Carraybound: Static Array Bounds Checking in C Programs Based on Taint Analysis

Fengjuan Gao
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
fjgao@seg.nju.edu.cn

Tianjiao Chen
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
chentj@seg.nju.edu.cn

Yu Wang
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
yuwang@seg.nju.edu.cn

Lingyun Situ
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
situlingyun@seg.nju.edu.cn

Linzhang Wang
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
lzwang@nju.edu.cn

Xuandong Li
State Key Laboratory of Novel Software Technology
Nanjing University
Nanjing, 210023, China
lxd@nju.edu.cn

ABSTRACT

C programming language never performs automatic bounds checking in order to speed up execution. But bounds checking is absolutely necessary in any program. Because if a variable is out-of-bounds, some serious errors may occur during execution, such as endless loop or buffer overflows. When there are arrays used in a program, the index of an array must be within the boundary of the array. But programmers always miss the array bounds checking or do not perform a correct array bounds checking. In this paper, we perform static analysis based on taint analysis and data flow analysis to detect which arrays do not have correct array bounds checking in the program. And we implement an automatic static tool, Carraybound. And the experimental results show that Carraybound can work effectively and efficiently.

CCS Concepts

•Software and its engineering → Software verification and validation; Software defect analysis; Software testing and debugging; •Security and privacy → Vulnerability scanners;

Keywords

Array index out-of-bounds; Static analysis; Taint analysis; Data flow analysis;

1. INTRODUCTION

In order to speed up execution, C programming language never performs automatic bounds checking. But bounds checking is absolutely necessary to be performed to ensure that a variable is within some bounds before it is used. When an array is used in the program, the index of the array must be within a certain range: less than the size of the array and not less than zero. If the index is out-of-bounds, further execution is suspended via some sort of error. Array index out-of-bounds have two scenes: a read operation or a write operation. A read operation scene means the array used as the right operand of an assignment operation. A write operation scene means the array used as the left operand of an assignment operation. For a read operation scene, the array index out-of-bounds may lead to unpredictable consequences. Because the bounds of an array are established when the array is defined. When the index is out-of-bounds, the variable may be returned a random value by the system, which may lead to unpredictable consequences. For a write operation scene, the array index out-of-bounds may lead to even worse consequences. For example, the loop variable stored next to the array may be rewritten by the out-of-bounds index, which may very likely lead to an endless loop. Moreover, array index out-of-bounds always leads to buffer overflows. And the empirical study in [34] shows that about 31% buffer overflows are caused by array index out-of-bounds. Buffer overflow is one of the most common type of software vulnerabilities. The buffer overflow vulnerability provides a way for attackers to corrupt data, crash the program, or execute malicious code. So it's absolutely necessary to perform bounds checking. But programmers always miss the array bounds checking or do not perform a correct array bounds checking. In this paper, we perform array bounds checking based on taint analysis and data flow analysis to find the indexes that have not been checked against the bounds of the array.

Static analysis methods and dynamic testing approaches have been proposed to perform array bounds checking. Dynamic methods always rely on the completeness of the test cases, which lead to these methods cannot achieve high coverage of the program. Static code analysis methods are commonly used to scan the program source codes to detect certain vulnerabilities. And data flow analysis is commonly

Internetware '16, September 18 2016, Beijing, China

DOI: http://dx.doi.org/10.1145/2993717.2993724

used in static code analysis, which is a technique for gathering information about the the flow of data values across basic blocks.

In this paper, we present a static analysis framework, Carraybound (C array bound), to perform array bounds checking based on taint analysis and data flow analysis to find the indexes that have not been checked against the bounds of the array in C programs. First, Carraybound performs taint analysis to decide all the variables' taint state. Then, Carraybound locates all statements containing array expressions and constructs array boundary information. Next, Carraybound performs backward data flow analysis to traverse the control flow graph to check whether there is any statement ensuring that the index is within the array bounds.

2. BACKGROUND

In this paper, we performs static analysis based on taint analysis and data flow analysis, which rely on control flow graph and call graph. In order to make our method clearer and easier to understand, we will explain the corresponding concepts of these techniques before we present our own method.

2.1 Control Flow Analysis

Control flow analysis, a static analysis technique, is used to determine the control flow of a program. A control flow graph (CFG) is a directed graph where each node represents a basic block and each directed edge represents a control flow[3]. A basic block is a straight-line piece of code with only one entry and one exit. A control graph shows how events are sequenced in the program. The control flow graph is very commonly used in static analysis techniques. Control flow analysis is the base of data flow analysis.

2.2 Call Graph

To perform static analysis on a given C program, we need first to fetch the runtime relationships between procedures in the program, namely the call graph of the program. A call graph is a flow graph where each node represents a procedure and each edge (f, g) indicates that procedure f calls procedure g. A call graph gives an inter-procedural view of a program. The call graph is a useful data representation for control and data flow programs which investigate interprocedual communication [25].

2.3 Data Flow Analysis

Data flow analysis is commonly used in static code analysis, which is a technique for gathering information about the the flow of data values across basic blocks. Data flow analysis is based on a control flow graph. A simple way to perform data flow analysis of a program is to set up data flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, i.e., it reaches a fixpoint [19, 18]. There are two kinds of data flow analysis—forward data flow analysis and backward data flow analysis. Forward data flow analysis, which starts from the entry node and ends at a targeted node, is along the normal path of the execution. And the exit state of a block is a function of the block's entry state. The function actually is a composition of effects on the entry state by the statements in the block. A block's entry state is a combination of its predecessors' exit states. On the contrary, backward data flow analysis, which starts from the targeted node and ends at the entry node, is in reverse of the direction of the edges in the control flow graph. And the entry state of a block is a function of the block's exit state. The function actually is a composition of effects on the exit state by the statements in the block. A block's exit state is a combination of its successors' entry states.

2.4 Taint Analysis

Taint analysis is a commonly used technique to detect vulnerabilities in programs. Programs may be insecure if an attacker inputs some malicious data into the program. These data that are influenced by the external inputs are marked as tainted in taint analysis. External inputs include inputs by users or files, arguments of main function. Taint analysis attempts to identify variables that have been tainted with user controllable input and traces them to possible vulnerable function (a sink). If the tainted data gets passed to a sink without first being sanitized it is flagged as a vulnerability. There are static taint analysis and dynamic taint analysis. Dynamic taint analysis need to execute the program and the coverage of the source code cannot be guaranteed. And static taint analysis relies on the analysis of abstract syntax tree of the program and does not need to actually execute the program. So static taint analysis can achieve higher coverage of the source codes than dynamic taint analysis. But static taint analysis may have false positives and false negatives for lack of runtime information.

3. ARRAY BOUNDS CHECKING

Our array bounds checking method is based on static taint analysis and data flow analysis, which are based on call graph and control flow graph of the program.

Our whole analysis is based on Clang's Abstract Syntax Tree (AST). We first build a call graph and a control flow graph based on the program's AST. Our interprocedural analysis relies on the call graph of the program to obtain calling relationships between procedures in the program. Then, we will perform a Depth-First-Search (DFS) analysis on the call graph to generate a call graph without any recursion.

As Figure 1 shows, we first use clang to compile the source code of the given program to get its AST files. Then, based on the AST files, we will build CFG and call graph of the program. Next, based on the CFG and call graph, we will perform taint analysis to get all expressions' taint states. Then, we will locate the array expressions and construct array's boundary information. Then, we will perform array bounds checking through backward data flow analysis. Finally, we get all the array bounds checking reports.

3.1 Case Study

In this section, we will give a small case to initially illustrate our method. The CFG of the example is shown in Figure 3. And the number after the colon is the entrance statement's line number.

Array boundary information construction: In *function*, there is an array expression *arr[i]* on line 17. As we can see, *arr*'s buffer size is *10*, so we construct the array boundary information—*line 19,i<10*. And there is an array expression *p.arr[i]* on line 17. This is a member variable of a structure and we construct the array boundary information—*line 20,i<15*.

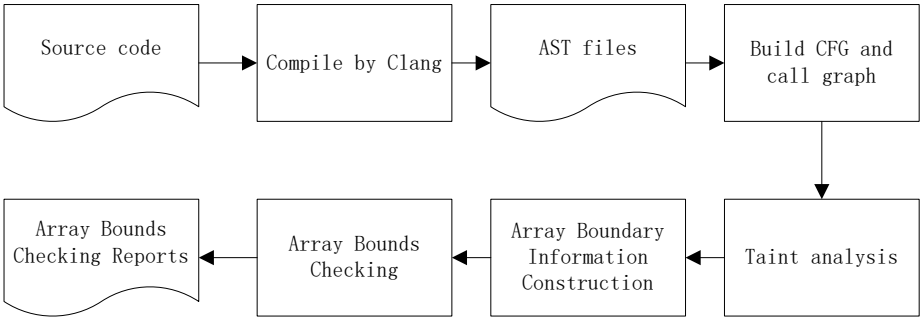

Figure 1: Framework of Carraybound

Table 1: Taint Propagation Rules

Expression Name	Expression Type	Handling Method	Comments
Integer Constant	Stmt::IntegerLiteralClass	return TaintValue()	Untainted by default
Float Constant	Stmt::FloatingLiteralClass	return TaintValue()	Untainted by default
Type Transformation	CastExpr	visitStmt(subExpr)	Handle the subexpression recusively
Parenthesis Expression	ParenExpr	visitStmt(subExpr)	Handle the subexpression recusively
Declared Variable	DeclRefExpr a	return taintValue(a)	Return its taint value directly
Declaration statement	DeclStmt int a=b;	taintValue(a)=taintValue(b)	Similar to assignment statement
Unary Operator	UnaryOperator	visitStmt(subExpr)	Handle the subexpression recusively
Binary Operator: assignment	BinaryOperator: x=y	taintValue(x) = taintValue(y)	Include +=, *=, etc
Binary Operator: not assignment	BinaryOperator: x op y	return visitStmt(x)+visitStmt(y)	Return taint if any operand is taint
Conditional Expression	ConditionExpr	result=taintValue(b)+taintValue(c)	Similar to assignment statement
Array	ArraySubscriptExpr	Left: return visitStmt(E->getBase(),inLeft) Right: return TaintValue((*visitStmt(E->getBase(),inLeft)) +(*visitStmt(E->getIdx(), inLeft)))	Taint as a whole
Struct	MemberExpr	return visitStmt(E->getBase(),FD)	Taint as a whole
If statement	IfStmt	StmtValue=visit(IfStmt)	
While statement	WhileStmt	StmtValue=visit(WhileStmt)	
Function call	CallExpr	return vistCallExpr(callExpr,FD)	Taint from arguments to parameters
Return Expression	ReturnStmt	visitStmt(subExpr)(if not void)	return untainted when Void
Other statement		return untainted	Untainted by default

Array bounds checking: We perform array bounds checking starting from block 2 which contains array expressions. First, block 2 will be traversed. Then we get block 2's predecessors—block 4. Next, we get block 4's successors (block 2 and block 3) to get block 4's in-state—*line 17,i<10;line 18,i<15;*. And as the second successor (block 2) of block 4 is on the false direction, so the if condition is *i<15*. Thus a corresponding boundary check is found for line 18 and the condition *line 18,i<15* will be removed from the array boundary information. Namely, block 4's out-state is *line 17,i<10;*. When encountering the for statement in block 5, the array boundary information will be updated to *line 14,n<11;*. When encountering the assignment statement in block 6, the array boundary information will be updated to *line 14,m<11;*. So when encountering function's entry and the array boundary information is not empty. If the configured detection depth is 1, then we will report this array bounds checking warning. Else, the interprocedual backward data flow analysis will be performed.

3.2 Taint analysis

In this paper, we use the static taint analysis method. Our taint analysis includes intraprocedual taint analysis and interprocedual taint analysis. Intraprocedual taint analysis perform analysis on each single function in the program. And interprocedual taint analysis starts at the entry function (usually the main function), and analyze the whole program.

Taint sources: External inputs include inputs by users or files, arguments of main function.

Intraprocedual taint analysis: First, a sequence of functions bottom-up are fetched according to the inverse topological sort of the call graph. For each function, we preserve the taint states of each variable in the function. The taint state of each variable can be tainted, untainted or related to the parameters of a function. Then we perform a taint analysis on each function by using a forward data flow analysis. For a given function, we perform a taint analysis on each statement in each block in the function. For each basic block in a function, its in-state is the combination of its predecessors' out-states, and its out-state is a function of its in-state with the effects of the statements in it. For a function containing loops in it, each basic block's in-state and out-state will be calculated iteratively until the states

```
1  typedef unsigned int UINT32;
2  typedef struct
3  {
4      UINT32 var;
5      UINT32 arr[15];
6  }myStruct;
7  myStruct p;
8  UINT32 arr[10];
9
10 void function(UINT32 m)
11 {
12     UINT32 n;
13     n=m;
14     for(UINT32 i=0;i<n;i++)
15     {
16         if(i >= 15) break;
17         arr[i]=0;
18         p.arr[i]=0;
19     }
20 }
```

Figure 2: Example

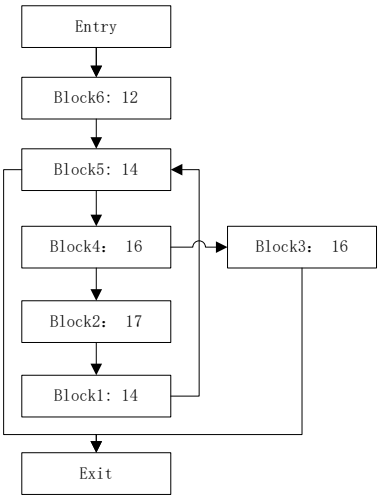

Figure 3: CFG of Example

do not change. And the function's taint state is the same as the taint state of the exit basic block's out-state. In this way, we can obtain the relationship between an expression to the corresponding function's parameters. And each function only needs to be analyzed only one time even if the function may be called many times by other functions.

For each statement Carraybound encounters, Carraybound will handle the statement according to the taint propagation rules listed in Table 1.

Interprocedual taint analysis: First, we will obtain a directed acyclic graph (DAG) by depth-first-search (DFS) traversing of the call graph and removing the cycles, namely a non-recursive call graph. DAG has a topological ordering, a sequence of the vertices such that every edge is directed from earlier to later in the sequence[16, 29]. Interprocedual taint analysis starts at the entry function and analyzes the program top-down according to a breadth-first-search (BFS) order. The parameters of the entry function are marked as tainted data. Each expression's initial taint state is untainted. Each variable's taint state will be calculated. We traverse each function according to topological order on the non-recursive call graph to make taint spread from top function's parameters to their related function's parameters. We get each calling edge in the call graph, and spread the caller's real arguments' taint values to the callee's parameters. If there are multiple functions call the same function, then the callee function's parameters' taint values are the combination of all its callers' real arguments' taint values. In this way, we can obtain the taint information of each function's parameters.

Sinks: As we need to perform array bounds checking next, the sinks are the statements including the use of arrays.

After the intraprocedual taint analysis, we know the taint relationships between the expressions in the function and the parameters of the function. After the interprocedual taint analysis, we know the taint relationships between the parameters of each function and the parameters of the entry function. To query the taint state of an expression in a function, we only need to combine the taint relationships between the expressions in the function and the parameters of the function with the taint value of the function's parameters.

3.3 Array Boundary Information Construction

To confirm whether the program has missed some array bounds checking, we need to know where the array is and what boundary conditions should be checked to avoid index out-of-bounds. To construct the desired array boundary conditions, we perform array localization, array base and index extraction, array size extraction and array bounds checking condition construction based on the AST of the program and the results of above taint analysis.

Array localization: Array bounds checking should be performed in every function of the program. Given a function, we first get the corresponding CFG. Then we traverse each statement in each block on the CFG. If the statement class is "ArraySubscriptExpr", an array has been located. The array can be a multidimensional array or the index of an array can also be an array. The array base can be a structure array whose index can also be an array. The array can be a structure variable's array member variable which can be multidimensional.

Array base and index extraction: Once an array is located, we need to construct its bounds checking condition. To do this, we need first to extract the base and indexes of the array. With the support of the APIs in Clang, we can get the last index of an array each time of the iterations. And the array base can be fetched in the last iteration. The array base will be used for array size extraction and the array indexes will be used for array boundary conditions construction.

Array size extraction: Based on Clang's AST, we can extract the declaration size of an array. We can extract an array's declaration size when it's a common array variable or structure array variable. But we cannot extract an array's declaration size when the array variable is an external variable since the AST will not preserve an external variable's size information. Once an external array variable is encountered, we just skip it. It is worth noting that we will get a list

Algorithm 1 Intraprocedual Backward Data Flow Analysis Algorithm
IntraBDFA(*CFG*, *ArrayBoundaryConditionSet*)

Input:
CFG, *ArrayBoundaryConditionSet*
Output:
ArrayBoundaryConditionSet

```
1: for each block in CFG do
2:   if block->pred_begin()==block->pred_end() then
3:     Entry=block; //entry point
4:   end if
5:   InState[block]=NULL;
6:   OutState[block]=NULL;
7: end for
8: blockSet=NULL;
9: for each ArrayBoundaryCondition in
   ArrayBoundaryConditionSet do
10:  arrayBlock=ArrayBoundaryCondition.getBlock();
11:  arrayStmt=ArrayBoundaryCondition.getStmt();
12:  ArrayBoundaryConditionSet=HandleBlock(
     arrayBlock,ArrayBoundaryConditionSet,arrayStmt)
13:  InState[arrayBlock]=ArrayBoundaryConditionSet;
14:  for each pred of arrayBlock do
15:    blockSet.push(pred);
16:  end for
17: end for
18: while !blockSet.empty() do
19:  block=blockSet.top();
20:  blockSet.pop();
21:  for each succ of block do
22:    OutState[block]=Union(InState[succ]);
23:  end for
24:  ArrayBoundaryConditionSet=HandleBlock(
     arrayBlock,ArrayBoundaryConditionSet,NULL)
25:  if ArrayBoundaryConditionSet!=InState[block]
     then
26:    for each pred of block do
27:      blockSet.push(pred);
28:    end for;
29:    InState[block]=ArrayBoundaryConditionSet;
30:  else
31:
32:  end if
33: end while
34: return ArrayBoundaryConditionSet;
```

of size number since the array may be a multidimensional array.

Array boundary conditions construction: As we already know the specific array's indexes and size, we can construct array boundary conditions. Each index should be less than the corresponding array size, like index $<$ size. Further more, if index is "signed" variable, index should also be not less than zero, like index ≥ 0.

Before constructing the array boundary conditions, we fist query whether the index is tainted. If the index is a constant, the index will be ignored. If the index is tainted, we construct the array boundary conditions with the index and the array size (like index $<$ size).

To support *Data Flow Analysis*, we preserve more information about the array subscript: which function, block and statement it is in, and its location, etc. All these information will be packaged in a self-defined structure and be used as the inputs of data flow analysis.

3.4 Array Bounds Checking

In this paper, since we want to find whether there is any check on the array boundary conditions in front of the array statement, we start at the array statement and perform a backward data flow analysis on the control flow graph of the program to search corresponding array bounds checking. If a corresponding array bounds checking exists, the array boundary conditions will be removed from our targets. Otherwise, the data flow analysis will be continued until a corresponding array bounds checking has been found or the function entry has been reached.

As stated in section *Array Bounds Information Construction*, we already extract and construct the array boundary conditions for each array index in a given function. Next, we perform a backward data flow analysis based on the control flow graph of the program to check whether each array boundary condition has been included in the program. We first perform an intraprocedual data flow analysis in the function according to Algorithm 1 and Algorithm 2. And if no array bounds checking has been found, we will perform an interprocedural data flow analysis based on the call graph according to Algorithm 3. And the depth to traverse in the call graph can be configured in a configure file by the user.

For a given function, we first get its control flow graph. Then, we traverse the CFG to find the entry point of the CFG. What we concern about is whether an array boundary condition has been included in any statement in any block. The in-state and out-state of each block are initialized as an empty set. Then each block in the function will be traversed to update its in-state and out-state.

For each block, we use a postorder iterative algorithm to solve the data flow equations. This step will be repeated until we reach a fixpoint—when the in-states and the out-states do not change.

For each array boundary condition, we first get its corresponding block and statement. Our backward data analysis start at the block where the array boundary condition is and ends at the entry point of the function where the array boundary condition is (during introprocedual data flow analysis). Of course, if all the array boundary conditions have been found in the function before the entry point having been reached, the backward data analysis will be terminated. Then, we collect all the statements before that array statement in the block where the array stays in. All these collected statements are pushed into a stack in a forward order. Next, the top statement in the stack will be popped and handled. Namely, the statements are handled in a backward order.

For each statement encountered, we handle it according to the rules listed in Table 2:

If the statement is an assignment statement and the array boundary condition's index is exactly the expression on the left of the assignment operator, the array boundary condition's index will be replaced with the expression (we call new index in the following text) on the right of the assignment operator. And if the new index is a constant, we will check whether the constant is less than the corresponding array size. If the constant is less than the corresponding array size, the corresponding array boundary condition will

Table 2: Backward Data Flow Analysis Handling Rules

Expression Type	Expression Pattern	Handling Method
Assignment with constant	A=const	Replace A with the constant, check whether condition can be satisfied
Assignment with modulus operator	A=expr%const	If const≤arraySize, condition will be satisfied
Assignment with other expression	A=expr	Replace A with the expression, update the condition
Compound assign operation with modulus operator	A%=const	If const≤arraySize, condition will be satisfied
if statement	if(A<const) or if(const>A)	If const≤arraySize, condition will be satisfied
for statement	for(...;A<const;...) or for(...;const>A;...)	If const≤arraySize, condition will be satisfied
for statement	for(...;A<B;...) or for(...;B>A;...)	Replace A with B, update the condition
while statement	while(A<const) or while(const>A)	If const≤arraySize, condition will be satisfied
while statement	while(A<B) or while(B>A)	Replace A with B, update the condition

be removed from the array boundary conditions set. Otherwise, the data flow analysis will continue. And if the new index is an expression containing modulus operator with a constant divisor, we will check whether the constant is less than or equal to the corresponding array size. If the constant is less than or equal to the corresponding array size, the corresponding array boundary condition will be removed from the array boundary conditions set because the modulo operation will ensure that the new index is not larger than the corresponding array boundary. And if the new index is other expression, the corresponding array boundary condition will be updated. In the following data flow analysis, we will concern whether the new array boundary condition is satisfied instead of the old one.

If the statement is a compound assign operation compounded by a modulus operator with a constant divisor and the dividend is exactly the array boundary condition's index, we will check whether the constant is less than or equal to the corresponding array size. If the constant is less than or equal to the corresponding array size, the corresponding array boundary condition will be removed from the array boundary conditions set.

If the statement is an "if" statement, a "for" statement, or a "while" statement, the array boundary condition's index is exactly the operand of the condition, the other operand is a constant and the operator is consistent with the array boundary condition's operator, we will check whether the constant is less than or equal to the corresponding array size. If the constant is less than or equal to the corresponding array size, the corresponding array boundary condition will be removed from the array boundary conditions set. If we do not find a check in the "for" condition or the "while" condition, then we will check whether the targeted index is a loop variable. If the "for" or "while" condition matches the pattern *index*<*var*, then the array boundary condition will be updated by replace the *index* with *var*. Namely, we will check whether *var*<*array size* is included before the "for" or "while" statement.

After all the statements have been traversed in a block B1, the in-state of the block B1 will only include the array boundary conditions having not been checked in the block B1. To continue to perform backward data flow analysis, the predecessors of the block B1 will be traversed. For each predecessor (B2) of the block B1, in order to get its out-state, which is a combination of its successors' in-states, we need to get all its successors and update the out-state of the block B2 by combining these successors' in-states. Then each predecessor of the block B1 (namely B2), will be iteratively traversed in postorder by handling the statements according to the rules listed in Table 2. This step will be repeated until we reach a fixpoint—when the in-state and the out-state do not change.

If the array boundary conditions are not found when the function's entry point have been arrived, an interprocedual backward data flow analysis will be performed by traversing the caller functions of the array's function (function A). First, we need to get all the caller functions of the array's function according to the call graph of the program. For each caller function (function B), we traverse its CFG and find the call statement that calling function A. Then the function A's declaration parameters will be fetched. For each array boundary condition, if its index is the same as one of the function A's declaration parameters, the corresponding real argument will be fetched to construct a new array boundary condition. The new array boundary conditions will be used in the caller function B. And the caller function B will be traversed by performing a backward data flow analysis. This interprocedual backward data flow analysis procedure will continue until all the array boundary conditions are found or the configured detection depth is reached.

3.5 Array Bounds Checking Reports

After the data flow analysis, each array boundary condition in the targeted set has been checked whether there is any corresponding array bounds checking included in the program. And only the array boundary conditions having no corresponding array bounds checking are left. In order to report the array boundary conditions that are not checked in the program, we first need to recover the array boundary conditions in our targeted set with the initial indexes. That is because that some array boundary conditions' indexes may have been changed by interprocedual data flow analysis or an assignment operation during the data flow analysis according to the rules listed in Table 2. As the location of the array has been preserved in the array boundary information according to the section *Array Boundary Information Construction*, the report process will be very simple. So the reports include the following information about each array index: the file, the line, the function and the array expression it is in, the boundary condition should be checked.

Algorithm 2 Backward Data Flow Analysis in Block Algorithm HandleBlock(block,ArrayBoundaryConditionSet,arrayStmt)

Input:
block, *ArrayBoundaryConditionSet*, *arrayStmt*
Output:
ArrayBoundaryConditionSet

```
1: Stack s=NULL;
2: r=UNFOUND;
3: for each stmt in block do
4:    if stmt==arrayStmt then
5:       break;
6:    else
7:       s.push(stmt);
8:    end if
9: end for
10: while s not empty do
11:    stmt=s.top();
12:    s.pop();
13:    r=HandleStmt(stmt,ArrayBoundaryCondition);
14:    if r==FOUND then
15:       ArrayBoundaryConditionSet.delete(
          ArrayBoundaryCondition);
16:       break;
17:    end if
18: end while
19: return ArrayBoundaryConditionSet;
```

Algorithm 3 Interprocedual Backward Data Flow Analysis Algorithm InterBDFA(*CallGraph*,*CFG*,*ArrayBoundaryConditionSet*, *Function*, *Depth*)

Input:
CallGraph, *CFG*, *ArrayBoundaryConditionSet*, *Function*, *Depth*
Output:
ArrayBoundaryConditionSet

```
1: if Depth ≤ 0 then
2:    return ArrayBoundaryConditionSet;
3: end if
4: parents=CallGraph->getParents(Function);
5: if parents.size()==0 then
6:    return ArrayBoundaryConditionSet;
7: else
8:    for each parent in parents do
9:       pCFG=CFG(parent);
10:      ArrayBoundaryConditionSet1=handleParameters(
         ArrayBoundaryConditionSet, Function, parent);
11:      ArrayBoundaryConditionSet2=intraBDFA(pCFG,
         ArrayBoundaryConditionSet1);
12:      ArrayBoundaryConditionSet=interBDFA(
         CallGraph,CFG,ArrayBoundaryConditionSet2,
         parent,Depth-1);
13:   end for
14: end if
15: return ArrayBoundaryConditionSet;
```

4. IMPLEMENTATION AND EVALUATION

In this paper, we implement a tool called Carraybound (C array bound) and its architecture is shown in Figure 4. Carraybound can run on both Windows and Linux operating systems. And Carraybound is based on Clang 3.6 to generate abstract syntax tree (ast) files of a program. First, Carraybound takes AST files of a program as inputs. Then, Carraybound builds control flow graph and call graph based on the abstract syntax tree. Next, Carraybound performs taint analysis and data flow analysis based on the control flow graph and call graph. Finally, Carraybound outputs array bounds checking reports. Carraybound is a total automatic tool, which can take user configurations to configure the depth to perform interprocedual data flow analysis.

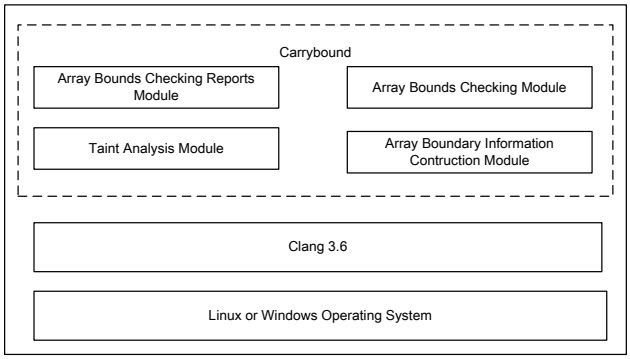

Figure 4: Carraybound Tool Architecture

Memory Optimizing: At the beginning, Carraybound was designed to read-in all the AST files at one time. There were all the contents of the ASTs. A large scale of program always contains lots of AST files. When we use Carraybound to scan the source code of PHP-5.6.16, which contains 250 thousand lines of source codes and 211 ast files, the execution will corrupt under the 2 GB memory limit. In order to support the scan of a hundred thousand lines, a million lines of code under limited memory resources, we implement a memory optimizing strategy in Carraybound. The key idea of the memory optimizing strategy is to use an AST queue to only preserve the latest used ASTs in the memory, e.g. 200 ASTs. Less ASTs, Less memory consumption. And the maximal capacity of the AST queue can be configured by the users. Users can configure an appropriate maximal capacity of the AST queue according to their demands and their computer's capacity. When analyzing the contents of an AST, Carraybound will first check whether the corresponding AST is in the memory. If the AST is in the memory, Carraybound will move the AST to the end of the queue. If the AST is not in the memory, Carraybound will read-in the AST's contents from the AST files. When the AST queue reaches its maximal capacity, Carraybound will remove the AST read-in at the earliest. It's remarkable that Carraybound will read the AST files more frequently if the user set a small maximal AST number. So if there are sufficient memory sources, the user should choose a bigger maximal AST number to reduce the frequent read operation and raise the efficiency of Carraybound.

Evaluation: To evaluate the effectiveness of Carraybound, we use Carraybound to scan the source codes of some programs. The results are shown in Table 3. The scale of the programs ranges from 1 KLOC to 250 KLOC. Carraybound reports some errors that may miss array index bounds checking. We randomly select some errors in each program to perform human validation. And Carraybound reports 2

Table 3: Results of Carraybound Scanning

Program	Lines of Code	Carraybound Warning Number	False positive rate
bzip2-1.0.6	8117	62	22.86%
pbzip2	1950	2	100%
PHP-5.6.16	250000+	175	12%
anomynity	N/A	245	79.2%

Table 4: Results of memory optimizing

Maximal AST Number	100	50	25	13
Total Memory Consumption	1570MB	854MB	400MB	235MB

errors in pbzip2. We find that the two reports are both false warnings because Carraybound cannot deal with the situation that the if condition includes a library function call. And when the statements include complex expressions, Carraybound may produce false positives. According to Table 3, the average false positive rate is 67.43%.

To evaluate the efficiency of Carraybound's memory optimizing, we use Carraybound to scan the source codes of PHP-5.6.16, which contains 250 thousand lines of source codes and 211 ast files, under the configuration of different maximal AST numbers. And all analysis can be done in about ten minutes. The results can be found in Table 4. The results show that the total memory consumption will be reduced by setting a less maximal AST number, which shows the effectiveness of Carraybound's memory optimizing method.

5. RELATED WORK

5.1 Taint analysis

Dynamic taint analysis is a popular means for analyzing both benign and malicious software components. Several different techniques have been proposed based on dynamic taint tracking for detecting unknown vulnerabilities in software [8, 10, 24, 27]. They taint potential input sources of malicious data such as network packets; monitor how the tainted input data propagate throughout program execution; and raise an alarm when the taint contaminates sensitive data like return addresses in the stack or user privilege configuration. Compared to dynamic taint analysis, static taint analysis traces information statically in source or binary. STILL [31] is a generic defense based on static taint and initialization analyses, to detect exploit code embedded in data streams/requests targeting at various Internet services such as Web services.

In order to reduce overhead of taint analysis, TaintPipe [23] performs very lightweight runtime logging to produce compact control flow profiles, and spawns multiple threads as different stages of a pipeline to carry out symbolic taint analysis in parallel. One of the other problems taint analysis faces is the manual effort. Most of the existing static taint analysis tools produce some warnings on potentially vulnerable program locations. It is then up to the developer to analyze these results by scanning the possible execution paths that may lead to these locations with unsecured user inputs. Ceara et al. /citeceara2010taint presents a Taint Dependency Sequences Calculus, based on a fine-grain data and control taint analysis, that aims to help the developer in this task by providing some information on the set of paths that need to be analyzed.

5.2 Array index out-of-bounds

Xu et al. [33] operates directly on the untrusted machine-code program, requiring only that the initial inputs to the untrusted program be annotated with typestate information and linear constraints. Detlefs et al [11] describe a static checker for common programming errors, such as array index out-of-bounds, null-pointer dereferencing, and synchronization errors (in multi-threaded programs). Their analysis makes use of linear constraints, automatically synthesizes loop-invariants to perform bounds checking, and is parameterized by a policy specification. Their safetychecking analysis works on source-language programs and also makes use of analyses that are neither sound nor complete.

Leroy and Rouaix [20] have proposed a theoretical model for systematically placing type-based run-time checks into interface routines of the host code. Their technique differs from ours in several respects: it is dynamic, it checks the host rather than the code, and it requires that the source of the host API be available. Furthermore, safety requirements are specified by enumerating a set of predetermined sensitive locations and invariants on these locations, whereas our model of a safety policy is more general.

ABCD [5] is a light-weight algorithm for elimination of array Checks on Demand. Its design emphasizes simplicity and efficiency. In essence, ABCD works by adding a few edges to the SSA value graph and performing a simple traversal of the graph. ABCD could removes on average 45% of dynamic bound check instructions, sometimes achieving near-ideal optimization.

Numbers of tools based on static analysis technique are available. Chimdyalwar [7] presents an evaluation of five such static analysis tools used for detection of this vulnerability. Among them, two are commercial tools Polyspace, Coverity; one is an academic tool ARCHER and the other two are open source tools UNO and CBMC. Polyspace is the only tool which is sound but with its memory intensive analysis, it cannot scale up on large applications with precise output. Whereas, Coverity scales up on million lines of code but its analysis is unsound. UNO was found to be unsound, imprecise and also could not scale up on large application. ARCHER claims to scale up on million lines of code but with unsound analysis. CBMC model checker does precise analysis but cannot scale up on large code with same precision.

5.3 Buffer overflow detection

Similar to array index out-of-bounds bugs, buffer overflow detection techniques are referable. Tance[30] presents a

black-box and combinatorial testing approach to detecting buffer overflow vulnerabilities. Dinakar et al. [12] describes a collection of techniques that dramatically reduce the overhead of an attractive, fully automatic approach for run-time bounds checking of arrays and strings in C and C++ programs. These techniques are essentially based on a fine-grain partitioning of memory, which bring the average overhead of run-time checks down to 12% for a set of benchmarks.

Approaches target debugging but work at the source level include Loginov's work on runtime type checking [21], rtcc [26]. All of these approaches focus on debugging and thus, their performance is not taken into consideration seriously. For instance, the reported overheads for Loginov's work are up to 900%. Some tools including SafeC [4] and Cyclone [17] use an augmented pointer representation that includes the object base and size of the legal target object for every pointer value. These pointers require significant changes to programs to allow the use of external libraries, typically introducing wrappers around library calls to convert pointer representations. Furthermore, writing such wrappers may be impractical for indirect function calls, and for functions that access global variables or other pointers in memory.

Mitigation techniques are other type of method could prevent the effect of array index out-of-bounds. For example, StackGuard [9] may terminate a process after it detects that a return address on the stack has been overwritten. Existing approaches to runtime prevention can incur significant runtime overhead. In addition, these approaches are in effect after potentially vulnerable programs are deployed. CFI [2] checks whether the control flow of program is hijacked during execution. This is in contrast with our work, which aims to develop and release programs that are free from array index out-of-bounds bugs prior to deployment.

5.4 Fuzzing

Fuzzing [28] is one of the most widely used black-box testing approaches in security testing. Fuzzing typically starts from one or more legal inputs, and then randomly mutates these inputs to derive new test inputs. Advanced fuzzing techniques [13] is generated-based fuzzing testing technique that enhance whitebox fuzzing of complex structured-input applications with a grammar-based specification of their valid inputs. [15] present an alternative whitebox fuzz testing approach combined with symbolic execution and dynamic test generation. Although fuzzing could detect array out of bounds bugs, it's poor code coverage is a major limitation. Moreover, some array out of bound bugs may only read the out of index array and thus don't lead to crash, which may not be detected by monitors in fuzzing [22]. Our method is based on static method which could achieve high code coverage and could detect different types of array index out-of-bounds.

Recently there has been a increasing amount of interest in approaches that combine symbolic execution with other testing techniques [6, 14, 32]. Symbolic execution in these techniques is used to collect path conditions consisting of a sequence of branching decisions, which could directly generate test cases. These branching decisions are then negated systematically to derive test inputs that when executed, will explore different paths. In order to detect array index out-of-bounds bugs, memory safety constraints are formulated and solved together with these path conditions. Two problems faced by symbolic execution are path explosion and limitation of constraint solver. Complicated and large scale program may lead to path explosion and thus the constraints can't be solved. Additionally, constraints introduced by array of bounds make the whole constraints much more complicated, which is another obstacle for constraint solver.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a static analysis method to automatically scan the source codes of C programs to check whether the right statements are included in the programs to ensure that the indexes of the arrays are within the array boundaries. Our static analysis method is based on static taint analysis and backward data flow analysis. And we implement an automatic static analysis tool—Carraybound. Carraybound can run on both Windows and Linux operation systems. And we evaluate the Carraybound tool by scanning the source codes of some real programs. And the results show that Carraybound can report many errors that array indexes don't have corresponding boundary checks in the program. Although Carraybound have some false positives and false negatives, which are not unavoidable in static analysis method, Carraybound can reduce lots of work of the programmers as it is a total automatic tool and is easy to use. In general, Carraybound can detect array bounds checking bugs in C programs effectively and efficiently.

As the experiments show, Carraybound reports a low rate of false positives and false negatives sometimes, but sometimes it also has a high false positive or false negative rate. So Carraybound's effectiveness depends on the style of the programs. For instance, Carraybound cannot handle complex expressions and library function calls. In order to make Carraybound more effective in any time, we think that it is a valid solution to use the Z3 theorem prover [1] to solve the constraints generated during our array bounds checking. Z3 is a high-performance theorem prover being developed at Microsoft Research. By using Z3, we can handle the complex expressions in the programs more easily. To handle the statements including library functions calls, we can implement some commonly used library function calls in Carraybound, like sizeof and strlen etc.

7. ACKNOWLEDGMENTS

The paper was partially supported by the National Natural Science Foundation of China (No. 91418204, 61321491, 61472179, 61561146394, 61572249).

8. REFERENCES

[1] Z3 theorem prover. https://z3.codeplex.com/.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[3] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. *Efficient detection of all pointer and array access errors*, volume 29. ACM, 1994.

[5] R. Bodík, R. Gupta, and V. Sarkar. Abcd: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating

inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[7] B. Chimdyalwar. Survey of array out of bound access checkers for c code. In *Proceedings of the 5th India Software Engineering Conference*, pages 45–48. ACM, 2012.

[8] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 133–147. ACM, 2005.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[10] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 235–248. ACM, 2005.

[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. 1998.

[12] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[13] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[14] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[15] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[16] J. L. Gross and J. Yellen. *Handbook of graph theory*. CRC press, 2004.

[17] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th international symposium on Memory management*, pages 73–84. ACM, 2004.

[18] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: theory and practice*. CRC Press, 2009.

[19] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[20] X. Leroy and F. Rouaix. Security properties of typed applets. In *Secure Internet Programming*, pages 147–182. Springer, 1999.

[21] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 217–232. Springer, 2001.

[22] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: the state of the art. Technical report, DTIC Document, 2012.

[23] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. Taintpipe: pipelined symbolic taint analysis. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 65–80, 2015.

[24] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[25] B. G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[26] J. L. Steffen. Adding run-time checking to the portable c compiler. *Software: Practice and Experience*, 22(4):305–316, 1992.

[27] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Acm Sigplan Notices*, volume 39, pages 85–96. ACM, 2004.

[28] M. Sutton, A. Greene, and P. Amini. Brute force vulnerability discovery, 2007.

[29] K. Thulasiraman and M. N. Swamy. *Graphs: theory and algorithms*. John Wiley & Sons, 2011.

[30] W. Wang, Y. Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. Kacker, and R. Kuhn. A combinatorial approach to detecting buffer overflow vulnerabilities. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 269–278. IEEE, 2011.

[31] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 289–298. IEEE, 2008.

[32] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 27–38. ACM, 2008.

[33] Z. Xu, B. P. Miller, and T. Reps. Safety checking of machine code. In *ACM SIGPLAN Notices*, volume 35, pages 70–82. ACM, 2000.

[34] T. Ye, L. Zhang, L. Wang, and X. Li. An empirical study on detecting and fixing buffer overflow bugs. In *ICST'16*, pages 91–101.