid P_1 \supset P_2 \mid \forall x P_x \mid e_1 \leq e_2 \mid e : \tau$
		$\mid SaferRead(x) \mid SaferWrite(x_1, x_2)$

Figura 2: Sintaxis de las expresiones de tipos.

Las figuras 2 y 3 establecen formalmente la política de seguridad por medio de la definición de las expresiones, tipos, predicados y reglas de tipado para el lenguaje "Mini". La categoría sintáctica *Expressions* define todas las expresiones matemáticas y booleanas. Las expresiones mateáticas 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 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 **write_int** también actualiza el valor del rango de la variable. La regla **write_array** garantiza que es seguro escribir el valor x

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

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

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

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

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

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

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

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

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

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

Figura 3: Reglas del sistema de tipos.

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** y **var_mayor**, las cuales indican los nuevos valores que toman el mínimo y el máximo, respectivamente.

5 El Código Intermedio: Arbol Sintáctico Abstracto

El código intermedio que genera el compilador certificante es un árbol sintáctico abstracto (ASA). Éste es una representación abstracta del código fuente y permite realizar distintos análisis estáticos, tales como análisis de flujo de control y análisis de flujo de datos. También puede ser utilizado para generar una gran cantidad de optimizaciones del código.

La estructura de los árboles sintácticos abstractos que utilizamos es similar a la de los ASAs tradicionales, aunque aquéllos permiten incluir anotaciones de código. 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.

6 El Compilador

Este componente es un compilador tradicional que toma como entrada código fuente "Mini" y produce como salida un árbol sintáctico abstracto. Se implementó siguiendo un esquema típico [1] y, al igual que la mayoría de los compiladores, sólo rechaza el código fuente que no cumple con las especificaciones léxicas y sintácticas impuestas por el lenguaje fuente.

7 El Generador de Anotaciones: GenAnot

Las anotaciones del código permiten utilizar el sistema de verificación. Estas anotaciones están constituídas por las invariantes de ciclos, la información de cota de todas las variables, la precondición y la postcondición del programa. En las próximas subsecciones se detalla la información generada, como así también los procesos que la generan.

A diferencia de otros compiladores certificantes, el componente **GenAnot** genera las anotaciones del código y realiza las verificaciones simultáneamente. Para lograr esto se realizan análisis de flujo de control y de datos sobre el árbol sintáctico abstracto descripto en la sección 5. Estos análisis determinan la inicialización y el rango de las variables utilizadas. Esto último permite asegurar si un valor es válido como índice a un arreglo. Además, en ciertos casos se está en condiciones de determinar la precondición y la postcondición.

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–, el esfuerzo requerido es mucho menor en comparación con la verificación formal de programas. 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 considerados inseguros por el compilador cuando en realidad no lo son.

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

Como se mencionó previamente, se pueden insertar chequeos en tiempo de ejecución a fin de ampliar el rango de programas considerados seguros a aquellos en los cuales el análisis estático no puede asegurar nada. Es decir, el mecanismo usado para verificar que el código es seguro puede ser una combinación de verificaciones estáticas –en tiempo de compilación– y de chequeos dinámicos –en tiempo de ejecución–. Cabe resaltar que en muchos casos es necesario insertar estos chequeos dinámicos, no sólo por las limitaciones del análisis estático particular sino porque los problemas a resolver no son computables. Por ejemplo, garantizar que el acceso a los arreglos se hace respetando sus límites es, en su caso general, equivalente al problema de la parada.

En lss siguientes subsecciones se detallan los procedimientos llevados a cabo por el módulo **GenAnot** para identificar las variables inicializadas y determinar los rangos de las variables.

7.1 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 5 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.

7.2 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.

7.3 Identificación de rangos de 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.
4. Una expresión e_1 *op* e_2 tiene un rango si todos sus operandos tienen un rango definido, y su rango está definido por: $Max = Maximo(e_1.Min\ op\ e_2.Min, e_1.Min\ op\ e_2.Max, e_1.Max\ op\ e_2.Min, e_1.Max\ op\ e_2.Max)$
   $Min = Minimo(e_1.Min\ op\ e_2.Min, e_1.Min\ op\ e_2.Max, e_1.Max\ op\ e_2.Min, e_1.Max\ op\ e_2.Max)$
5. Cuando a una variable se le asigna una expresión que tiene un rango definido, el rango de la variable queda definido por el rango de la expresión. En el caso en que la expresión no tenga un rango definido, 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`). El rango de las variables para el resto de programa está definido por la unión de los rangos de las variables de 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 los rangos 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 $3 \leq z \leq 16$.

7.4 Identificación de rangos de 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.

7.5 Identificación de rangos de variables inductivas que son modificadas dentro de un ciclo

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 intervie 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:

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

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:

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

8 Un ejemplo

Este programa fue compilado por CCMini que pudo demostrar que todos los accesos a arreglos son seguros, por lo cual generó código certificado cuya única verificación dinámica consiste en convalidar la precondición impuesta a *limite*.

```
int bubble_sort (int limite (0,99) ) /* en Mini se puede especificar el rango de
                                                variacion de los parametros    */
{
  int [100] a; /* declaracion del arreglo a */
  int t,j, uno i=0; /* vble temporaria t e indices i,j */

  uno=365-364;
  i=0;    /* ordena el arreglo... */
  while (i<limite) {
     j=limite;    /* ...usando el metodo de burbujas */
     while (j>i) {
        if (a[j]<a[j-1]) {
           t=a[j]; /* ...y haciendo swapping con una vble temporaria */
           a[j]=a[j-uno];
           a[j-uno]=t;
        }
        j=j-uno;
     }
     i=i+uno;
  }
  return 0;
}
```

9 Conclusiones y Trabajos Futuros

9.1 Conclusiones

El prototipo desarrollado (CCMini) muestra la factibilidad de utilizar técnicas de análisis de flujo de control y flujo de datos para la implementación de compiladores certificantes.

La observación informal de programas seleccionados al azar muestra que generalmente los accesos a arreglos se realizan mediante variables inductivas, lo cual CCMini sugeriría que puede determinar en la mayoría de los casos prácticos la validez de los accesos, evitando así la necesidad de insertar chequeos dinámicos.

En los casos de prueba utilizados[2] CCMini logró determinar la seguridad sin la necesidad de introducir verificaciones dinámicas en la mayoría de los casos.

CCMini es en si un compilador certificante que puede ser utilizado en un ambiente confiable, pero además produce gran parte de las estructuras requeridas para la implementación de un entorno de producción de PCC.

CCMini solo descarta aquellos programas para los cuales pudo determinar formalmente que su comportamiento es inseguro ya que en los otros casos introduce verificaciones dinámicas en todos los puntos de seguridad incierta.

9.2 Trabajos Futuros

El grupo se encuentra trabajando en la extensión del desarrollo expuesto a un ambiente de PCC cuyo compilador certificante estará basado en CCMini.

[2]Los casos de prueba fueron obtenidos solicitando a programadores que no participaron del desarrollo de CCMini la confección de programas centrados en la manipulación de arreglos. Para obtener una muestra significativa habría que contar con bibliotecas numerosas cosa que solo puede lograrse utilizando un lenguaje estandar

Sería interesante estudiar la extensión del uso de las técnicas estáticas a políticas de seguridad mas generales, por ejemplo que incluyan la manipulación de punteros.

Sería interesante poder contar con una estadística, computada sobre una muestra real y numerosa de programas, sobre los resultados de aplicar las técnicas de análisis estático usadas para garantizar la seguridad de acceso a arreglos.

Referencias

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

[2] A. Appel, "Modern Compiler Implementation in Java". Cambridge University Press. 1999.

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

[4] F. Bavera, M. Nordio, J. Aguirre, M. Arroyo, G. Baum, R. Medel, "Grupo de Procesadores de Lenguajes - Línea Código Móvil Seguro", en *Proceedings del WICC'04*, Universidad del Comahue, Neuquén, Argentina, 2004. http://dc.exa.unrc.edu.ar/lenguajes/Files/Entorno_codigo_movil.pdf

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

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

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

[8] 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.

[9] G. Huet et al, "The Coq Proof Assistant, Reference Manual, Version 6.1", INRIA-Rocquencourt CNRS-ENS Lyon, 1997.

[10] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang, "Cyclone: A safe dialect of C", en *USENIX Annual Technical Conference*, Monterrey, California (EE.UU.), Junio 2002.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.