

# Static Analysis: C Code Error Checking for Reliable and Secure Programming

Karthik.S, and Jayakumar.H.G

**Abstract**—This paper aims at presenting techniques to track the common programming and security flaws using static analysis of the C source code. These flaws may be serious errors or simple ones caused by programmer's carelessness and most of them may not be detected by the compilers. Manual reviewing of code for errors may take a lot of time especially if the code is big and it will also be very expensive, also simple flaws maybe overlooked. In this paper we present our techniques for automated error checking of C source code using static analysis.

**Keywords**—Error checking, secure programming, static analysis, verification and validation.

## I. INTRODUCTION

THERE are many techniques of eliminating flaws. The flaws maybe due to many factors like human carelessness or just ignorance. When the programs are large, then mechanical human review becomes time consuming and still many simple errors may go undetected. The present software development process does not involve techniques for prevention of such flaws

Our design suggest an alternative approach for error checking, rather than observing the program executions, we analyze the source code directly, thus checking for all possible program executions rather than test case executions. The approach is 'Do not wait till a bug is found to patch the code'. This is the basis of static analysis. Our goal is to create an easy to use tool for error checking that makes the job of the programmer easier so that he can concentrate mainly on the logic.

### A. Importance of Static Analysis

Many companies lose lots of money due to some trivial flaws in the programs. For example the company by name Sunsoft was about to lose some \$20 million due to a typo in the asynchronous I/O library.

According to research conducted by many firms most of the flaws in the softwares can be detected using static analysis like Bloor Research Ltd. stated in its report that 60% of the

Manuscript received on July 31, 2005.

Karthik S. is currently studying in Department of Computer Science, RV College of Engineering,, 8<sup>th</sup> mile, Mysore Road, Bangalore-560059, India (phone: 919886485443; email: karthiks25@hotmail.com).

Jayakumar H. G. is currently studying in Department of Computer Science, RV College of Engineering, 8<sup>th</sup> Mile, Mysore Road, Bangalore-560059, India (phone: 918026694279; email: jayakumarhg@gmail.com).

software faults in released software products could have been detected by means of static analysis.

Most of the security attacks exploit human weaknesses. When security attacks are analyzed, we find that the attackers do not find new kinds of flaws but rather repeatedly exploit well known flaws which can be detected using static analysis.

The root problem is that though the security vulnerabilities such as buffer overflow are well understood, techniques to avoid them are not codified into the development process. Our techniques aim at detecting such errors and likely security vulnerabilities using static analysis which is very useful in today's world where hackers and crackers are not hard to find.

## II. STATIC ANALYSIS TECHNIQUES

Static program analysis is a set of techniques to retrieve valuable information about the program without actually executing the program but by analyzing the code.

Static Analysis tools scan for various vulnerabilities within source code. Since the Static Analyzer does not know what the application or function is intended to do, it will not make assumptions that a developer or code reviewer might.

Various static analysis techniques are:

- 1) Semantic checking
- 2) Strong type checking
- 3) Memory allocation checking
- 4) Logical statement checking
- 5) Interface and include problem checking
- 6) Security checking

The stages involved in static analysis include control flow analysis and data flow analysis collectively called flow analysis.

### A. Control Flow Analysis

Control flow analysis involves in general of setting up a Control flow Graph i.e. a directed acyclic graph (DAG) to represent the control flow in a program. Each node in the DAG corresponds to a program point and edges from one node to another represent possible flow of control.

Control flow graphs for usual language constructs:

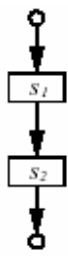


Fig. 2 (a) S1, S2

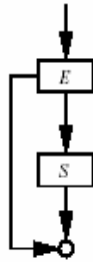


Fig. 2 (b) If E then S

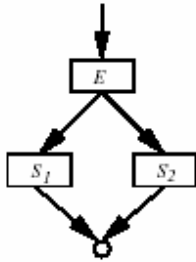


Fig. 2 (c) If E then S1 else S2

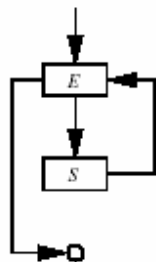


Fig. 2 (d) while E do S

Control flow graphs for more complex statements can be constructed inductively from control flow graphs of simple statements.

Each function in source is analyzed to build the control flow graph for each. Function calls are also represented as nodes in a control flow graph. However while traversing a control flow graph if one comes across a node representing a function call then control flow graph of corresponding function (if it exists) is also traversed.

### B. Static Data Flow Analysis

Data flow analysis determines different properties a variable can have when it takes different paths in the program. This information can be extracted from the program to check for potential flaws.

The nodes in the CFG are used to store information about certain properties of data like initialization of variables, allocation of a resource, references to variables etc.

Combining of data-flow information collected requires setting up and solving data-flow equations.

Consider a data-flow equation example for memory allocation:

$$\text{AllocOut [N]} =$$

$$\text{AllocGen [N]} \cup (\text{AllocIn [N]} - \text{AllocFree [N]}).$$

i.e. set of variables that are allocated at end of the block N is either internally allocated in that block or has allocated previously and is not deallocated in this block.

In the previous example the data flow out of a node was computed in terms of data flow into the node. The data flow into a node N is usually the data-flow out of the predecessor of node N. However it is possible that a node has more than one predecessor, in which case information flowing out of the predecessors need to be combined using an appropriate 'confluence operator'.

For example for initialization information reaching a node we need to apply union as confluence operator.

Consider the following code:

```
int a, x;
if (condition)
{
    a = initial_value;
}
```

```
x = a;
```

Though the variable 'a' has been initialized in the if statement it is not always guaranteed that a will be initialized before its access (when condition is false 'a' is uninitialized). On reaching the statement  $x = a$ ; we say that variable 'a' is uninitialized (at least in one of the paths).

## III. IMPLEMENTATION TECHNIQUES

### A. Tracking Bugs Caused by Typos

Many of the bugs caused due to typing mistakes or programmer ignorance can be tracked by modifying the C grammar specification to generate appropriate warning messages.

Some of the examples are:

- *Using = instead of ==*

```
int error_flag = 0;
if ( error_flag = 1 )
    print_error (error_msg);
```

We find to our dismay that this code always calls the print\_error function. So to warn whenever an assignment statement appears in conditional expressions we have the following partial grammar

<if-stat> :

if (<cond-expr>) statement

<cond-expr>:

<unary-expr> = <expression> {warn ( ) ;}

<assign-expr>

<expression> , <assign-expr>

/\*warn ( ) is used to generate a warning that = maybe used instead of ==\*/

In this grammar action corresponding to assignment is to warn of possible typo. If the programmer has intentionally used this construct then warning can be suppressed by using parenthesis around them.

Fall through in case statement

```
switch ( tokentype )
```

```
{
```

```
case INT: process_int();
```

```
case FLOAT: process_float();
```

```
case CHAR: process_char();
```

```

}
/* no break statement */
Even in this case the pattern is recognizable by the grammar
where the pattern is case statement followed by statement list
(excluding case statements) with no break statement. When
this is recognized by the grammar we can print out a suitable
warning.
Grammar can be modified as follows
<statement>: <non_case_statement>
| IDENTIFIER ':' <statement>
| CASE const_expression ':'
  <non_case_statement> ';' break ';'
| CASE const_expression ':'
  <non_case_statement> ';' { warn(); }
/* warn() used to generate a warning that break statement at
end of case statement maybe missing*/
;
<non_case_statement> :
<expression_statement>
| <compound_statement>
| <selection_statement>
| <iteration_statement>
| <jump_statement>
;

```

### B. Tracking Loop Errors

Loop condition does not vary

e.g.

```

while( i < n )
{
  printf( "%d", a[i] );
}
/* i is not incremented leading to an infinite loop*/
if( c == 'y' || 'Y' )
continue;
/* the condition always evaluates to true */

```

The latter condition has a constant in 'or' expression that always evaluates to true. We can detect whether a given condition always evaluates to true / false by calculating value of expression that can be determined at compile time.

The former example can be resolved as follows. We build a set of variables S in test condition.

We check in the loop body whether at least one variable v in S should be redefined in at least one path in the control flow graph. If this is not the case an appropriate warning message can be emitted.

### C. Tracking Format Bugs

Forgetting to put an ampersand (&) on arguments

Here's an example:

```

int x;
scanf( "%d", x);
/* & required to pass address to scanf() */

```

Using the wrong format for operand.

C compilers do *not* check that the correct format is used for arguments of a scanf () call. The most common errors are using the %f format for doubles (which must use the %lf format) and mixing up %c and %s for characters and strings.

We can implement tracking of format strings of library functions (printf, scanf, fprintf etc) only if the format string is a string literal otherwise it is difficult to determine contents of a character array. This is usually the case so we match each format specifier with the argument passed and check if it complies with type of argument. If there is a mismatch we can flag it as potential bug.

### D. Tracking Memory Related Errors

Resource handling errors like dereferencing uninitialized pointers, dereferencing null pointers, not freeing allocated memory, not closing open files etc are very common and serious flaws in a C Source code.

We wish to identify

- Dereferencing uninitialized pointers
- Dereferencing null pointers
- Memory Leakage
- Freeing Deallocated resource
- Dangling pointers

The approach here is to use flow analysis and constructing control flow graph containing nodes with information required like

Whether a variable is initialized or not: For this we maintain a set called NotInit. A variable x belongs to NotInit if it has not been defined after its declaration. Conservative Confluence operator for this set is Union

The target of a pointer variable: Target can be a malloced region or a address of a variable like &id or another pointer. We define a function called Target( p ) which returns a set of targets of pointer p. A target of a pointer is of form REF(id) or ALLOC(x) where x is a unique number to identify different allocations.

Whether a pointer variable is null or not: We maintain a set called NullPtr. A variable x belongs to NullPtr if its value is possibly null. Conservative Confluence operator for this set is Union operator.

Each of the sets defined above is specific to a node in CFG.

Now we need to set up data flow equations to compute value of these set for each node N

$$NotInit[N] = U_{M \text{ pred}(N)} NoInit[M] - Def(N)$$

Here Def(N) are set of variables defined in node N

A pointer variable x in Def(N) if

```

ptr = &id
ptr = constant
ptr = function()
ptr = ptr1 if ptr1 ∉ NotInit[N]

```

$$NullPtr[N] = U_{M \text{ pred}(N)} NotNull[M] - NZDef(N)$$

*M pred N*

Here NZDef(N) are set of pointer variable which have non-null definition.

A non-null Definition is

ptr = constant-non-zero then ptr ∈ NZDef

ptr = &id then ptr ∈ NZDef

ptr = ptr1 where ptr1 ∈ NZDef

Here we have not included ptr = function() since we make an assumption that any function returning pointer could return a null-pointer which is valid in most of the cases.

Target(p) The reason that target of p is a set is that it is possible that though different paths p can have different targets. So updating rules for the set Target(p) are:

a) If ptr = &id then

Target(ptr) = { REF(id) }

b) If ptr = ptr1 then

Target(ptr) = Target(ptr1)

c) If ptr = malloc then

generate a new unique value for x

Target(ptr) = { ALLOC(x) }

d) If free( ptr ) then

If ALLOC(x) which now belongs to Target( ptr ) remove all occurrences of Alloc( x ) from each of Target(p). This is required to update all pointers having reference to the same memory block being deallocated.

Now let us see how using these information we can detect the common memory allocation flaws.

#### Dereferencing or Using an Uninitialized Pointer

If we come across dereferencing of a pointer variable ptr in a node N

If ptr ∈ NotInit[N] then an uninitialized pointer is being dereferenced. So a warning can be generated.

#### Dereferencing of Null Pointers

If we come across a dereferencing of a pointer variable ptr in a node N

If ptr ∈ NullPtr[N] then possibly a null pointer was being dereferenced. So a warning about a possible null dereferencing can be given out.

#### Memory Leakage

If we come across a pointer definition such as ptr = malloc(..) or ptr = AllocatingFunction() like fopen and such, we check the Target(ptr).

If

Alloc(x) ∈ Target( ptr ) for some x and

Alloc(x) ∉ Target( ptr1 ) for all ptr1 ≠ ptr

then it is a potential point in program where a reference to previously allocated block is lost.

So a warning about a memory leak can be given out.

#### Deallocating a Deallocated Resource

If we come across a deallocating function such as free( ptr ) or fclose( ptr ) then we need to do the following check

If Alloc( x ) ∉ Target( ptr ) for all x

Then a DeallocatingFunc( ptr ) is probably trying to deallocate an already deallocated pointer.

#### Use of Dangling Pointers

If we come across a dereferencing of a pointer variable ptr in a node N then we need to check

If Target( ptr ) = ∅ then we know that we are trying to dereference a pointer whose target no longer exists. So we may generate a warning about use of dangling pointers.

Examples:

Memory leakage

```
int *Array;
```

```
for(i = 2; i < n ; i++)
```

```
{
```

```
    Array = malloc(sizeof(int)*i);
```

```
    ...
```

```
    initialize(Array);
```

```
    do_something(Array);
```

```
    ...
```

```
}
```

```
free(Array);
```

```
/*
```

Note that Array is not freed within loop, so if the loop executes more than once then Array will point to a newly allocated block and reference to the old block is lost leading to memory leakage

```
*/
```

Fig. 3 (a) A sample code

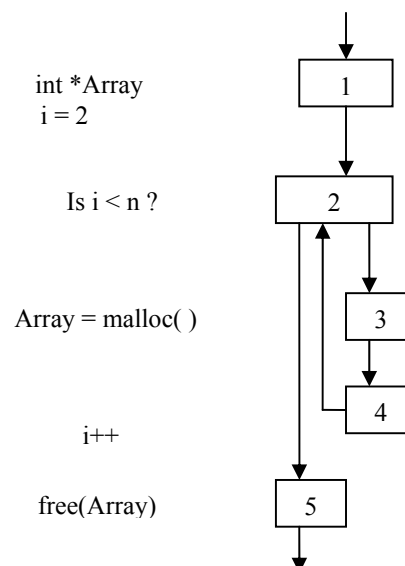


Fig. 3 (b) CFG for code in Fig. 3 (a)

Now solving the data flow equations for NotInit, Target( p ) for the above CFG.

We note that node 2 has two predecessors' node 1 and node 4. However while evaluating the dataflow equations we come across a cycle where in values in node 4 depends on node 2 which in turn depends on node 4. To resolve this we use a modified depth first search traversal where in the only modification being that we traverse the back edges exactly once.

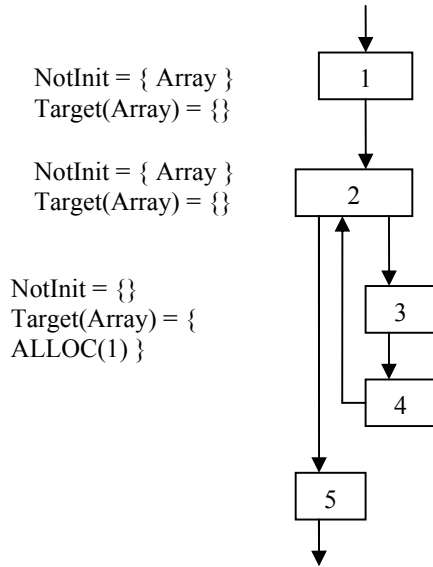


Fig. 3 (c) CFG for code in Fig. 3 (a) along with partially computed values of the set NotInit and Target sets

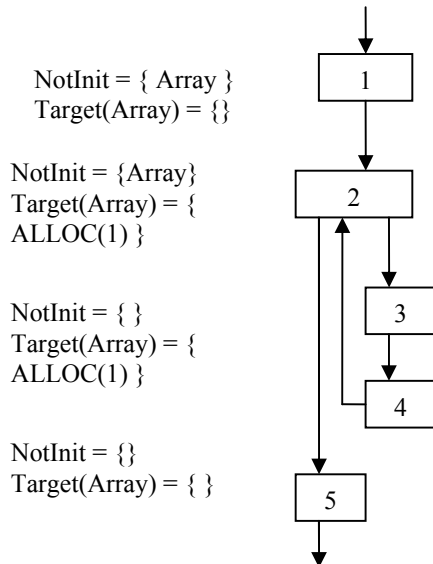


Fig. 3 (d) CFG for code in Fig 3 (a)

In the above example, initially we consider node 1 as the only predecessor of node 2 and calculating values for node 2, 3 and 4. Then the values of node 4 act as predecessor for node 2 (Traversing the back edge 4-2) and values are computed for

node 2, 3 and 4 again.

Once the values of the set NotInit, Target and Null Ptr are calculated for each node in CFG we can apply the various tests described in this section.

On Applying the Memory leak test we find that in node 4 we have Alloc (1) ∈ Target (Array) and Array is being redefined. So it is a clear case of memory leak.

#### E. Tracking Data Faults

Some of the potential bugs involving data are:

Unused variables

Some variables are declared but never used in the program.

Use of uninitialized variables as r-value

int x;

if ( x == VAL)

found = 1;

/\* x is uninitialized and is being used a r-value \*/

The above faults can be tracked by using the method used for pointer variables. We can make use of NotInit set for all variables. The updating of NotInit set follows the equation given previously.

Whenever a variable v is accessed as r-value in node N then we check if v ∈ NotInit [N]. If it is so then v is being used before its initialization and we can generate a warning.

Returning pointer to a local variable

```
char *copy(char *src)
```

```
{
```

```
    char str[MAX_CHAR];
```

```
    char *p = str;
```

```
    while( *p++ = *src++);
```

```
    return p;
```

```
}
```

/\* this function returns a pointer to a variable whose lifetime ends with the function \*/

We can detect returning pointer to a local variable by checking if REF (id) ∈ Target (p) where id is automatic variable and p is being returned by the function.

#### IV. DRAWBACK

The main drawback is that these techniques may sometimes generate false warnings, like the programmer might have really intended to use certain constructs but our techniques may detect them as bugs. The main goal is to construct a tool which produces useful results so that the programmer can avoid long sessions of debugging using this automated error checking tool.

#### V. PREVIOUS WORK

There exist softwares like *lint*, *splint* which perform error checking of C source code but many serious flaws described in this paper, for example the loop errors described in section 3.2 and memory leakage flaws described in section 3.4, are not detected and other static analysis tools are commercial tools and not open source tools. Therefore we have presented

our own techniques to detect flaws in source code in this paper.

## VI. CONCLUSION

- Static analysis is not a replacement for conventional testing nor can it completely eliminate manual review but rather it is an add-on which automates analysis of certain kinds of faults.
- Static analysis is a promising technique for detecting likely software vulnerabilities, helping programmers fix them before software is deployed rather than patch them after attackers exploit the problem.
- There is an immediate need for better open source static analysis tools, hence we are implementing the techniques describing in this paper to develop a new tool.

## VII. FUTURE IMPROVEMENTS

- Detect null dereference in recursive structures
- Stack Overflows, Infinite recursion, Violation of segment rights
- Extensible checks, customizable & mostly global
- Validation through annotations.

## REFERENCES

- [1] Aho, A.V, Ravi Sethi and J.D Ullman, "*Compilers Principles Techniques and tools*", Pearson Education, 1997.
- [2] Kernighan B.W and D.M Ritchie, "*The C programming language*", Prentice Hall Inc., 2<sup>nd</sup> Edition, 1998.
- [3] Peter Van Der Linden, "*Expert C programming – Deep C Secrets*", Prentice Hall Inc., 1998
- [4] David Evans and David Larochelle. "*Improving security using extensible lightweight static analysis*", IEEE Software, January/February 2002.
- [5] Michael I. Schwartzbach, "*Lecture Notes on Static Analysis*", BRICS, Department of Computer Science, University of Aarhus, Denmark.