Inter-procedural pointer analysis targeting systems software

Nikhil Arvind Ranade

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Ranade, Nikhil Arvind, "Inter-procedural pointer analysis targeting systems software" (2002). Retrospective Theses and Dissertations. 21296.
https://lib.dr.iastate.edu/rtd/21296

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Inter-procedural pointer analysis targeting systems software

by

Nikhil Arvind Ranade

A thesis submitted to the graduate faculty
in partial fulfillment of the requirement for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Suraj Kothari (Major Professor)
Simanta Mitra
Manimaran Govindarasu

Iowa State University
Ames, Iowa
2002
Graduate College
Iowa State University

This is to certify the master’s thesis of

Nikhil Arvind Ranade

has met the thesis requirements of the Iowa State University

Signatures have been redacted for privacy
Table of Contents

Acknowledgement .............................................................................................................. vi
Abstract .............................................................................................................................. vii

1 Introduction .................................................................................................................. 1

2 Literature survey .......................................................................................................... 3
  2.1 Pointer analysis and issues................................................................................... 3
  2.2 Representation of pointer aliases ......................................................................... 5
  2.3 Algorithms ........................................................................................................... 7

3 Edg2XML .................................................................................................................. 12
  3.1 Edg front end ...................................................................................................... 13
  3.2 Edg2XML tool ................................................................................................... 13
  3.3 XML filter .......................................................................................................... 13
  3.4 The final XML file ............................................................................................. 13
  3.5 XML format ....................................................................................................... 14

4 The sysprobe architecture and pointer analysis ......................................................... 16
  4.1 XML centric analyzers ....................................................................................... 17
  4.2 Pointer analysis algorithm ................................................................................ 18
  4.3 Main components ............................................................................................... 19
    4.3.1 Call order graph generator ............................................................................. 19
    4.3.2 BLAST generator .......................................................................................... 19
    4.3.3 Bottom-up analyzer ....................................................................................... 19
    4.3.4 Top down analyzer ........................................................................................ 20
    4.3.5 Veras .............................................................................................................. 20
  4.4 Pointer analysis algorithms ................................................................................ 21
9.1 Sample code and XML

54
Acknowledgement

I would like to thank Dr Kothari for the constant support and motivation he has provided me towards the completion of this thesis. I would also like to thank Dr Mitra and Dr Manimaran for the valuable suggestions they have given me. I would like to thank Jeremias, Yunbo, Yogy, Sastra and all other lab members for their help and support. Lastly I want to thank my parents and all my friends who have always helped and encouraged me.

This work was performed at Ames Laboratory under Contract No. W-7405-Eng-82 with the U.S. Department of Energy. The United States Government has assigned the DOE Report number IS-T 2454 to this thesis.
Abstract

Pointer Analysis is critical for analyzing systems software written in 'C'. Systems software like Linux, in general make heavy use of pointers, structures, and function pointers. Developers often spend significant time and find it very tedious to chase pointers to understand complex interactions within systems software. The problem is even worse for novices who are getting trained. Moreover, without a tool for automating the analysis, users are more likely to make errors in analyzing large software. This thesis presents a tool called *SysProbe*, which can help the user to easily navigate through complex systems software. The tool uses a flow and context sensitive pointer analysis algorithm to analyze the source code.
1 Introduction

Pointer analysis is critical for analyzing systems software written in ‘C’. Pointer analysis can help the user comprehend code in many ways. For small programs, with complex algorithms involving heavy use of pointers, the analysis can help the user understand how the algorithm works. For large programs with thousands of lines of code, pointer analysis can help the user in many other ways like determining the location to which a pointer may point to at a given place in a code or to understand the relationship between various data structures. We are targeting our analysis towards such large systems software. Operating system software, network protocols and embedded software are some important examples that fall into this category. The issues one encounters while analyzing such software are, heavy use of structures, function pointers and data pointers. Furthermore, there are hundreds of global data structures. An important view that software analysis tools extract from the code is the call order graph, which gives the sequence of function call invocations. In most systems software, the call order graph is not a single tree, but a forest of trees. This is due to the fact that the program execution is interrupt-driven and the interrupt service routines are not explicitly called from the program. Furthermore, use of function pointers makes the construction of call order graph more complex.

Given its importance it is not surprising that pointer analysis has been subject of considerable research. The pointer analysis algorithms fall into two main categories: Flow and context insensitive algorithms [3,4,10,14,17,23] and flow and context sensitive algorithms [9,15,18,25]. The flow and context insensitive algorithms are typically faster than their counterpart but do not give precise results. Pointer analysis in general is a hard problem [12,24]. We provide an efficient context sensitive and flow sensitive algorithm,
which gives, precise analysis results. We are able to make the analysis efficient because of certain assumptions that are applicable to systems software.

Our tool provides various ways in which the user can understand complex systems software. The tool provides the call order graph. The user also has the ability to navigate through the source code from the call order graph. It provides the user, for each invocation of a function in the program, the status of all the pointer variables. It can give the points-to information for any pointer variable at any place in the program. Systems software usually consists of many global data structures that are linked together through chain of pointers. By helping the user understand the interaction between these global structures, the tool becomes a valuable source of information for the user.

We give a case study of the XINU operating system [8]. XINU operating system software is written in C and makes heavy use of data pointers, function pointers, structures and arrays.

The thesis is organized as follows. Chapter 2 gives a literature survey of the work done on pointer analysis. Chapter 3 gives an overview of our intermediate language representation (XML) and the tool that builds the XML. Chapter 4 gives the algorithm and walks through some examples. Chapter 5 gives an overview of our tool SysProbe. Chapter 6 covers a case study of the XINU Operating system, followed by results and conclusion.
2 Literature survey

There is a lot of research done on pointer analysis [1,5,13,16,19,20,22]. Here is a survey on the work done.

2.1 Pointer analysis and issues

As stated earlier, points-to analysis gives for a pointer variable in a code, the locations to which the variable can point to. Various pointer analysis algorithms are available in literature and we will give an overview of some of them in this chapter. Pointer analysis is known to be difficult to solve. Landi[24], Ramalingam[12]. Alias analysis is implemented in commercial compilers like gcc. Next, we discuss the issues for dealing with pointer analysis of a code Jonathan [13].

One of the main issues that arise in pointer analysis is due to inter-procedural analysis. When a pointer is passed as a reference parameter to a function, and that parameter is modified, then that change has to be incorporated in the calling function.

In many systems programs, we encounter large amount of global data structures. These data structures are linked to each other by a chain of pointers. Keeping track of this global pointer variable is difficult.

Resolving function pointers is also a must while doing pointer analysis. The use of function pointers modifies the call order graph.
Structures have to be handled carefully as they introduce other type of aliasing. Steensgaard[2], Yong[21].

```c
int read ()
struct devsw
{
    int (*dvread)();
}

main ()
{
    struct devsw devtab, * devptr ;
    devptr = & devtab ;
    devptr->dvread = read ;
}
```

**Figure 1: Use of pointers in structures**

Consider the code shown in Figure 1. Here devtab.dvread and devptr->dvread point to the same location. This has to be taken into account and our algorithm does take care of this. Recursive data structures may lead to infinite number of aliases. Taking care of these is also an issue.

Dealing with pointer arithmetic is also an issue. This arises when we have a pointer pointing to an element of an array. When the pointer is incremented, then it is difficult to keep track of where it is pointing. This also deals with the issue of whether all the elements of an array should be aggregated to one field.
2.2 Representation of pointer aliases

The pointer analysis results are represented in various ways.

```
main ()
{
    int c, *a, *d, **b;
    a = &c;
    d = &c;
    b = &a;
}
```

**Figure 2 : Sample Code to illustrate representation of points-to analysis**

Consider the code in Figure 2. In the complete representation of alias pairs all the aliases are stored separately. For the code in Figure 2, the alias pairs will be stored as 

\[(<a,c>,<d,c>,<b,a>,<b,c>,<a,d>,<b,d>).\]

In the compact representation of alias pairs, only the basic alias pairs are stored and the other alias pairs are derived from the basic alias pairs by applying transitivity, commutativity etc. In this case for the code in Figure 2, the alias pairs will be stored as 

\[(<a,c>,<d,c>,<b,a>).\]
In the points-to graph approach the results are stored in the form of a graph as shown in Figure 3.

Figure 3: Points-to graph
2.3 Algorithms

Various pointer analysis algorithms can be classified according to their flow and context sensitivity. A flow-sensitive algorithm considers the order of statements in a program, while a flow-insensitive algorithm does not. A context-sensitive algorithm considers the call-return sequences of procedures in a program, while a context-insensitive algorithm does not.

Figure 4 gives an example that explains flow sensitivity.

Figure 4: Example illustrating flow sensitivity in points-to analysis

Figure 4 shows the points to analysis results using flow sensitive and flow insensitive approach. In the flow insensitive approach, the order of statement execution is not considered, so the pointer p may point to a or b as seen. But in the flow sensitive approach, the earlier assignment of p to address of a will be killed by the later assignment to address of b. The flow sensitive graph in Figure 4 shows this result.
Figure 5 gives an example that explains context sensitivity.

Figure 5 shows a sample code and the results using context sensitive and context insensitive points-to analysis. Four functions $f_1$, $f_2$, $f_3$ and $f_4$ are shown. The code is shown in the form of a call order graph. The code for each of the function is given in the boxes. The graph indicates that $f_1$ calls $f_2$ and $f_3$ both of which call $f_4$. The results shown are the points-to analysis results for $f_4$. For context insensitive analysis the calling function context is not taken into account and hence only one points-to analysis result is obtained. For context-insensitive analysis, the calling function context has to be taken into account. Hence as $f_4$ has two distinct paths coming to it starting from the root node, there are two points-to analysis results for $f_4$, One for the path $f_1$-$f_2$-$f_4$, and other for the path $f_1$-$f_3$-$f_4$. 
Flow and context insensitive algorithms are conservative, in the sense that the set of locations pointed to by a pointer variable are a superset of the set of variables that the pointer variable will actually point to during runtime. Typically, such algorithms are less complex and provide good practical performance and scalability. The disadvantage of these algorithms is that the results are not precise. Flow and context sensitive algorithms give more accurate results than flow and context insensitive algorithms at the cost of time and space.

We now present the various algorithms present in literature.

Address-taken analysis is a flow-insensitive algorithm often used in production compilers that records all variables whose addresses have been assigned to another variable. This set includes all heap objects and actual parameters whose addresses are stored in the corresponding formal parameters. This analysis is efficient because it is linear in the size of the program and uses a single solution set, but can be very imprecise.

Steensgaard’s [2] analysis consists of a flow and context insensitive algorithm that employs a fast union-find data structure to represent all alias relations. This results in a linear time algorithm that makes one pass over the program.

Andersen [14] has a flow and context insensitive algorithm that can be more precise than Steensgaard’s analysis, as it does not perform the merging required by the union-find data structure.

Burke et al [4] have a flow-insensitive algorithm that also iterates over all pointer-related statements in the program. It differs from Andersen’s analysis in that it computes an alias
solution for each procedure, requiring iteration within each function in addition to iteration over the functions.

Choi et al. [15] have a flow-sensitive algorithm that computes a solution set for every program point. It associates alias sets with each CFG node in the program and uses worklists for efficiency.

Landi and Ryder [25] present an approximation algorithm for inter-procedural pointer induced aliasing, based upon conditional may-alias information that describes aliasing within a procedure, assuming certain conditions hold true at its entry.

Liang and Harrold [6] describe a context-sensitive flow-insensitive algorithm while Emami and Ghiya [9] present an algorithm that is flow and context sensitive. Wilson and Lam [18], present a context sensitive, flow-sensitive algorithm for pointer analysis. In their algorithm, they summarize the effects of procedures using partial transfer functions. A partial transfer function (PTF) describes the behavior of a procedure assuming that certain alias relationships hold when it is called. A PTF can be reused in many calling contexts. The analysis assumes potential values of the pointers at each statement in the program.
Table 1 states in short the various algorithms that were discussed in the above section.

<table>
<thead>
<tr>
<th>Name</th>
<th>ContextSensitive</th>
<th>FlowSensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Landi, Ryder</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Andersen</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Steensgaard</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ShapiroHorwitz</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Wilson</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Zheng</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Ghiya</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Liang, Harrold</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
3 Edg2XML

Here is an overview of the Edg2XML tool. The tool converts source code in C, C++, Fortran and Java into an XML representation of the source code. All the analysis takes place using this XML representation.

![Edg2XML tool](image)

**Figure 6 : Edg2XML tool**

Figure 6 shows the conversion of a source file into the corresponding XML representation.
3.1 Edg front end

The EDG front End is a front end designed by the Edison Design Group. The tool takes a source file in C, C++, Fortran or Java as input and outputs an intermediate representation, called as the il file. We use this il file as an input to the Edg2XML tool.

3.2 Edg2XML tool

The tool converts the (intermediate) il file to an XML representation of the source code. The structure of the il file is the same regardless of the source language used. There are some language specific features, which are unique to an il file of a particular source language. The Edg source code contains a header file il_def.h. This file has data structures which essentially describe the grammar of a specific language. We have a tool that generates the Edg2XML tool given the il_def.h file. In this way, the tool can easily scale to accommodate future other languages, supported by the front end.

3.3 XML filter

The output of the Edg2XML tool is a huge XML file which we call the BIG XML file. It has every detail regarding the source code. Many of these details are not required for a given analysis. We have a filter that filters the BIG XML to produce the final SMALL XML file. The filter is implemented using XSLT. It filters some information about variables which is repeated amongst other things.

3.4 The final XML file

The XSLT filter produces the XML file which is used as an input for various types of analyses. The XML file is a tree representation of the source code. It closely resembles
the AST (Abstract Syntax tree) for the source code. An example of the SMALL XML is given in the appendix.

### 3.5 XML format

The format of XML given in EBNF form is shown in Figure 7. Extended BNF is a standard based on Backus-Naur form. It adds the regular expression syntax of regular languages to the BNF notatation, in order to allow very compact specifications.

**Notations:**

- `*` Stands for repetition from 0 to n times.
- `+` Stands for repetition from 1 to n times.
- `?` Stands for either 0 or 1 times.
Root : HEADER (globals)* (function)+;
globals : decl;
decl : (VARIABLE | STRUCT)*;
function : (funcType)? (locals)* block
funcType : (formals)*;
formal : decl;
locals : decl;
block : (Statement | If | For | While)*;
Statement : expr;
expr : (EXPR)+;
If : expr ifpart (Elsepart)?;
ifpart : block;
elsepart : block;
For : (Init)? (test)? (Post)? block
Init : Statement;
test : expr;
Post : expr;
While : expr block;

Figure 7: Grammar explaining the structure of the XML file
The KCS framework is a product of the work done in the Knowledge Centric Software Engineering laboratory (KCS). The KCS framework consists of various analyzers which are used for program comprehension. The KCS framework is built upon a 3-tier architecture. As seen in Figure 8, the first tier consists of the EDG front-end where the source code is converted into an equivalent XML representation. In the second tier, the various analyzers take this XML as an input and produce results in the form of XML documents. The third tier is the visualizer that takes these results and gives visual output. 

_SysProbe_ is a program comprehension tool built on the KCS framework.
4.1 XML centric analyzers

All the analyzers produce and consume XML files. As the analyses are memory intensive, it is not practical to store all the results in memory. Thus the overall analysis is divided into parts and the various parts exchange data in the form of XML Documents. Example of sample XML documents is given in the appendix.
4.2 Pointer analysis algorithm

The algorithm for the pointer analysis is given in Figure 9.

![Figure 9: Schematic of pointer analysis algorithm](image)

The analysis starts from the SMALL XML's that are produced by the Edg2XML Tool.
4.3 Main components

4.3.1 Call order graph generator

It constructs a call order graph from the SMALL XML. The call order graph gives the call invocation sequence in the program. It is stored in an XML format.

4.3.2 BLAST generator

The BLAST generator takes as input the SMALL XML files and the call order XML and produces the BLAST (Block Level Abstract Syntax Tree) XML. The BLAST is a tree structure, which consists of control nodes like ExprNode, For, While, If and Block. The BLAST is constructed by pruning the original Source XML and retaining the information at the Block level. The ExprNode in the BLAST records the pointer assignments.

4.3.3 Bottom-up analyzer

The pointer analysis algorithm we use is divided into two distinct phases. The first phase is the bottom-up analyzer. Here the analysis starts at the root node in each call order tree, and each function is analyzed by doing a depth first search of the tree. So, during analysis, when a function call is encountered in the calling function, the bottom-up analyzer is called recursively for each called function. After analysis of a function is completed, the result has to be passed to the calling function in the call order tree. Before this is done, the points-to graph is pruned, so that the points-to set for local pointer declarations pointing to local variables is removed. The calling function receives the
results of analysis of the called function and modifies the results by doing a parameter mapping from the formal parameters to the actual parameters. The aim of the bottom-up analysis is to get a points-to analysis graph for each function, without taking into account the calling context. The results of the bottom-up analyses are stored in XML. The bottom-up analyzers also produce a function map XML. This XML records the possible functions that can be invoked by a function pointer in the code.

4.3.4 **Top down analyzer**

The second phase is the top-down analysis. The top-down analyzer does a breadth first traversal of each tree in the call order graph. At each node, the calling function $f_1$ passes to the called function $f_2$, the calling context, which is the result of points-to analysis up to the point of invocation of $f_2$. The local pointer declarations are pruned before passing the calling context. Subsequently, the function $f_2$ can be analyzed in a context-sensitive way using the context passed from the caller function $f_1$.

4.3.5 **Veras**

Veras is an utility which is used to visualize the graphs that are produced by the analysis.
4.4 Pointer analysis algorithms

4.4.1 Assumptions

We make the following important assumptions while doing pointer analysis.

- The call order graph contains no cycles. Hence the analysis dealing with recursive calls will not be precise.
- Pointer arithmetic is not handled.
- All the elements of an array are considered as one single entity.

4.4.2 Bottom-up analyzer

The next three functions show the working of the Bottom-up analyzer.

The `analyze-Function-Bottom-up` analyzes the call order graph in a bottom-up manner. The core function is the `analyzeBlock-Bottom-up` function, which takes as input the BLAST for that function and returns the PTA for that function. PTA is the points-to analysis graph for a function. After computing the PTA for that function, it is pruned to remove pointer assignments caused due to local variables and the result is returned to the calling function.

```c
/* Analyzes a function and returns the output points-to set */
PTA analyze-Function-Bottom-up (String funcName, BLAST blast)
{
    BLAST tree = getAST (funcName);
    PTA result = analyzeBlock-Bottom-up (BLAST tree);
    PTA returnSet = pruneLocals (result);
    return returnSet;
}
```

Figure 10: Bottom-up Analyzer
This function recursively analyzes a block in the BLAST. The BLAST is a tree structure consisting of various nodes. The nodes are of type ExprNode, If, While, For and Block. If the node is of type Block then that node is analyzed recursively and the PTA is obtained. If the node is of type ExprNode, then the analyze-ExprNode-Bottom-Up is called which analyzes the expression (either pointer assignment or function call) and returns the PTA for that expression. If the node is of type For or While, then the loop block is analyzed. The result of this loop analysis is again fed to the block and the final result is stored. If the node is of type If then the corresponding If and Else block are analyzed and their results are merged. This process goes for all children of the BLAST node and the results are merged.
/* Bottom-up analysis */
/* This function recursively analyzes a block in the Block Level Abstract Syntax Tree */

PTA analyzeBlock-Bottom-Up (BLAST blast)
{
    /* Initialize the PTA to be null */
    PTA currentPTA = null, resultPTA = null;

    /* analyze all nodes */
    for (int i=0;i<nodesInBlock;i++)
    {
        Node node = block.node[i];

        /* If it is an ExprNode then analyze the expression Node */
        if (node.value == "ExprNode")
            resultPTA = analyze-ExprNode-Bottom-Up (node);

        /* If it is a Block then recursively analyze the block */
        if (node.value == "Block")
            resultPTA = analyzeBlock-Bottom-Up (node);

        /* If it is a IF Block, analyze the condition, the if block and the else block if there is one. */
        else if (nodeValue == "IF")
        {
            PTA cond = analyze-ExprNode-Bottom-Up (node.condition);
            PTA ifBlock = analyzeBlock-Bottom-Up (node.if);
            PTA elseBlock = analyzeBlock-Bottom-Up (node.else);
            resultPTA = merge (cond, ifBlock, elseBlock);
        }

        /* Analyze loops iteratively */
        else if (nodeValue == "For")
        {
            PTA initPTA = analyze-ExprNode-Bottom-Up (node.init);
            PTA testPTA = analyze-ExprNode-Bottom-Up (node.test);
            PTA postPTA = analyze-ExprNode-Bottom-Up (node.post);
            PTA bodyPTA = analyzeBlock-Bottom-Up (node.body);
            resultPTA = merge (initPTA, testPTA, bodyPTA, postPTA);
            /* Iterate until the results converge */
            PTA newPTA;
            do
            {
                newPTA = resultPTA;
                resultPTA = merge (newPTA, testPTA, bodyPTA, postPTA);
            } while (newPTA != resultPTA);
        }

        /* Merge the results */
        currentPTA = merge (resultPTA, currentPTA);
    }
}

Figure 11: Analysis of a block in bottom-up analysis
The *analyze-ExprNode-Bottom-Up* function analyzes an *ExprNode* during a bottom-up analysis. If the expression is a pointer assignment, then it updates the corresponding PTA. If it is a function call, then the function is analyzed. From the result PTA set obtained from the analysis of that function, the formal parameters are mapped back to the actual parameters, and the set is updated with the current PTA.

```c
/* This analyzes an expression Node in the tree during the bottom up analysis. */
PTA analyze-ExprNode-Bottom-Up (ExprNode e)
{
  // Iterate over all expression nodes
  for (int i=0;i<e.size();i++)
  {
    /* Initialize current PTA to null */
    PTA currentPTA = null , resultPTA = null ;

    /* If it is a pointer assignment */
    if (e.kind == "pointer-assignment")
    {
      resultPTA = updatePTA(e) ;
    }

    /* If it is a call */
    else if (e.kind == "call")
    {
      /* Analyze the called function */
      PTA result = analyseFunction-Bottom-Up (functionName)

      /* Map the formal parameters to the actual parameters */
      resultPTA = unMapParameters (formals, actuals, result) ;
    }
    currentPTA = merge (currentPTA, resultPTA));
  }
  return currentPTA ;
}
```

*Figure 12: Analysis of an Expression Node during bottom-up analysis*
4.4.3 Top-down analyzer

The *analyze-Function-Top-Down* analyzes the call order graph. The core function is the *analyzeBlock-Top-Down* function, which takes as input the BLAST and an input PTA for that function and returns an output PTA for that function. After computing the PTA for that function, it is pruned to remove pointer assignments caused due to local variables and the result is returned to the calling function.

```java
/* Stores context sensitive, points to analysis results for each function context. */
PTA analyze-Function-Top-Down (String functionName, PTA inputPTA, String prefix) {
    String name = prefix + functionName + "in";
    /* Store the context sensitive PTA for that calling context in the store. */
    store.put (name,inputPTA);
    BLAST blast = getAST (functionName);
    /* Analyze the block */
    analyzeBlock-Top-Down (blast, inputPTA);
}
```

**Figure 13: Top-Down Analyzer**

The *analyzeBlock-Top-Down* function is similar to the *analyzeBlock-Bottom-Up* function. The only difference is that an inputPTA set is passed to the function and the currentPTA is initialized to that inputPTA set instead of null. Also on analyzing a node of type *ExprNode* the function *analyze-ExprNode-Top-Down* is called.
The `analyze-ExprNode-Top-Down` function analyzes an `ExprNode` during a top-down analysis. If the expression is a pointer assignment, then it updates the corresponding PTA. If it is a call, then it does two things. It gets the result of the bottom-up analysis for that function, and updates the current PTA with it. Secondly, before updating the currentPTA, it prunes it to remove local assignments and calls `analyseFunction-TopDown` with the parameters as the pruned set, the function-name and the prefix.

```java
/* This analyses an expression Node in the tree during the top up analysis. */
PTA analyseExprNode-Top-Down (ExprNode e, String funcName, String prefix)
{
    /* Iterate over all expression nodes */
    for (int i=0;i<e.size();i++)
    {
        /* Initialize current PTA to null */
        PTA currentPTA = null, resultPTA = null;

        /* If it is a pointer assignment */
        if (e.kind == "pointer-assignment")
        {
            resultPTA = updatePTA(e);
        }

        /* If it is a call */
        else (e.kind == "call")
        {
            /* Get the results from the bottom-up analysis of that function */
            PTA result=getResultFromBottomUp (funcName);  

            /* Map the formal parameters to the actual parameters */
            resultPTA=unMapParameters(formals, actuals, result); 

            /* This is the set which will be the input to the called function. */
            passingSet=pruneLocals(currentPTA);

            /* Create prefix */
            prefix = prefix + funcName;

            /* analyze that function Top-down */
            analyseFunction-Top-Down (functionName, passingSet, prefix);
        }
        currentPTA = merge (currentPTA, resultPTA);
    }
}
```

Figure 14: Analysis of an Expression Node during top-down analysis
4.5 Pointer analysis issues

As we said before, a variety of issues come up during analysis of large systems software namely, heavy use of data pointers, function pointers, global shared variables and structures. We discuss how these issues are addressed.

4.5.1 Handling of function pointers

When a call is invoked through a function pointer, the points-to analysis at that point can determine the functions that may be invoked due to that pointer[7,11]. Each of these functions is analyzed at such a point. The results of the analysis are not merged, but the user is asked to select the function that he wishes to analyze. We illustrate this point with an example.

```c
int dsread (); /* disk read */
int lfread (); /* file read */
int dgread (); /* datagram read */
int ethread (); /* ethernet read */

main ()
{
  int (*dvread) ();
  if ()
    dvread = dsread ;
  else if ()
    dvread = lfread ;
  else if ()
    dvread = dgread ;
  else
    dvread = ethread ;

dvread ();
}
```

Figure 15 : Use of function pointers
In the presence of function pointers, the call order graph cannot be constructed by a simple syntactic analysis of the program, because a function pointer call-site cannot be bound to a unique function at compile time. A set of functions can be invoked from such a call-site. Due to the presence of function pointers, the analysis is carried in the following sequence. First a call order graph is constructed, without taking into account the possible bindings for function pointers. Then the bottom up analysis is done. During the bottom up points-to analysis, a function pointer map is created. This map contains for each function pointer, the set of functions that can be invoked at run-time due to an invocation through that function pointer. We would like to point out here that we are assuming that we are analyzing a closed system, which means that all the possibilities of function pointer bindings to actual functions can be determined by static analysis. This is a reasonable scenario to assume for systems software.

With the function pointer map, the call order graph is regenerated. In case of large systems programs, the call order graph is actually a forest of trees. In the regenerated call order graph, the total number of trees decreases because some trees become sub-trees through function pointer bindings.

4.5.2 Handling of arrays

The individual fields of an array are not distinguished and are treated in the same way.
4.6 Illustrative example

Now we present an example that illustrates the points-to analysis algorithm.

```c
void funcl (int **m, int **n)
{
  *m = *n;
  if (c)
    *m = &y;
  func2 (m);
}

void func2 (int **t)
{
  int c = 0;
  if (c)
    *t = &global;
}
```

Figure 16: Sample code for illustrating pointer analysis algorithm
Consider the code in Figure 16. So before the function pointer analysis, the call order graph will look like one given in Figure 17.

![Call Order Graph](image)

**Figure 17: Call Order Graph**

### 4.6.1 Bottom-up analysis

We start analysis from the leaf nodes and go up to the root. The results of analysis of ‘func2’ are shown in Figure 18.

![Points-to graph for func2](image)

**Figure 18: Points-to graph for ‘func2’**

Analysis of ‘func1’ up to line 20 will yield the following points-to graph shown in Figure 19. The ‘=’ node means that ‘m’ points to the same location that ‘n’ points to.

![Points-to graph after analysis of func1](image)

**Figure 19: Points-to graph after analysis of ‘func1’ up to line 20**
At line 21 ‘func1’ is analyzed. Taking the points-to set from ‘func2’ (shown in Figure 18) and after converting the formals to actual parameters we have graph shown in Figure 20.

So the points-to graph for ‘func2’ after analyzing line 21 is as shown in Figure 21.

While analyzing main up to line 9 the points-to graph is as shown in Figure 22.
Line 10 in ‘main’ is fp (&a,&b)

Now we know from the above points-to set that fp points-to func1. So func1 is analyzed. Taking relevant points-to set from func1 and converting formal to actual parameters we have points-to graph in Figure 23.

![Figure 23: Points-to graph for 'func1' after mapping parameters and pruning locals](image)

Merging with the current points-to set we have, graph in Figure 24.

![Figure 24: Points-to graph in 'main' after line 10](image)

After analyzing line 12 in ‘main’ we get the graph shown in Figure 25.

![Figure 25: Points-to graph after analysis of 'main' up to line 13](image)
Line 13 in `main` is `func2 (&b)`. Taking points-to set from `func2` and converting formal to actual parameters we have, graph in Figure 26

![Figure 26: Points-to graph for `func2` after mapping parameters](image)

Merging with the current points-to set we have graph in Figure 27, which is the final points-to analysis graph.

![Figure 27: The final Points-to graph for `main`](image)
4.6.2 Recomputing the call order tree

There were two trees in the call order graph seen in Figure 17. After the bottom up points-to analysis is completed, a function map shown in Figure 28 is generated.

![Function pointer map](image)

**Figure 28 : Function pointer map**

With this information, the call order graph as shown in Figure 29 is regenerated.

![New call order graph](image)

**Figure 29 : New call order graph**
4.6.3 Top-down analysis

After the bottom up analysis a top down analysis is carried out. As stated before, the points-to analysis at each calling context is computed and stored. So the points-to set for \textit{main-func1}, \textit{main-func2} and \textit{main-func1-func2} are stored. Note that as func2 has two parents in the call order tree, there are two points-to analysis graphs stored for func2.
4.7 Sample XML documents for the code

4.7.1 Call order tree

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <Function Name="main" Source="test.c">
    <Function Name="func2" Source="test.c" />
    <Function Name="fp" Source="" />
  </Function>
  <Function Name="func1" Source="test.c">
    <Function Name="func2" Source="test.c" />
  </Function>
</root>
```

Figure 30: Call Order Tree before function pointer analysis
4.7.2 BLAST for ‘main’ function

<?xml version="1.0" encoding="UTF-8" ?>
<ROOT>
  <BLOCK>
    <EXPR kind="passign" lineNo="11">
      <VAR name="b" kind="variable_address" />
      <VAR name="c" kind="variable_address" />
    </EXPR>
    <EXPR kind="passign" lineNo="12">
      <VAR name="fp" kind="variable_address" />
      <VAR name="func1" kind="routine_address" />
    </EXPR>
    <EXPR kind="call" lineNo="13">
      <FUNC name="fp" />
      <ATTR name="a" />
      <ATTR name="b" />
    </EXPR>
    <IF>
      <BLOCK>
        <EXPR kind="passign" lineNo="15">
          <VAR name="b" kind="variable_address" />
          <VAR name="d" kind="variable_address" />
        </EXPR>
      </BLOCK>
    </IF>
    <EXPR kind="call" lineNo="16">
      <FUNC name="func2" />
      <ATTR name="b" />
    </EXPR>
  </BLOCK>
</ROOT>

Figure 31: BLAST for the “main” function.
4.7.3 Bottom-Up PTA for main function

```
<?xml version="1.0" encoding="UTF-8" ?>
<ROOT>
  <Node name="b">
    <PointsTo>c ,11 ,test.c d ,15 ,test.c global ,34 ,test.c</PointsTo>
    <PointsFrom />
  </Node>
  <Node name="c">
    <PointsTo />
    <PointsFrom>b ,11 ,test.c a ,23 ,test.c</PointsFrom>
  </Node>
  <Node name="fp">
    <PointsTo>func1 ,12 ,test.c</PointsTo>
    <PointsFrom />
  </Node>
  <Node name="func1">
    <PointsTo />
    <PointsFrom>fp ,12 ,test.c</PointsFrom>
  </Node>
  <Node name="a">
    <PointsTo>c ,23 ,test.c global ,34 ,test.c</PointsTo>
    <PointsFrom />
  </Node>
  <Node name="global">
    <PointsTo />
    <PointsFrom>a ,34 ,test.c b ,34 ,test.c</PointsFrom>
  </Node>
  <Node name="d">
    <PointsTo />
    <PointsFrom>b ,15 ,test.c</PointsFrom>
  </Node>
</ROOT>
```

Figure 32: Points-to analysis results for the "main" function.
4.7.4 Recomputed call order tree

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<root>
  <Function Name="main" Source="test.c">
    <Function Name="func1" Source="test.c" Type="functionPointer">
      <Function Name="func2" Source="test.c" Type="" />
    </Function>
  </Function>
  <Function Name="func2" Source="test.c" Type="" />
</Function>
</root>
```

Figure 33: Call Order Tree after function pointer analysis
5 SysProbe tool

5.1 Overview of system architecture

We give an overview of our framework. The aim of this framework is to build a program comprehension tool. One of the important features of our tool is the intermediate representation. The tool converts source code in C, C++, Fortran and Java into XML representation of the source code. All the analysis takes place using this XML representation. The pointer analysis algorithms take as input the Block Level Abstract Syntax Tree (BLAST).

The tool gives a call order graph, which shows the call invocation sequence. The call order graph is actually a forest of call order trees. The call order graph is incomplete when function pointers are used in the program. Our analysis takes this into account and gives a precise call order graph. The call graph for the example covered earlier is shown in Figure 34.
The user can navigate through the source code, by clicking a node in the call order graph. Upon clicking a node, the source code that contains the function represented by that node, is shown.

We provide points-to analysis results for each function. The points-to set is provided for each pointer variable. This help the user in understanding, where each pointer variable points to. The analysis is flow and context sensitive. Figure 35 shows points to analysis graph for a function.
The user can browse through the source code and get the points to analysis graph for a particular variable at any place in the code. As the analysis is context sensitive, several points-to graphs may exist at a given statement in the function depending on the number of paths that reach that function from the root of its call order tree. The user specifies one such path. From this information, the stored results for that particular calling context is used to provide the points-to analysis graph. Figure 36 shows this can be done.
Figure 36: Points-to analysis at a point in the program
5.2 Call order tree snapshot

Figure 37 shows the call order tree for the main function of the XINU Operating System source code. Dark edges imply that those functions are invoked through a function pointer.

Figure 37: Call order graph for a function in XINU
6 Case study of XINU operating system

6.1 XINU operating system

The XINU Operating System codebase consists of 143 files with 174 subroutines. Here are some of the features of the system.

1. Large use of function pointers.
2. Large of data pointers.
3. Large use of global shared data.
4. Large use of structures.

All these features make XINU a challenge to analyze.

6.2 Quantitative results

Table 2 shows the pointer analysis statistics for the XINU operating system.

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td># Nodes</td>
<td>11136</td>
</tr>
<tr>
<td># LOC</td>
<td>5987</td>
</tr>
<tr>
<td># Subroutines</td>
<td>174</td>
</tr>
<tr>
<td># Assignments</td>
<td>1115</td>
</tr>
<tr>
<td># Pointer Assignments</td>
<td>290</td>
</tr>
<tr>
<td># Structures</td>
<td>31</td>
</tr>
<tr>
<td># Global pointer Assignments</td>
<td>235</td>
</tr>
<tr>
<td># Local pointer Assignments</td>
<td>55</td>
</tr>
</tbody>
</table>
Table 3: Number of nodes in call order tree

<table>
<thead>
<tr>
<th></th>
<th>Before Analysis</th>
<th>After Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>XINU</td>
<td>595</td>
<td>626</td>
</tr>
<tr>
<td>Main</td>
<td>63</td>
<td>212</td>
</tr>
<tr>
<td>Read</td>
<td>1</td>
<td>489</td>
</tr>
<tr>
<td>Write</td>
<td>1</td>
<td>468</td>
</tr>
<tr>
<td>Open</td>
<td>2</td>
<td>220</td>
</tr>
<tr>
<td>Close</td>
<td>1</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 4: Calls through function Pointers

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>XINU</td>
<td>42</td>
</tr>
<tr>
<td>Main</td>
<td>14</td>
</tr>
<tr>
<td>Read</td>
<td>5</td>
</tr>
<tr>
<td>Write</td>
<td>5</td>
</tr>
<tr>
<td>Open</td>
<td>3</td>
</tr>
<tr>
<td>Close</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5: Number of Nodes in the forest

<table>
<thead>
<tr>
<th></th>
<th>Before Analysis</th>
<th>After Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>XINU</td>
<td>68</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 2 shows general statistics for the XINU code. The results show the heavy use of pointers in the code. Approximately 26% of the assignments in the code are through
pointers. Out of those 81% are assignments involving global pointer variables. The code also contains many structures and most of the pointer assignments involve global variables.

Table 3 shows the total number of function invoked by a function pointer before and after analysis. The first row has the statistics for the entire XINU code. The other rows represent sample individual functions. The difference in the two numbers indicates heavy use of function pointers. For example consider the open function. The number of functions invoked before analysis is just 2 but after analysis are 220. This is due to the fact that some functions are invoked through function pointers and that binding is not available before analysis.

Table 4 shows the use of function pointers in various routines. Total of 42 (25%) functions are invoked through function pointer calls.

Table 5 shows the reduction in the number of root nodes in the call order graph. The original XINU forest consists of 68 tree nodes. The number is reduced to 41, when the function pointers are resolved.

6.3 An example

We give one example to show, how the analysis can help in understanding the relationships between data structures in the XINU operating system software. As stated before, the XINU code contains various data structures interconnected by chains of
pointers. We give an example that would help the user understand the data structures and routines that come into play in implementation of the XINU file system.

The points-to analysis diagram, showing the implementation of the file system for XINU is shown in Figure 38 above.

We explain in short how the pointer assignments take place in XINU. In the file conf.c, the conf() function initializes the device table data structure, 'devtab'. 'devtab' consists of an array of device switch tables devsw. The device switch table is a structure containing various fields, one of which is the dvioblk pointer. The main() function calls the init() function, which calls the dsinit() function through a function pointer. In dsinit(), the devtab.dvioblk pointer for one entry of device table is set to point to dstab, the disk table. Similarly in the lfin() function, which gets called in a similar way as dsinit(), the devtab.dvioblk for another entry of device table (devtab) is set to point to fltab, the file table. When a user function opens a file, dsopen() gets called through a function pointer. The function dsopen assigns the fltab.fl_dent pointer to the in-core directory, and also
assigns the $ddir$ pointer of $dstab$ to the in-core directory. Thus the above diagram is completed.

As seen from the description, this is quite a complex process. Our tool helps the user in the following way. $SysProbe$ comes up with a diagram, as shown in Figure 38. It shows how the various data structures interact with each other through chain of pointers. In Figure 38, the nodes are structures and the edges represent pointers. When one clicks on an edge, a list of functions, where assignments through that pointer have taken place are given to the user. The user selects the function of his choice, which takes him to that place in the source code, where the assignment actually takes place. This information is very valuable to the user.
7 Conclusion and future work

We have developed a flow and context sensitive points-to analysis algorithm, targeting large operating system software. We have taken a practical approach to solving the pointer analysis problem. We are building a knowledge-centric environment for solving problems related to a particular domain. One future work is to provide support for analyzing the concurrency and deadlock scenarios.
8 References


   *In OOPSLA '96 Conference Proceedings, San Jose, CA*, October 1996.


22. Susan Horwitz, Marc Shapiro, Modular Pointer Analysis, July 1998.


9 Appendix

9.1 Sample code and XML

/* Sample 'C' code for test purpose. */

char c,d;

struct devsw
{
    int (*dvinit) () ;
    int (*dvopen) (...) ;
    char *dvioblk ;
};

struct ds
{
    char *ddir ;
};

void init (struct devsw * devptr) ;

struct devsw devtab[4] ;
struct ds dstab[2] ;

main()
{
   void (*fp)(struct devsw * devptr) ;
   fp = init ;
   struct devsw * devptr ;
   devptr = & devtab[0] ;
   devptr->dvioblk = &c ;
   fp (devptr) ;
}

void init (struct devsw * devptr)
{
    if (1)
    devptr->dvioblk = & d ;
}

Figure 39 : Code example to illustrate the XML representation of a source code
This is the XML for the above code. This XML is shown with only the top 2 levels. The full expanded XML is shown after this.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Root language="C">
  <SourceFile ScopeNumber="">
    <Header>
      <Struct type="struct" typeName="devsw" name="devsw" unmangledName="">
      <Struct type="struct" typeName="ds" name="ds" unmangledName="">
      <Variable name="c" type="integer" typeName="" next="4984860" size="" seqNumber="3" columnNumber="6">
      <Variable name="d" type="integer" typeName="" next="4984952" size="" seqNumber="3" columnNumber="8">
      <Variable name="devtab" type="array" typeName="" next="4985044" size="" seqNumber="19" columnNumber="14">
      <Variable name="dstab" type="array" typeName="" next="0" size="2" seqNumber="20" columnNumber="11">
      <Function name="main" unmangledName="" ScopeNumber="">
      <Function name="init_FPSdevsw" unmangledName="init" ScopeNumber="">
    </SourceFile>
  </Root>
```

The file is divided into various sections. On the top you have the global variables. Each variable is a node in the XML. There are individual nodes for each function in the source file. The expanded function nodes are shown in the figure below. Each function consists of local declarations followed by a Block. The Block in turn contains various nodes like If, ForLoop, While and ExprNode nodes.
<?xml version="1.0" encoding="UTF-8" ?>
<Root language="C">
  <SourceFile ScopeNumber="">
    <Header>
      <SourceInfo fileName="struct_test.c" fullName="struct_test.c" language="C" includeMethod="local" isIncludeFile="False" lastSeqNumber="37" nameAsWritten="struct_test.c" firstSeqNumber="1" firstLineNumber="1" relatedFileImplicitlyIncludeDone="False"/>
    </Header>
    <Struct type="struct" typeName="devsw" name="devsw" unmangledName="">
      <Field name="dvinit">
        <Pointer type="" typeName="" />
      </Field>
      <Field name="dvopen">
        <Pointer type="" typeName="" />
      </Field>
      <Field name="dvioblk">
        <Pointer type="" typeName="" >
          <Integer kind="char" enumType="False" boolType="False"/>
        </Pointer>
      </Field>
    </Struct>
    <Struct type="struct" typeName="ds" name="ds" unmangledName="">
      <Field name="ddir">
        <Pointer type="" typeName="" >
          <Integer kind="char" enumType="False" boolType="False"/>
        </Pointer>
      </Field>
    </Struct>
    <Variable name="c" type="integer" typeName="" next="4984860" size="" seqNumber="3" columnNumber="6">
      <Integer kind="char" enumType="False" boolType="False"/>
    </Variable>
    <Variable name="d" type="integer" typeName="" next="4984952" size="" seqNumber="3" columnNumber="8">
      <Integer kind="char" enumType="False" boolType="False"/>
    </Variable>
  </SourceFile>
</Root>

Figure 40: Expanded XML file for the source code in figure
<Variable name="devtab" type="array" typeName=""
next="4985044" size="4" seqNumber="19"
columnNumber="14">
<Array numberOfElements="4" isVariableSize="False">

<Struct type="struct" typeName="devsw" name="devsw" unmangledName=""
><Field name="dvinit">
  <Pointer type="" typeName="" />
</Field>
<Field name="dvopen">
  <Pointer type="" typeName="" />
</Field>
<Field name="dvioblk">
  <Pointer type="" typeName="" />
  <Integer kind="char" enumType="False" boolType="False" />
</Field>
</Struct>
</Array>
</Variable>

<Variable name="dstab" type="array" typeName=""
next="O" size="2" seqNumber="20"
columnNumber="11">
<Array numberOfElements="2" isVariableSize="False">

<Struct type="struct" typeName="ds" name="ds" unmangledName=""
><Field name="ddir">
  <Pointer type="" typeName="" />
  <Integer kind="char" enumType="False" boolType="False" />
</Field>
</Struct>
</Array>
</Variable>

<Function name="main" unmangledName=""
ScopeNumber="">
</FunctionType />
<Variable name="fp" type="pointer" typeName=""
next="5046576" size="" seqNumber="24"
columnNumber="10">
  <Pointer type="" typeName="" />
</Variable>

<Variable name="devptr" type="pointer" typeName=""
next="0" size="" seqNumber="26"
columnNumber="18">
  <Pointer type="struct" typeName="devsw" />
</Variable>

<Block meta="" key="101" scopeNumber="" line="">
  <Statement meta="" key="102" kind="expr"
    line="" lines="( 0,0 )">
    <ExprNode kind="operation"
      operatorKind="call" meta=""
      name="_main" unmangledName="" />
  </Statement>
  <Statement meta="" key="103" kind="expr"
    line="" lines="( 25,25 )">
    <ExprNode kind="operation" meta=""
      operatorKind="passign">
      <ExprNode kind="variable_address"
        meta="" name="fp" />
      <ExprNode kind="routine_address"
        meta="" name="init" unmangledName="init" />
    </ExprNode>
  </Statement>
  <Statement meta="" key="104" kind="expr"
    line="" lines="( 27,27 )">
    <ExprNode kind="operation" meta=""
      operatorKind="passign">
      <ExprNode kind="variable_address"
        meta="" name="devptr" />
      <ExprNode kind="array_element"
        meta="">
      <ExprNode kind="variable"
        meta="" name="devtab" />
      <ExprNode kind="constant"
        meta="" value="0" />
    </ExprNode>
  </Statement>
</Statement>
<Statement meta="" key="105" kind="expr" line="" lines="(28,28)">
  <ExprNode kind="operation" meta="">
    <ExprNode kind="operation" operatorKind="passign">
      <ExprNode kind="operation" operatorKind="field">
        <ExprNode kind="variable" name="devptr" />
        <ExprNode kind="field" name="dvioblk" />
      </ExprNode>
      <ExprNode kind="variable_address" name="c" />  
    </ExprNode>
  </ExprNode>
</Statement>

<Statement meta="" key="106" kind="expr" line="" lines="(29,29)">
  <ExprNode kind="operation" operatorKind="call">
    <ExprNode kind="variable" name="devptr" />
    <ExprNode kind="variable" name="fp" unmangledName="fp"/>
  </ExprNode>
</Statement>

<Statement kind="return" key="107" line="" lines="(0,0)">
  <ExprNode kind="constant" value="0"/>
</Statement>
</Block>
</Function>

<Function name="init___FP5devsw" unmangledName="init"
  ScopeNumber="">
  <FunctionType>
    <Variable name="devptr" type="" typeName="">
      <Pointer type="struct" typeName="devsw" />
    </Variable>
  </FunctionType>
</Function>
<Block meta="" key="108" scopeNumber="" line="">
  <If meta="" key="109" line="" lines="(35,36)">
    <ExprNode kind="constant" meta="" value="1" />
    <Block meta="" key="110" scopeNumber="" line="">%
      <Statement meta="" key="111" kind="expr" line="" lines="(36,36)">
        <ExprNode kind="operation" meta="" operatorKind="passign"">
          <ExprNode kind="operation" meta="" operatorKind="field">
            <ExprNode kind="variable" meta="" name="devptr" />
            <ExprNode kind="field" meta="" name="dvioblk" />
          </ExprNode>
        </ExprNode>
        <ExprNode kind="variable_address" meta="" name="d" />
      </Statement>
    </Block>
  </If>
  <Statement kind="return" meta="" key="112" line="" lines="(0,0)" />
</Block>
</Function>
</SourceFile>
</Root>