CCMini: A Prototype of Certifying Compiler based on Annotated Abstract Syntax Trees[*]

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

(1) Universidad Nacional de Río Cuarto, Departamento de Computación
Río Cuarto, Argentina
{pancho,nordio,jaguirre}@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

2005

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 simple 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, Certifying Compilers.

[*]This work was supported by grants from the SECyT-UNRC, the Agencia Córdoba Ciencia, and the NSF.

[†]This author's work was supported by the NSF project *CAREER: A Formally Verified Environment for the Production of Secure Software*–#0093362.

1 Introduction

Code certification is an approach used to guarantee the safety of untrusted mobile code. Particularly, this technique is based on the analysis of critical qualities, such as type safety and memory safety. The basic idea is to require the code producer provide a formal proof (a *certificate*) showing that the code satisfies the consumer's security policy. Thus, the code consumer can use a small verifier to check the proof, and it can execute safely the mobile code only if the proof is successfully verified.

This approach is the base of techniques for secure execution of mobile code, such as PCC, developed by Necula and Lee [8]. In order to minimize the consumer's *Trusted Computing Base* (TCB), the burden of the proof is left to the code producer. But, in order to automatize the certifying code approach, it is necessary for the code producer to use a *certifying compiler* as a mean to produce the code and its certificate.

The mentioned Necula and Lee developed the first certifying compiler, Touchstone [9]. After that, several advances in certifying compilation techniques were introduced by the certifying compilers Special J [2], Cyclone [6, 5], TIL [10], FLINT/ML [12], and Popcorn [7]. Furthermore, the well-know javac compiler of Sun, which produces Java bytecode, is considered a certifying compiler. This last compiler has a similar purpose than Special J, but its output is more simple because the bytecode language is more abstract than the generated by Special J. Compilers TIL and FLINT/ML maintain type information through the compilation process, but this information is eliminated after the code generation. By the other hand, Popcorn and Cyclone are certifying compilers whose target language is the typed assembly language TAL [7], which allows to include type information in the generated code.

A certifying compiler can be divided in two phases. In the first phase the source code is translated to an intermediate language, and security annotations are introduced. These annotations will guide the verification steps of the next phase, and they include the security conditions that the execution of the code must satisfy in critical points. The second phase proves that the annotated intermediate code satisfies the consumer's security policy. If the verification is successful, then the code can be executed safely.

The use of certifying compilers has an additional advantage over the traditional ones. Since a greater amount of information about the program behavior is generated by a certifying compiler, several several code optimizations techniques can be applied.

In this work we present CCMini, a certifying compiler for a extended subset of the C language, that guarantees that compiled programs does not read uninitialized variables and does not access to undefined array positions. The verification process is done over an abstract syntax tree by using static analysis techniques. Particularly, CCMini uses control-flow and data-flow analysis techniques.

Compared against certifying compilers based on logical frameworks, such as Touchstone [8], the advantage of our method is that the complexity of the certificate generation algorithm is linear with respect to the size of the input program. Moreover, compared against javac, our certifying compiler inserts a lesser amount of dynamic verifications.

Furthermore, the advantage of CCMini over certifying compilers based on type systems, is that several interesting security policies either cannot be verified by using a type system, or the verification process is too costly. For example, the initialization of variables and the control of out-of-bound array accesses cannot be verified efficiently with a compiler based on such formal system. Our approach is based on the fact that the information required to guarantee compliance with these security properties can be generated by making a static analysis of the program's control flow [3, 4].

Our main goal is to provide an efficient and effective solution to the verification of security policies that cannot be verified efficiently by using an approach based on type systems. The authors Haldar, Stork, and Franz [4] argue that the efficiency problems of type systems are due to the semantic differences between the source code and the low-level object code. In contrast, CCMini uses a high-level intermediate language: an *abstract syntax tree* with type annotations. This intermediate representation enable us to use static analysis techniques to generate and verify the typing information, and also apply several code optimization techniques.

Static analysis can be used for the verification of security properties because a concrete approximation to the dynamic behavior of a program can be obtained at compile-time. There exist several static analysis techniques that provide a good balance between the design costs and the solutions provided, such as *array-bound checking* [3].

This paper is structured as follows: in Section 2 we present the general architecture of the developed certifying compiler. We discuss the more important features of the source language, the security policy, and the intermediate code in Sections 3, 4, and 5, respectively. We describe in detail the more relevant components of the certifying compiler in Sections 6 and 7. The last section is devoted to the conclusions of our work and some proposals of future work.

2 The CCMini Certifying Compiler

The CCMini certifying compiler is composed by a traditional compiler plus a generator and verifier of annotations. The source language of the compiler is the language Mini. This compiler generates as intermediate code an abstract syntax tree (AST). The second phase, an **Annotations Generator and Verifier** (**GenAnot**) performs various static analysis of control-flow and data-flow over the AST, generating an annotated abstract syntax tree. These annotations include information about bounds of the variables and certain loop invariants. **GenAnot** verifies that the code does not violate the consumer's security policy. If the code violates the security policy, then the code is rejected. Otherwise, object code can be generated with or without a certificate, depending on if the code will be executed in the consumer's TCB or in the producers environment.

In the first case, the **Code and Proof Generator** takes the AST and generates object code and a proof that the code complies with the consumer's security policy. Both, object code and proof, are sent to the code consumer. In the second case, the **Object Code Generator** takes the AST and generates only object code.

In those points of the program where it cannot be determined if the security policy is followed or not, **GenAnot** inserts code that perform a verification in run-time. This allows us to extend the range of certified programs without put in risk the security of the code consumer. Figure 1 shows the components cited before and their interaction.

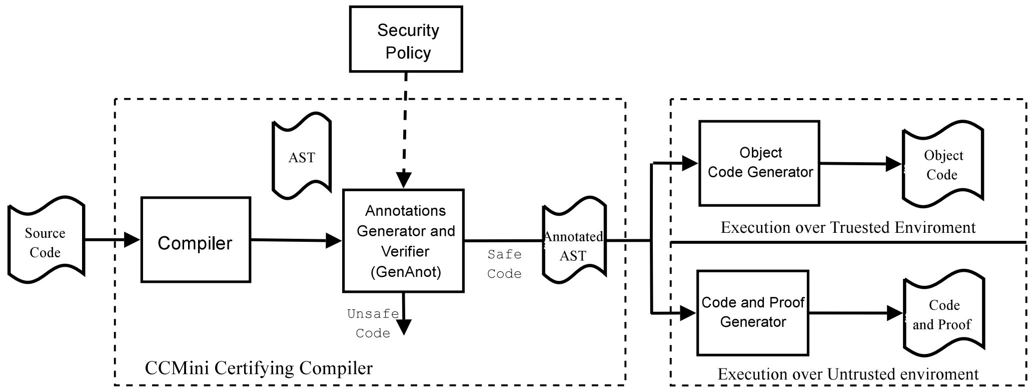

Figure 1: General Structure of the CCMini Certifying Compiler.

3 The Mini Language

In this section we define the structure and meaning of a Mini program. This source language has a syntax similar to the language C. Since Mini is a very simple language, but it possesses some features not present in C, we say that Mini is an extended subset of the C language.

Basically, a Mini program is a function that must have at least one parameter and to return a value. Both, the arguments and the return value, must be of a basic type (integer or boolean). Different to C, in which zero represents false and other number represents true, Mini has a boolean type. Moreover, Mini has unidimensional arrays whose elements can be of a basic type.

The language includes the conditional sentences `if` and `if-else`, the loop (`while`), and the `return` sentence. Different to C, the assignment is a sentence instead of being an expression. This design decision was made in order to simplify Mini's semantics. Blocks of sentences are marked by { and }. In each block variables can be declared and initialized. The visibility and live-range rules of variables are the same that in C.

Integer parameters can be bounded when they are declared. In order to do this, the declaration of a parameter includes the range of values accepted as arguments. For example, if the function profile is `int func(int a(0,10))`, then the value a_0 of the argument passed in the parameter `a` during any call to the function `func` will verify the condition $0 \leq a_0 \leq 10$. With this extension we pretend to increase the amount of bounded variables and expressions used in the programs.

The identifier names follow the usual rules, and so it is done by the number denotations and the boolean constants (`true` and `false`). Operators include arithmetical operators ($+$, $-$, $*$, and $/$), boolean operators (|| and &&), and relational operators ($==$, $>$, $<$, $\geq$, $\leq$, and $\neq$) with the usual meanings.

4 Security Policy

A security policy is a set of rules or that defines under which conditions a program is safe for execution. Since a program can be seen as the codification of a set of possible executions, it satisfies the security policy if a security predicate is true for the set of all possible executions of the program [11].

The security policy that we use in this work guarantees type and memory security. Furthermore, it guarantees that programs do not read uninitialized variables and do not access to undefined array positions. In the following subsections, we formalize our security policy.

4.1 Typing Rules of Security Policy

The security policy is formalized by a type system in order to avoid any ambiguity. There exist several advantages in the use of a type system to formalize a security policy. First, type systems are very flexible and easy of configure. Once types are selected, many interesting security properties can be obtained just by declaring the type of the values. Moreover, different security policies can be obtained by modifying the type system.

Expressions:	$e ::=$	$ae \quad \vert \quad be$
ArithExpressions:	$ae ::=$	$x \quad \vert \quad ae_1\ op\ ae_2 \quad \vert \quad -\ ae$
ArithOperations:	$op ::=$	$+ \quad \vert \quad - \quad \vert \quad * \quad \vert \quad /$
BoolExpressions:	$be ::=$	$b \quad \vert \quad be_1\ bop\ be_2 \quad \vert \quad not\ be \quad \vert \quad ae_1\ rop\ ae_2$
BoolOperations:	$bop ::=$	$\&\& \quad \vert \quad \vert\vert$
RelationalOperations:	$rop ::=$	$> \quad \vert \quad < \quad \vert \quad \geq \quad \vert \quad \leq \quad \vert \quad == \quad \vert \quad \neq$
Types:	$\tau ::=$	$bt \mid \text{array}\ (bt\ ,\ e)$
BasicTypes:	$bt ::=$	$\text{int}\ (Min,\ Max) \mid \text{boolean}$
Predicates	$P ::=$	$P_1 \wedge P_2 \quad \vert \quad e : \tau$
		$\vert \quad SafeRead(x) \quad \vert \quad SafeWrite(x_1, x_2)$

Figure 2: Syntax of Type Expressions

Figures 2 and 3 formally establish the security policy by defining the syntax of expressions, types, and predicates, and the typing rules for the Mini language. The *Expressions* syntactic category define all arithmetic and boolean expressions. Arithmetic expressions are defined by

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

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

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

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

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

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

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

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

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

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

Figure 3: Rules of the Type System.

ArithExpressions, while boolean expressions are defined by *BoolExpressions*. The syntactic category *Types* classifies the types in two: basic types (integers and booleans) and the array type. A value of type integer contains an integer value and two attributes, *Min* and *Max*, representing the range of possible values that it can take in some state. The array elements can be of integer or boolean type. The $SafeRead(x)$ and $SafeWrite(x_1, x_2)$ syntactic categories define when is safe to read from a variable x and when is safe to write on a variable x_1 the value x_2, respectively. The $e : \tau$ predicate means that the e expression is of type τ.

As usual, the type system is defined by a set of inference rules. In the rules shown in figure 3, basic types are tagged with an initialization flag. The flags $+$ and $-$ indicate an initialized and an uninitialized value, respectively. The **op_int**, **op_boolean**, **op_rel_boolean**, **minus_int** and **not_boolean** rules define the type of boolean, arithmetic, and relational operations. The **read_int** and **read_boolean** rules define when is safe to read from a variable, i.e., when the variable is initialized. The **read_array** rule defines when is safe to read from an array position. This rule indicates that is safe read from position i of an array a only if the value i is an initialized integer and its value is between zero and the array's length minus one ($0 \leq i < length(a)$).

The **write_int** and **write_boolean** rules show when it is safe to modify the value of a variable, and they establish that the modified variable is initialized after the modification. Furthermore, the **write_int** rule updates the range of the variable. The **write_array** rule guarantees that it is safe to write the value x in the position i of an array a only if x is of the same type that the array's elements and i is a defined integer whose possible values are between zero and the array's length minus one ($0 \leq i < length(a)$).

The modification of the variable's range by applying some arithmetic operator is expressed by means the **var_minor** and **var_mayor** rules. These rules establish new values for the minimum and the maximum, respectively.

5 The Intermediate Code: Abstract Syntax Tree

The intermediate code generated by CCMini is an abstract syntax tree (AST). This structure is an abstract representation of the source code, and allows us to perform control-flow analysis and data-flow analysis. Moreover, it can be used to generate several optimizations.

The structure of abstract syntax tree we use is similar to a traditional AST, although our AST allow us to include code annotations. These annotations show the state of the objects of the program, and they contain, for example, information about the initialization of variables, loop invariants, and variable's range.

Each sentence is represented by an AST. The nodes of a sentence, as well as the label that characterize it, contain information or references to the sentences that form it and a reference to the next sentence. Each expression is represented by a tree. We use two different labels to classify the different accesses to an array: `unsafe` and `safe`. These labels mean that it is not safe and that it is safe to access and use an element of the array, respectively. This technique allow us to eliminate some dynamic verifications on array accesses by just updating the label of the node.

6 The Compiler

This component of CCMini is a traditional compiler, whose input is Mini source code and it produces an abstract syntax tree. This module was implemented by following a typical compiler architecture [1] and, similar to most of the traditional compilers, it only reject source code that does not comply with the lexical and syntactic specifications imposed by the source language.

7 Annotations Generator: GenAnot

Code annotations are used by the verification process. These annotations consist of loop invariants, information about bounds of all variables, and the precondition and postcondition of the program. In the following paragraphs we will discuss in detail the generated information and the process used to generate it.

Different to other certifying compilers, the **GenAnot** component of CCMini simultaneously generates the code annotations and perform the required verifications. In order to do so, **GenAnot** performs control-flow and data-flow analysis on the AST described in Section 5. These flow analysis determine the initialization and range of all the variables used. The latter allows us to guarantee that a value is valid as an array index. Furthermore, in certain cases **GenAnot** can determine the precondition and postcondition of the program.

As it was mentioned before, **GenAnot** inserts run-time checks in program points where static analysis cannot guarantee the security. So, the mechanism that we use to verify that the code is safe is a combination of static verifications –in compilation time– and dynamic checks –in run-time–. We want to stress that in many cases these dynamic checks are necessary not only by the limitations of the static analysis, but also because some problems to solve are not computable. For example, the problem of guarantee that arrays are accessed respecting their bounds is reducible, in the general case, to the halt problem.

In the following sections we describe the procedures carried out by the **GenAnot** module to identify initialized variables and determine the ranges of the variables.

7.1 Identification of Initialized Variables

In order to generate the information needed to verify that each variable is initialized when it is referenced for first time, it is needed to analyze all the possible execution paths of the program.

GenAnot uses the abstract syntax tree presented on Section 5 to analyze statically the flow of each program. The initialized variables are identified by following the next rules:

1. At the beginning, just the parameters are initialized.

2. The constants are initialized values.
3. When a variable is declared, it is considered uninitialized.
4. An expression is initialized if all its operands are initialized.
5. When an initialized expression is assigned to a variable, this variable is now initialized.
6. In the conditional sentence, it is necessary to verify that all the variables used in both branches are initialized beforehand or that they are initialized in each branches. A variable is considered initialized for the rest of the program only if it was initialized in both branches.
7. In the loop case, it is necessary to verify that the variables used in the body of the loop are initialized beforehand or that they are initialized in the body. However, the variables that are initialized in the body of the loop are not considered initialized for the rest of the program, because the condition of the loop could be always false and its body never executed.

7.2 Identification of the Range of a Variable

In order to determine if it is safe to access to a position i of an array, **GenAnot** must to determine the value of the i expression. Obviously, in many cases an expression can take different values depending on the program flow. For example, let's consider the follow block of sentences:

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

In this case, the value of the variable i at point `(*)` could be 2 or 4. However, in our analysis, if the index of an array could be 2 or 4, then we can say that the index is valid for any value i such that $2 \leq i \leq 4$. So, we say that the variable i has the range $(2, 4)$.

Thus, in order to determine if an access to an array is safe or not, **GenAnot** must determine the range of the integer variables. In other words, **GenAnot** determines the bound of the integer variables. However, the determination of the bound of variables is not computable in general. Therefore, **GenAnot** only can solve some cases of identification of ranges, which are classified in two categories:

1. Identification of the range of variables that are not updated in a loop.
2. Identification of the range of inductive variables that are updated in a loop.

7.2.1 Identification of the range of variables that are not updated in a loop

Analyzing the abstract syntax tree, **GenAnot** identifies the ranges of variables that are not updated in a loop by following the rules:

1. At the beginning, just the parameters of integer type with declared ranges are bounded.
2. The integer constants are values with defined range, and they are bounded by their value.
3. When a variable is declared, we consider that its range it is not defined.

4. An e_1 *op* e_2 expression has a range only if all its operands have a defined range, and its range is defined by:

   $Max = Maximum(e_1.Min\ op\ e_2.Min, e_1.Min\ op\ e_2.Max, e_1.Max\ op\ e_2.Min, e_1.Max\ op\ e_2.Max)$

   $Min = Minimum(e_1.Min\ op\ e_2.Min, e_1.Min\ op\ e_2.Max, e_1.Max\ op\ e_2.Min, e_1.Max\ op\ e_2.Max)$

5. When we assign an expression with a defined range to a variable, the range of the variable is defined by the range of the expression. In the case that the expression does not have a defined range, the range of the variable is not defined.

6. In the case of a conditional sentence, we identify the range of variables in both branches. The range of a variable for the rest of the program is defined by the union of the ranges of such variable in both branches. The union of the branches is calculated as follows: for each variable x, if x is bounded in both branches, then the range of x is formed by the minimum and maximum value of the original ranges. If the variable has an undefined range in some branch, then the result is an undefined range.

This process can be applied to the following program:

```
int program (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;
}
```

As a result, we obtain the ranges for the last occurrence of the variables x, y, and z: $0 \leq x \leq 11$, $3 \leq y \leq 5$, and $3 \leq z \leq 16$.

7.3 Identification of the range of inductive variables that are updated in a loop

We made an informal study in order to determine how the programmers access to an array. We asked to programmers of the *Departamento de Computación* of the *Universidad Nacional Río Cuarto* (students, teachers, and researchers) to write programs Mini that use accesses to arrays. This informal study allowed us to determine that the more common way for accessing arrays is by using inductive variables. A variable is *inductive* if its value is updated inside of a loop only for a constant value in each iteration of the loop. A variable is called *linear inductive* if its value is increased or is decreased by a constant in each iteration of the loop.

For example, in the following program, that initializes an array in zero, the variable i is a linear inductive variable.

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

If the variable used in the loop condition is a linear inductive variable, and the loop condition is of either the form:

$$\textit{Variable Relationship BoundedVariable}, \text{ or } \textit{BoundedVariable Relationship Variable},$$

where *Variable* is initialized, and it has a constant range (i.e., the maximum and minimum are the same); and *BoundedVariable* is a variable that is bounded (i.e., it has a range[1]). Then, we can calculate the number of loop iterations by the following algorithm:

Let's suppose that the variable i is linear inductive initialized with the value i_{init}, and that k is an expression of constant value with range $(k.min, k.max)$, where $k.min = k.max$.

1. If i is an increasing linear inductive variable (i.e., i is used in the body of the loop in an operation $i = i + increase$, with $increase > 0$) and the loop condition is of the form $i \leq k$ or $i < k$:

$$\text{Let } k_m \text{ be} = \begin{cases} k.max + 1 & \text{if the loop condition is of the } i \leq k \text{ form} \\ k.max & \text{if the loop condition is of the } i < k \text{ form} \end{cases}$$

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

Then, the number of loop iterations is determined by the integer quotient:

$$iterations = \frac{k_m - i_{init}}{increase} + l$$

2. If i is a decreasing linear inductive variable (i.e., i is used in the body of the loop in an operation $i = i - increase$, with $increase > 0$) and the loop condition is of the form $i > k$ or $i \geq k$:

$$\text{Let } k_m \text{ be} = \begin{cases} k.min - 1 & \text{if the loop condition is of the } i \geq k \text{ form} \\ k.min & \text{if the loop condition is of the } i > k \text{ form} \end{cases}$$

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

Then, the number of loop iterations is determined by the integer quotient:

$$iterations = \frac{i_{init} - k_m}{increase * (-1)} + l$$

If we suppose that the i variable is linear inductive and initialized on i_{init}, and the number of times that the loop executes ($iterations$) can be calculated by using the previous algorithm, then we can calculate the ranges of the inductive variables by the following way:

1. For each increasing linear inductive variable the invariant is determined by:

$$v.min = i_{init} \qquad v.max = (increase * (iterations - 1)) + i_{init}$$

Whereas the postcondition is:

$$v.min = (increase * iterations) + i_{init} \qquad v.max = (increase * iterations) + i_{init}$$

2. For each decreasing linear inductive variable the invariant is determined by:

$$v.min = i_{init} \qquad v.max = i_{init} - (increase * (iterations - 1))$$

Whereas the postcondition is:

$$v.min = i_{init} - (increase * (iterations)) \qquad v.max = i_{init} - (increase * (iterations))$$

[1]Note that the constants are considered as bounded variables where the maximum and minimum are the same.

8 An Example

The program in Figure 4 was compiled by CCMini. Our certifying compiler could prove that all accesses to the array are safe, and it generated certified code which only dynamic verification consist of validating the precondition imposed to *limit*.

```
int bubble_sort (int limit (0,99))
{
  int [100] a;               /* declaration of an array a */
  int t,j, one i=0;          /* temporary variable t and indexes i,j */

  one=365-364;
  i=0;                       /* sort the array... */
  while (i<limit) {
     j=limit;                /* ...by using the bubble method */
     while (j>i) {
       if (a[j]<a[j-1]) {
         t=a[j];             /* ...and by swapping with a temporary variable */
         a[j]=a[j-one];
         a[j-one]=t;
       }
       j=j-one;
     }
     i=i+one;
  }
  return 0;
}
```

Figure 4: Example of program CCMini.

9 Conclusions and Future Work

CCMini is a certifying compiler that it can be used for the implementation of a certified code production environment. The developed prototype shows the possibility of using flow-control and data-control analysis techniques to implement certifying compilers.

Although static analysis techniques require a greater effort than the techniques applied by a traditional compiler –in the latter case, only verification of types and other simple analysis are applied–, the required effort is significantly smaller than the required for a complete functional verification.

An informal study of programs selected at random shows that generally inductive variables are used to access array positions. This result suggest that CCMini could determine the validity of the access in most of the practical cases, avoiding the need of insert dynamic checks. Moreover, when applied to the programs used for such study[2], CCMini determined the security of such programs without introduce and dynamic verification in most cases.

CCMini only reject a program when it can determine that the program's behavior is unsafe. In other cases, CCMini inserts dynamic verifications on all the programs points where the security is uncertain.

At this moment, the group is working on the creation of complete *certified code* environment based on the techniques implemented in CCMini. As a future work, the group will study the application of this work to more complex security policies, such as those that include pointer manipulation.

[2]The programs were obtained by requesting programmers outside of the CCMini implementation group to write programs based on array manipulation.

Another interesting future work would be to extend the developed prototype to accept some source language standard, and to use a publicly available functions library to obtain more statistically relevant data about the quantity of array accesses that the compiler can solve statically.

References

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

[2] 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), June 2000.

[3] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis". IEEE Software, pp. 42-51, January-February 2002

[4] 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.), March 2002.

[5] 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.), 1999.

[6] 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.), June 2002.

[7] 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.), May 1999.

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

[9] 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), June 1998.

[10] 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.), May 1996.

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

[12] 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.