Function Pointers in C - An Empirical Study

Anand Shah and Barbara G. Ryder

May 24, 1995

Abstract Interprocedural analysis requires a statically determinable call multigraph to represent the program. Programs that use function pointers or function-valued variables present a difficult problem for static analysis. In C, function pointers can be formal parameters, actual arguments to functions, or global/local variables. The difficulty of precisely determining the call multigraph of a program – or statically determining the aliases of a function pointer at a call site – depends on the types of function pointers used (i.e., local or global) [ZR94]. In this study, we have statically gathered empirical information on C function pointer usage to better predict appropriate interprocedural analyses required for C programs.

1 Introduction

To interprocedurally analyze programs, the analyzer must be able to statically construct a call multigraph of a program; in other words, the analyzer must be able to statically bind function (or procedure) call sites to the function that will be called at that call site. This information is required to statically propagate data flow information from call sites to within the called function and from within the called function back to the call sites. For languages that do not contain function-valued variables, determining the call multigraph at compile-time is a simple problem.

For languages that support function pointers, or function-valued variables, a call site may either be a *direct* call site – where the name of the function being called is specified, or an *indirect* call site – where the value of a function pointer determines which function may be called. In such languages, determining the call multigraph statically is difficult, because it is tantamount to statically determining the functions that may be aliased to a function pointer at a call site.

Programming languages support function pointers to varying degrees. Some allow function pointers *only* as formal arguments or as formal arguments with an assignment operator. C allows very general function pointer usage; a C program can contain function pointers as formals, globals or local variables, arrays of function pointers, pointers to function pointers, functions that return function pointers and function pointers as fields of structs and unions.

In [ZR94] we showed that the problem of statically determining the functions that may be aliased to a function pointer at a call site is often provably $\mathcal{NP} - hard$ except in special cases. Program transformation systems make safe assumptions about the possible functions that might be called at an indirect call site.

This study gathers empirical information on how C programmers use function pointers. We built a tool to statically extract function pointer declaration and usage information from C programs. A set of small to medium sized programs were passed through this tool to collect the data. The motivation for the study was to analyze empirical data to determine whether it is worthwhile to improve the existing algorithms for analysis in the presence of function pointers.

Overview Section 2 outlines the focus of our study. Section 3 describes the sources used as input. Section 4 briefly describes the implementation of the analysis tool. Section 5 contains the output tables and explains the various columns of the output with a description of C constructs that map into these columns. Section 6 draws conclusions based on the results of the study.

2 Study focus

We have a theoretical categorization of the kind of programs that can easily be analyzed to construct a call multigraph in the presence of function pointers [ZR94]. The results of this work show that:

- there is a polynomial time aliasing algorithm which can analyze some programs with single level, non-global function pointers. Here, *single level* refers to the allowed level of indirection of pointer usage. With only single level function pointers and pass by value parameter passing in C, the changes made to a function pointer within a called function can never be reflected in the caller. Intuitively, a polynomial function pointer aliasing algorithm exists in this case because side effects to function pointers cannot occur.

- for programs with global function pointers, arrays of function pointers, multiple level function pointers or function pointers in structures, the problem becomes $\mathcal{NP}-hard$. For some programs with functions that return function pointers, the problem is also $\mathcal{NP}-hard$.

The data gathered by our study were:

1. the number of function names used as actual arguments. In worst case analysis, knowing that no other function address was taken, we could limit the number of functions that could be called from an indirect call site to this value.

2. the number of formal function pointer arguments in a program.

3. the number of local function pointer variables. *Local variables* are declared within a particular scope; the formal parameters of a function are not considered local variables by our classification.

4. the number of global function pointer variables. This value, if small, would indicate that C programmers tend to use more local and formal function pointer variables; a large value would mean C programs are *hard* to analyze.

5. the number of indirect call sites. This is further partitioned by the type of the called function pointer variables, meaning we calculate the number of indirect call sites where the called function pointer variable is formal, local or global separately.

6. the number of assignments to function pointer variables. This yields the number of program points where modification analysis of these variables need to be done.

7. the number of arrays of function pointers and their dimensionality.

8. the number of structure (or union) variables with at least one function pointer field. These fields may be directly part of the struct or may be fields of structure variables which themselves are directly or indirectly part of this struct variable.

3 The programs analyzed

The programs from which we extracted function pointer declaration and usage information were largely public domain programs. They ranged in size from small (about 500 lines) to substantial (about 100,000) lines of C code. Most of the programs analyzed were serious systems and application programs, widely used in the C programming community. We tried to obtain results from programs targeted to solve varied problems. Among the programs that we analyzed were compilers, editors, interpreters, Unix shells and typesetters: *g++, gcc, troff, perl, bash, DOSE*, etc. Some of the other programs were X applications such as *Xanim, XImage, XMosaic*, etc. We tried various other programs (e.g., the Xlib sources), but could not manage to get them through our analysis tool. Table 1 lists the 24 programs in our study.

4 Implementation

Our tool, developed to extract relevant function pointer information, was based on a tool developed at Siemens Corporate Research called *ptt* [PWC91]. *ptt* is a C language analysis tool which parses a C program, creates a parse tree with a symbol table, generates an intermediate representation and performs various data flow analyses on the program. We used *ptt* as the front-end of our system. Our tool then makes a pass over the symbol table to extract function pointer declaration information. Gathering information on the usage of function pointers was done by a walk over the parse tree. This pass, along with the parse information, uses type information and decides on interesting constructs based on the context within which these appear.

The range of programs that we analyzed was limited due to the implementation of *ptt*. We were restricted to programs that conformed to the old Kernighan and Ritchie C syntax. Moreover, *ptt*'s memory requirements disallowed larger programs from being analyzed.

Some limitations to our tool as it exists today are:

- functions returning function pointers are ignored. The tool does not fail in the event that such constructs appear, but just quietly ignores these occurrences from its calculations.

- the tool performs an incomplete analysis with respect to library routines called from the program being analyzed. It does not analyze these routines and hence does not take into account function pointers used from within those libraries.

5 The output description

The output is split into four tables. Table 1 is a summary of the 24 programs analyzed. Table 2 gives the details of function pointer declarations seen in the various programs. Table 3 details the number of indirect calls and the type of the variables that made these calls. Table 4 details the counts of other use information regarding function pointers. These tables are explained in the following subsections.

5.1 Programs Analyzed

Table 1 lists the programs that were passed through the function pointer analyzer. *Program* is the name of the program. *Lines* gives the size of the program in terms of lines of C code. *Total functions* gives the total number of functions in the program. *Involved functions* gives the number of functions that were in some way involved with function pointers (i.e., these functions either had a local or formal function pointer variable, or made a call to some function pointer variable, or assigned to a function pointer variable). This column was calculated to see if programs have a concentrated use of function pointers in only a small number of their functions or whether usage is widespread throughout the program.

5.2 Declarations of function pointers

Table 2 deals with the questions regarding declaration of function pointers and the scopes within which they are declared. In *Program*, we list the name of the program and its lines of code. Then we have columns split according to type: *global*, *local* (which is the same as automatic), and *formal*. Within each type, we split the column according to dimensionality. We tracked scalar function pointers and one/two-dimensional arrays of function pointers. Columns which were largely empty have not been included; the number of such special occurrences are reported at the bottom of the table. The columns titled *Variables* are not arrays but just simple variables. The columns titled *1D* are one dimensional arrays.

Within the dimensionality columns, two further columns are defined. These are *Scalar* and *Stru*. The entries under *scalar* are function pointers which are not fields of structs. The entries under *Stru* are structure variables with at least one field being of function pointer type. A point to note about entries under the *Stru* column. If a scalar structure contains an array of function pointers as a field, this occurrence is counted under *Scalar*. This is because classification becomes difficult when the struct contains function pointer fields with different arities. Examples are given in Figure 1.

5.3 Calls using function pointers

Table 3 describes the counts we obtained for the indirect call sites. These are again divided according to their type; within the column of each type, we have further subdivisions:

- *scalar* – these are indirect call sites calling simple single level function pointers.
- *struct* – these are indirect call sites where the call is made through a single level function pointer which is a field of a structure variable.
- *deref* – this column is a grouping for calls made through some kind of indirection. The call could be made either through dereferencing a multiple level function pointer or by dereferencing an element of an array of function pointers. Note, that a call through an element of an

Table 1: **Summary of programs analyzed**

Program	Lines	Functions		
		Total	Involved	Percent involved
tp	809	38	6	16
compress	1486	16	1	6
ed	1766	50	10	20
simulator	3594	108	3	3
vn	6138	120	8	7
graphedit	6748	114	8	7
troff	7184	230	14	6
XImage	10375	260	115	44
readline	12683	196	44	22
espresso	13204	355	24	7
XDataSlice	14588	322	67	21
xgobi	17836	404	81	20
Xanim	17878	263	27	10
Mosaic	19735	310	43	14
TimberWolfMC	23854	224	11	5
bash	27202	598	90	15
perl	29111	295	11	4
nethack	29226	680	33	5
rtl	35180	678	150	22
gemacs	47484	1081	174	16
DOSE	49056	934	14	1
gdb	72243	1184	414	35
gcc	81671	1386	246	18
g++	99582	1596	275	17

Table 2: **Summary of function pointer declarations**

Program	Globals				Local		Formal	
	Variables		1D		Variables		Variables	
	scalar	stru	scalar	stru	scalar	stru	scalar	stru
tp (809)	1						2	
compress (1486)	1							
ed (1766)	2				3		2	
simulator (3594)	1		1	2				
vn (6138)	4							
graphedit (6748)	4	40	1			1		
troff (7184)				3			7	
XImage (10375)	45	45	2	1		26	20	12
readline (12683)	14	33		36	3	5	6	8
espresso (13204)	1		1				8	
XDataSlice (14588)	32	21	2	1		39	18	6
xgobi (17836)	46	66	2	1				
Xanim (17878)	31	28	1	1		13		1
Mosaic (19735)	36	31	1	14		3		2
TimberWolfMC (23854)	4		1					
bash (27202)	16	7	6	4	9	8	15	
perl (29111)	4	5			3	7	1	
nethack (29226)	3			4	3	4	6	
rtl (35180)	15	273	1				17	9
gemacs (47484)	26	666			5	1	9	1
DOSE (49056)	4	12			1	1	3	
gdb (72243)	17	266		6	8	229	9	206
gcc (81671)	1	18	3			10	2	
g++ (99582)	3	22	5			12	6	6

- 2 occurrences of global 2D scalar arrays

- 3 occurrence of a local and formal 1D arrays

```
int (*fp1)();                /* this is a scalar variable */

int (*fp2[10])();            /* this is a scalar 1D */

struct x {
   int (*fp3)();
} xv1, xv2[5];               /* xv1 is a stru variable, xv2 is a stru 1D */

struct y {
   int (*fp4[12])();         /* in spite of fp4 being an array */
} yv1, yv2[8];               /* yv1 is a stru variable, yv2 is a stru 1D */
```

Figure 1: Examples of function pointer declarations

array of function pointers which is a field of a struct would be counted as a deref. The reason for deref being a special column is that a level of complexity is added to analyzing programs with calls through multiple level (or array elements) function pointers. Figure 2 shows the details of which constructs would map to which column.

5.4 Other usage information regarding function pointers

Table 4 contains counts of other usage information. These are:

- *Assignments* – this is a count of assignments made to function pointer variables.
- *Actuals* – We already have a count of how many function pointer formal variables exist in these programs. This column is a count of the function-valued actual variables. These are further subdivided into:
  - *Name* – A count of how many function names are passed as actual arguments. We have seen in Section 2 that this is a significant program attribute.
  - *Ptr* – A count of how many function pointers are passed as actual arguments.
  - *Structs* – In C programming, often a pointer to a struct is passed to functions that may not require access to all the fields of the struct. This is just a programming convenience wherein related data is grouped together as a struct and then a pointer to the struct is passed to any function that requires any field from within the struct. Some of these structs may have function pointer fields, but these fields may not be accessed by the called function. This column is a count of structs or pointers to structs with at least one function pointer field in them, which were passed as actual arguments to functions. Without further analysis of the called functions, it is not possible to know whether the

Table 3: **Summary of indirect calls**

Program	Calls						
	Formal		Local		Global		
	scalar	struct	scalar	struct	scalar	deref	struct
tp (809)	1				1		
compress (1486)							
ed (1766)	2						
simulator (3594)						3	
vn (6138)					4		
graphedit (6748)							
troff (7184)	10					1	
XImage (10375)					89		4
readline (12683)	1				10	1	
espresso (13204)	6						
XDataSlice (14588)					75		
xgobi (17836)							
Xanim (17878)				8	13		
Mosaic (19735)					53		
TimberWolfMC (23854)					19		
bash (27202)	6		4	2	9	1	
perl (29111)				3	1		
nethack (29226)	5			3	3		
rtl (35180)	6				8		
gemacs (47484)	8		15		22		1
DOSE (49056)	1				1		
gdb (72243)	2	14	14	25	15		24
gcc (81671)	3					20	
g++ (99582)	7					25	

- 1 occurrence of a formal deref call

- 1 occurrence of a local deref call

```
int (*fp1)();                /* func. ptr declaration */
int (**fp2)();               /* ptr to func. ptr declaration */
int (*fp3[10])();            /* array of func. ptr declaration */
struct {
    int (*fp4)();            /* func. ptr as field of struct */
    int (**fp5)();           /* ptr to func. ptr as field of struct */
    int (*fp6[10])();        /* array of func. ptr as field of struct */
} x;

    /* The calls */
fp1();                       /* counted as scalar */
(*fp2)();                    /* counted as deref */
fp3[1]();                    /* counted as deref */
x.fp4();                     /* counted as struct */
(*x.fp5)();                  /* counted as deref */
x.fp6[1]();                  /* counted as deref */
```

Figure 2: Examples of calls through function pointers

called function actually accessed the function pointer field. Such argument passing may be viewed as a potential function pointer actual.

- *Pairs* – As explained in Section 2, function pointer actuals used in an indirect call lead to potentially imprecise call multigraphs. This column was calculated to know how frequently C programmers use constructs which would potentially add to the imprecision of interprocedural analysis. This column is also subdivided into:
  - *Ptr* – as in the actual arguments, this is a count of function pointers which are actual arguments at an indirect call site.
  - *Struct* – again, as in the actual arguments, these are struct variables or pointers to structs which have at least one function pointer field, which are passed as an actual argument at an indirect call site. This does not imply that the function pointer will be used by the called function.

6 Conclusions

From the tables presented, we draw these conclusions.

1. We observe that calls to global function pointer variables far outnumber the calls to any other kinds of function pointers. This implies that the analysis of a large percentage of C programs

Table 4: **Summary of function pointer usage other than calls**

Program	Assigns	Actuals			Pairs	
		Name	Ptr	Structs	Ptr	Struct
tp (809)	2	8				
compress (1486)	1	2				
ed (1766)	5	14	4			
simulator (3594)	1					
vn (6138)	4	20				
graphedit (6748)	0	134				
troff (7184)	1	14	10			
XImage (10375)	48	196	20	316		11
readline (12683)	72	23	11	11		
espresso (13204)	0	29	4			
XDataSlice (14588)	39	17	17	164		1
xgobi (17836)	0	139		159		
Xanim (17878)	51	23		146		12
Mosaic (19735)	30	59		45		
TimberWolfMC (23854)	6					
bash (27202)	100	63	25	9		
perl (29111)	15	3	5			
nethack (29226)	20	23	4			
rtl (35180)	6	12	13	291		
gemacs (47484)	573	133	10	657		
DOSE (49056)	3	20	2			
gdb (72243)	625	281	13	1115		38
gcc (81671)	60	893		4		
g++ (99582)	90	917	2	22		

will be $\mathcal{NP} - hard$ and that some good approximate aliasing algorithms need to be developed to build call multigraphs.

2. We further observe that structs with pointer fields are frequently used as actuals. This implies further approximation in the solution. This kind of usage may be typical of graphical user interface applications where a callback routine is supplied to the library through a struct.

3. We see a negligible number of arrays of local or formal function pointers and a negligible number of 2-dimensional arrays of function pointers. This seems to imply that primarily global function pointer 1-dimensional arrays are used as jump tables in programs.

4. Function pointer usage seems to vary with to the application area represented by a program. We see high function pointer usage in programs such as X-applications, compilers, debuggers etc. This may mean some kinds of applications require function pointers more than others.

Acknowledgments We wish to thank Sean Zhang whose results provided further motivation and theoretical background to this study. Bill Landi and Hemant Pande were a great help in hacking *ptt*.

References

[PWC91] Michael Platoff, Michael Wagner, and Joseph Camaratta. An integrated program representation and toolkit for the maintenance of C programs. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society Press, October 1991.

[ZR94] Sean Zhang and Barbara G. Ryder. Complexity of single level function pointer aliasing analysis. Technical Report LCSR-TR-233, Laboratory for Computer Science Research, Rutgers University, October 1994.