A Full Program Control Flow Representation for Real Programs

He Zhu and Ian Watson

Manchester University

Abstract. This paper reports on constructing an exhaustive full program control flow framework for precise data flow analysis of real programs. We discuss the problem of ambiguous calling relations in the presence of function pointers. A flow insensitive analysis is suggested and implemented for real C programs.

1 Introduction

This paper discusses the problem of constructing a full *Interprocedural Control Flow Graph* (ICFG) for real programs in the presence of function pointers. Without ambiguous calling relations through function pointers, it is quite easy to build an ICFG for a program consisting of well-defined control constructs, such as those defined by the C language. But in a real program, such as those in the SPLASH-2 suite, the procedure calling relations become complex because there are *pointer addressed call sites*, that is, call sites through function pointers (see Table 1). So, to solve interprocedural flow-sensitive problems for real programs, a precise ICFG which considers function pointers is necessary.

A few *ICFG Frameworks* (ICFFs) have been suggested in the literature for various flow-sensitive problems, such as those in [6, 1, 5, 3]. A few papers have mentioned the function pointer problem [8, 4, 7, 2].

Our ICFF endeavours to design a general framework which is applied to real programs in the presence of function pointers for flow-sensitive interprocedural problems.

2 Interprocedural Control Flow Representation

For simplicity without losing generality, in this paper programs consist of C language statements with low-level control constructs.

Our ICFF combines the intraprocedural control flow representation with the interprocedural call graph which is represented by links connecting call sites and the callees. This idea is similar to those in the literature [6, 1, 5]. But our ICFF is extended to facilitate real C programs by introducing *a general call site node* which can keep a list of its callees and *a skeleton procedure* which will represent *source-absent procedures* for a full ICFF representation.

The control flow of a single procedure is represented by a directed Control Flow Graph (CFG) $G = (V, E)$, where the set of nodes V consists of a *start*

node, an *exit node* and other nodes which correspond to each statement in the procedure. The set E consists of directed edges which connect the nodes in V if the statement for the source node can reach the statement for the destination node without executing any other statement.

To represent an ICFG, interprocedural edges are inserted between call sites and their callees. If call site cs_i calls a procedure p_i, an edge (cs_i, p_i) is created. If a call site is to call a procedure through a function pointer, the callee could be any one of the candidate procedure set to which the function pointer could point. In this case, edges are created from the call site to every procedure in the callee set.

3 An Approach to Computing the Callee Sets

The difficulty of constructing an ICFG is in computing the callee set for each pointer addressed call site. To compute the set precisely, we have to predict all the possibilities to which the function pointer may point. It could be possible by intraprocedural analysis to compute the set if the analysis of the pointer is not interprocedural flow-sensitive. But in most of cases, the function pointer gets its value from an interprocedural argument. So, initially, we have to predict the set using a flow-insensitive analysis.

An initial approach is designed to compute the callee set for a call site without considering any data flow information. It is inevitably conservative, but it exploits other information in the program to make this approach far less conservative than it may appear.

The idea is to use flow-insensitive information such as type information to refine the callee sets. This method was considered not safe in [2], but [7] has shown the possibility of a flow-insensitive pointer analysis by type inference. The method checks all procedures for each call site to match the required procedure pattern. Given a call site c, the computed callee set of c by this initial approach is represented by $\mathcal{I}(c)$.

We designed the following algorithm to compute $\mathcal{I}(c)$.

1. Mark each procedure as an aliased procedure if its address is taken, that is, the procedure address is assigned to a function pointer. There are three cases where a procedure name may possibly be an alias,
   (a) the procedure name appears on the right hand side of an assignment without an argument list,
   (b) the procedure name is used as an actual argument,
   (c) the procedure name is used to initialize a function pointer variable.
2. For each pointer addressed call site c which calls a procedure through a function pointer p,
   (a) set $\mathcal{I}(c)$ to be null.
   (b) check every aliased procedure $proc_i$. If one of the following conditions is true, add $proc_i$ to $\mathcal{I}(c)$. If there is any type cast between two function types, we assume conservatively that the two function types are compatible unconditionally.

- The function type t_p referenced by p is compatible with the function type t_i of the procedure $proc_i$. t_p and t_i are compatible if they have compatible return types and have the same number of arguments and the corresponding argument types are also compatible.
- The function type t_p referenced by p has a null argument list, and its return type is compatible with the return type of $proc_i$, and the type of every formal argument of $proc_i$ is compatible with the type of the corresponding actual argument in the argument list of the call site c. Note that the number of formal arguments could be less than the number of actual arguments.

Apparently, the computed callee set $\mathcal{I}(c)$ includes the real callee set $\mathcal{R}(c)$. It is possible that $\mathcal{I}(c)$ contains redundant procedures which are those that will never be called by call site c. Some of the redundant entries could be reduced by *an incremental approach* which , in theory, improves the set by gradual refinement with the help of an interprocedural flow-sensitive analysis based on the initially created callee set.

4 Experiments

We implemented an ICFG builder based on the SUIF system in C++ using the approach discussed above. It uses linked lists of edges. The start node and the exit node are combined into one *start-end node* for convenience.

In a real program, source lines of procedures are not always available, such as for external and library procedures. If a source-absent procedure has any call site in its body, interprocedural links are lost if the procedure is not treated specially. For such a procedure, we define a *skeleton procedure* which is not necessarily complete, but describes the correct interprocedural calling relations.

Table 1 lists the statistics from our experiments on some programs from the SPLASH-2 suite.

Program	water	barnes	raytrace	fmm	radiosity	ocean
lines	1776	2303	10022	3847	22118	4712
call sites	153	242	697	463	663	210
procedures	38	82	163	120	208	36
procs not called	2	10	44	7	25	1
aliased proc	1	2	1	12	24	1
pointed call sites	1	1	11	7	10	1
pointed sites(no callees)	0	0	4	0	9	0
interproc edges	153	242	693	495	677	210
interproc edges(ambi sites)	1	1	7	39	24	1

Table 1. Statistics on Programs in SPLASH-2

It shows that our FCFG builder works on programs of various sizes, from

1776 lines to 22118 lines. The table also reveals that, on average, each procedure is called at 3 to 6 call sites. Usually only a few procedures, less than 1% , are aliased, but in some programs, such as *fmm*, up to 12% of procedures are aliased. Among the call sites, less than 2% are pointer addressed call sites. The number of interprocedural edges for a pointer addressed call site could be greater than one because a callee site could have a few aliased procedures. We have also noticed that a real program could have some procedures which are never called and there are call sites which call no callees because programs developed at various stages could have test codes which remain in the final code.

5 Conclusions

This paper has addressed the problem of building an ICFF of real programs in the presence of function pointers. We suggested a flow-insensitive initial approach to compute callee sets. The approach has been implemented in our FCFG builder and some preliminary results have shown that the approach is consistent with that in [7]. Further research is required on refinement of callee sets and improvement of the approach for a better performance.

References

1. D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN Conf. on Prog. Lang. Design and Imple.* June 1988.
2. M. Emami and et al. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 242–256, New York, NY, USA, June 1994. ACM Press.
3. M. W. Hall and et al. Fiat: A framework for interprocedural analysis and transformation. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 522–545, Portland, Oregon, August 12–14, 1993. Springer-Verlag.
4. Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.
5. Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.
6. E. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth annual ACM Symposium on Principles of Programming Languages*, pages 219–230. ACM, ACM, January 1981.
7. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. *Lecture Notes in Computer Science*, 1060:136–??, 1996.
8. William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.