Accurate Call Graph Extraction of Programs with Function Pointers Using Type Signatures

Darren C. Atkinson
Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053-0566 USA
datkinson@scu.edu

Abstract

Software engineers need to understand programs in order to effectively maintain them. The call graph, which presents the calling relationships between functions, is a useful representation of a program that can aid understanding. For programs that do not use function pointers, the call graph can be extracted simply by parsing the program. However, for programs that use function pointers, call graph extraction is nontrivial. Many widely used C programs utilize function pointers for efficiency and ease of implementation. We present a technique called type signature filtering for improving call graph extraction in the presence of function pointers. Filtering can be accomplished in a single pass after pointer analysis is complete, making it reusable across different analyses. Our results show that for many programs our technique yields a call graph that is nearly identical to the true call graph, even if a naive pointer analysis is used.

1. Introduction

1.1. Motivation

As a society, we rely increasingly on software that is itself becoming more complex and interconnected. As such, errors in software systems become ever more apparent, worrisome, and costly to the general public. A recent study by the National Institute of Standards and Technology (NIST) estimated that software errors cost the American economy at least sixty billion dollars a year, and that 80% of all software development cost is applied toward maintaining in software [16].

Software maintenance itself can take a variety of forms. A programmer may need to incorporate an enhancement requested by the customer. The software may need to be adapted for a new architecture or platform. Defects in the design or implementation may need to be corrected. Finally, the engineer may simply wish to restructure the system, improving its design and organization, to ease incorporation of future changes.

For all of these applications, the engineer needs to thoroughly understand the system in order to correctly maintain it. A lack of understanding could lead to additional defects being introduced. For example, anticipating the effect of a proposed change such as adding a new feature requires knowledge of the control-flow and data-flow properties of the program, if the change is to be made without introducing errors. Ideally, software engineers would have adequate documentation to assist them in these tasks. Requirements, specifications, and design documents present information about the program in ways that are easier for the engineer to understand, as compared with hand-examination of the source code. Unfortunately, this documentation is often not available or is out of date.

To assist the software engineer in understanding a system, static analysis tools such as program slicing tools and invariant checkers have been proposed as a solution. For example, a backward program slicer [24] computes the set of statements that may have affected the value of a given variable, which may aid programmers during debugging. As another example, an invariant checker infers facts about the state of the program and checks those facts against assertions provided by the programmer [17]. Such tools analyze the program source, computing information about the program, and present that information in a way that is most useful to the tool user.

1.2. Call graphs

A useful analysis is the extraction of the *call graph* of a program. The call graph is simply a visual representation of the calling relationships between functions or procedures. Each function is represented as a graph node. Figure 1

Copyright 2004 IEEE. Published in the Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC-2004), November 30–December 3, 2004, Busan, Korea. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE.

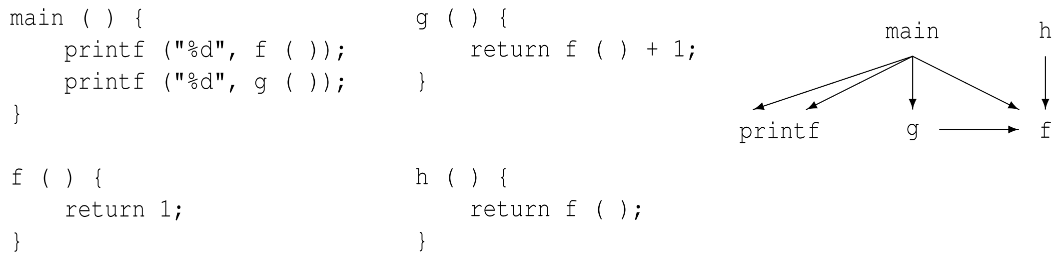

Figure 1. Example program and its call graph.

shows an example C program and its call graph. If function a calls function b then $a \rightarrow b$ is an edge in the graph. Thus, the call graph can be used to determine which functions are called by a particular function, as well as which functions call a particular function. Both queries are of interest to the software engineer when trying to understand a system.

Not only is the call graph useful alone, but the proper determination of calling relationships is a prerequisite for other analyses as well. For example, determining how a proposed change to one module of the system will affect the rest of the system requires the knowledge of which functions call which other functions. Accurate call graph extraction is also useful in software testing, since one common goal of testing is, for example, to ensure that each function is executed on at least one test run [15].

Unfortunately, the problem of call graph extraction is nontrivial. Modern programming languages allow functions to be used in ways other than just in traditional call expressions. For example, most languages allow functions to be passed as parameters to other functions. Languages such as C and C++ allow the addresses of functions to be taken and used as pointers. Many significant, large scale programs such as GCC [21] rely heavily on function pointers to implement dispatch tables (tables in which the key is an integer value designating an operation and the corresponding value is the address of a function that performs that operation), as shown in Figure 2, or to implement object-oriented dispatch. In effect, functions and procedures are used more as "first-class" objects such as integers or pointers than solely through simple function and procedure calls. These uses are necessary for efficiency and often greatly simplify design. However, they complicate the construction of the program's call graph since the calling relationships are not apparent simply by examining the lexical and syntactic structure of the program.

Most program analyses account for these problems by first computing the sets of pointer values (points-to sets) for functions, and then using the pointer values to construct an accurate call graph. Although flow-sensitive and context-sensitive points-to algorithms potentially produce the most precise results, they typically have running times that are not acceptable for use on larger programs (e.g., programs with at least 50,000 lines of code). For example, many algorithms have $O(n^3)$ running times, where n is the number of lines of code [1]. Therefore, less expensive and less precise analyses are often used that trade precision for performance to enable analysis of large or complex programs. These analyses are typically not fully flow-sensitive or context-sensitive [20, 25]. For example, efficient, near-linear time points-to algorithms are well known [6, 22].

Unfortunately, the results of these inexpensive analyses are often less than desirable since they are overly conservative. That is, they must be correct for any input and execution path of the program. In addition, for C programs they have to make additional conservative assumptions about the program due to C's weak type system. The resulting points-to sets are therefore often quite large. For example, the resulting points-to data may indicate that a call made through a function pointer could possibly call any function in the system whose address is taken. This leads to many "false" edges in the call graph, hindering program understanding.

```
# define SIN 0
# define COS 1
   ...
# define POW 10

struct {
  double (*func) ( );
  int args;
} functab [ ] = {
  {&sin, 1},
  {&cos, 1},
  ...
  {&pow, 2},
};
```

(a)

```
n = functab [i].args;
f = functab [i].func;

x = pop ( );

if (n == 1)
  return (*f) (x);

y = pop ( );

if (n == 2)
  return (*f) (x, y);

return 0;
```

(b)

Figure 2. Example of dispatch tables: (a) a simple dispatch table, and (b) its use.

Level	Construct	Rule
Weak	Qualifiers	Ignored
	Arguments	Number of actuals at least number of formals
	Specifiers	One is assignable to other; structure tags need not match
	Declarators	Match at outermost level only, unless one is pointer and other is integer
Strong	Qualifiers	Ignored
	Arguments	Number of actuals equal to number of formals
	Specifiers	One is assignable to other; structure tags must match
	Declarators	Match at all levels, unless one is pointer to void and other is pointer

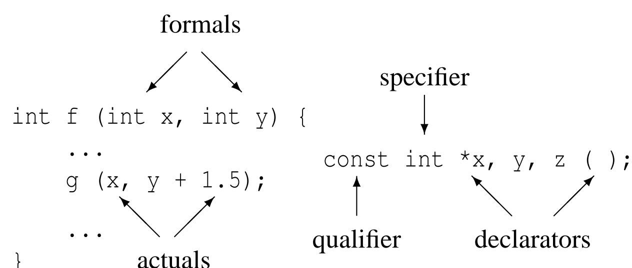

Figure 3. Filtering rules for both weak and strong type signature filtering.

In this paper, we present a technique for improving the points-to sets computed by points-to algorithms. Our approach, called *type signature filtering*, uses the declared program types to filter the resulting points-to sets in order to eliminate functions that could not be called during any legal program execution. Our technique can be implemented for any points-to analysis, since it consists of a simple post-processing step that is independent of the manner in which the points-to sets are themselves constructed. However, since violation of the type system (i.e., type casting) is possible in languages such as C, our approach may in fact be unsafe for some programs in that some calls may be omitted from the call graph when they in fact can occur. However, for programs that conform to standard calling conventions (e.g., passing the correct number of arguments), type signature filtering is safe.

Our results show that for many typical C programs, filtering is safe and can dramatically reduce the number of graph edges. In many cases, results from a poor points-to analysis can be made equivalent to the results from better analyses by using signature filtering. Often, the resulting call graph is identical to the true graph obtained through manual examination of the source code. For a few programs, filtering is unsafe. However, even in these cases, the resulting call graph is closer to the true graph than if no filtering had been performed, and can therefore still be quite useful to the software engineer for the purposes of program understanding and maintenance. Furthermore, the fact that filtering is unsafe can tell the software engineer interesting things about the program (e.g., the program itself may actually contain errors). We found errors in two commonly used programs by comparing the filtered graph with a graph constructed by hand. (In practice, one could use an approximation of the actual call graph obtained through profiling.) In particular, this paper makes the following contributions:

- Using type information for function pointers is generally safe for C programs.
- Using type signature filtering can dramatically increase the precision of the underlying points-to analysis used.
- Even when filtering may be unsafe, the resulting call graph is in fact closer to the actual call graph than the unfiltered call graph. That is, the difference between the filtered graph and the actual graph is less than the difference between the unfiltered graph and the actual graph.
- The fact that filtering is unsafe can provide insights into discovering complex program behaviors. We found errors in widely used programs using this technique.

- We constructed the true call graph by hand for a variety of C programs commonly used in the research community and provide detailed data from our experiments.

The remainder of this paper is organized as follows. In Section 2, we discuss our technique and its implementation in a call graph extraction tool for C programs. Then, we present our experimental setup in Section 3. Section 4 presents and discusses our results, detailing where type signature filtering is unsafe and what we learned about the programs used in our experiments. Finally, Section 5 discusses related work, and in Section 6 we present our conclusions and discuss future work.

2. Type signature filtering

2.1. Approach

We use type information to improve the accuracy of the points-to sets for function pointers. The points-to sets for other types of pointers are unaffected. In general, using type information for C and C++ programs is unsafe since the type system can be violated by means of type casts. However, type casting of function pointers is uncommon, and the type information itself is easy to compute. Such information is often already available in the program representation (e.g., the symbol table or syntax tree). Consequently, type information is an attractive alternative to other techniques such as implementing a more sophisticated and expensive points-to algorithm, which still may not yield an improved call graph, as we shall see in Section 4.

Another advantage of type *filtering* is that it can be performed as a post-processing step after points-to analysis is complete. This separation allows our technique to be used with any points-to analysis. It also does not further complicate an existing points-to algorithm, reducing the chance of introducing errors into the algorithm. One disadvantage of filtering is that the points-to algorithm may require more time and space than if type information was used during the analysis itself. However, many commonly used points-to analyses run in near-linear time and space, so any savings are likely to be minimal.

We compute type signatures for all function definitions and compare them against the signature implied by the function call through the function pointer. The points-to set of the pointer is filtered in that only functions whose signatures satisfy the matching rules are retained. Functions whose signatures do not match are removed. Type signatures are similar to function prototypes. The ANSI standard provides function prototypes, which provide additional typing information for static semantic checking by ensuring the type and number for formal and actual arguments agree. However, prototypes are not required and many programs were developed before the standard, but may themselves be ANSI-compliant. Therefore, the signatures are computed from the actual function definitions and function calls. Any function prototypes are in fact ignored.

```
void (*p) ( );            int g (int x) {
int (*q) ( ), y;            return x;
                          }
int main ( ) {
  ...                     int h (int x, void *p) {
  (*p) (1);                 return x + *(int *) p;
  (*q) (2, "a");          }
  (*q) (3, &y);
}                         int i (int x, char *p) {
                            return *p + x;
void f (int x) {          }
  y = x;
}
```

Figure 4. Example using function pointers.

For flexibility, two levels of filtering are supported: weak and strong. Weak type signature filtering assumes that the program is only weakly compliant with ANSI standards, i.e., the program obeys pre-ANSI conventions as loosely defined by [10] and [9]. Strong type signature filtering assumes that program is strictly ANSI-compliant. The rules for both levels of checking are shown in Figure 3. For example, under strong filtering, the declarators must match at all levels, unless one is a pointer to `void`, which is the “generic” pointer type according to the standard. Under weak filtering, only the outermost declarator must match. Older programs used pointer to `char` as the generic pointer type, so requiring that the declarators match at all levels would be too restrictive for these programs. As another example, under weak filtering the number of actual arguments must be at least the number of formal arguments, i.e., it is permissible to call a function with too many, but not too few, arguments. Under strong filtering, the number of actual and formal arguments must match. The C language does support functions with a variable number of arguments, and these functions are treated specially during filtering by ignoring the variable argument list. In practice, however, pointers to such functions are rare and did not occur at all in the programs in our test suite.

Consider the example program in Figure 4. This program makes use of two function pointers, both of which have a declaration, but not a complete prototype, and none of the functions defined have a prototype. The type signature for `f ( )` is simply $int \rightarrow void$, and the signature for `h ( )` is $int \times pointer(void) \rightarrow int$.

Let us assume that the four functions, `f ( )`, `g ( )`, `h ( )`, and `i ( )`, have all been merged into the same points-to set

Expression	None	Weak	Strong
`(*p) (1);`	{f,g,h,i}	{f}	{f}
`(*q) (2, "a");`	{f,g,h,i}	{g,h,i}	{h,i}
`(*q) (3, &y);`	{f,g,h,i}	{g,h,i}	{h}

Table 1. Effect of type signature filtering on the program in Figure 4.

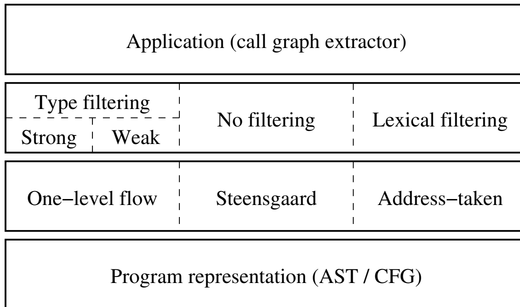

Figure 5. Layered software architecture for the call graph extractor.

and that both `p` and `q` point to this set, as shown in the second column of Table 1. By applying the rules for strong type filtering to the signatures of the function call and definition, the first function call in `main()` can only refer to function `f()` since the other three functions return an `int` and `p` is declared to return `void`. The second call can refer to either function `h()` or function `i()` since they both require two arguments and a string is assignable to the `void` pointer argument in function `h()`. The third call can only refer to function `h()` since a pointer to `int` is assignable to a `void` pointer, but not to a `char` pointer (i.e., string literal). The results with strong prototype filtering enabled are shown in Table 1. The results after applying the rules for weak filtering are also shown in the table. Note that `g()` is included in the points-to sets because it is now permissible to pass extra arguments to a function.

2.2. Tool design

As mentioned, filtering the points-to sets has the added advantage that the points-to algorithm itself is unaltered. Figure 5 shows the software architecture for our call graph extractor, `cgraph`, which is part of the ICARIA tool set for analyzing C programs [14, 2, 5]. As shown in the figure, the type signature filtering component is separate from the points-to analyses. Three different points-to analyses have in fact been implemented: a simple address-taken scheme in which a pointer may point to any function whose address has been taken, Steensgaard's near-linear time algorithm that computes equivalent classes of memory locations [22], and Das's one-level flow algorithm that is an improvement on Steensgaard's [6]. The one-level flow algorithm has been shown to yield results as precise as Andersen's well-known algorithm [1]. The results from Andersen's algorithm, in turn, have been shown [11] to be comparable to results from some other well-known pointer analysis algorithms [11, 18]. Having the filtering component separate was key to successfully implementing and testing each points-to algorithm.

The call graph application interfaces with the points-to data through the filtering layer. The application can request no filtering, weak type signature filtering, or strong type signature filtering. In order to determine the effectiveness of our technique, we also computed the true points-to sets for function pointers by examining the code by hand. These points-to sets are used to extract the actual or true call graph, which can be used as a baseline for evaluation. During this process, we discovered that many programs use naming conventions for functions. For example, for the `find` application, any call through a pointer named "`parse_function`" resolved to any function whose name began with "`parse_`". Therefore, to make specification of the true points-to sets easier and more compact, we decided to exploit such conventions by adding a module for lexical filtering, as shown in Figure 5. Both the function call expression and set of called functions can be specified as regular expressions.

3. Experiments

To evaluate our technique, we computed the call graphs of several C programs. We chose to include programs from the SPEC 2000 benchmark suite since they are of considerable size, perform a variety of different computations (graphics, compression, mathematical computations, simulation), and are used in practice. (They are submitted to the SPEC consortium based on their relevance and representativity of actual computing practice.) We also included programs that others have used in their call graph experiments [13, 3], many of which are commonly used UNIX utilities. Although there is no general consensus on what constitutes a typical C program, we have tried to include a sufficient number and variety of programs to substantiate our claims.

Table 2 provides sizes and descriptions of programs used in the experiments. The number of lines of code was determined simply by using the UNIX `wc` utility. The number of indirect call sites indicates the number of static occurrences of functions calls through pointers.

Program	Lines of code	Indirect call sites	Source	Description
gzip	7757	4	SPEC 2000	compression
diff	11755	3	GNU (v2.7)	file comparison
grep	13084	18	GNU (v2.4.2)	pattern matching
find	13122	22	GNU (v4.1)	filesystem searching
ammp	13263	24	SPEC 2000	molecular dynamics
m4	14007	5	GNU (v1.4)	macro processing
unzip	17759	305	Info-ZIP (v.5.50)	compression
less	18305	4	GNU (v358)	text file viewing
mesa	49701	671	SPEC 2000	graphics
burlap	49845	18	FELT (v3.05)	finite element solver
vortex	52633	15	SPEC 2000	object-oriented database
perlbmk	58221	57	SPEC 2000	text processing
gcc	205743	140	SPEC 2000	compilation

Table 2. Sizes and descriptions of programs used in the experiments.

Since we will evaluate the merits of our technique (Section 4) by comparing the number of edges in a call graph, a discussion of which edges are included and counted is appropriate. Many studies do not discuss how edges are counted, and others such as [13] do not provide edge counts at all, but rather discuss factors of improvement, making comparison between techniques difficult. To assist in reproducing our results, we discuss here several issues in call graph construction:

- **Multiple calls:** Multiple calls to a function b from a result in multiple edges $a \rightarrow b$ in the graph. For example, in Figure 1, `main()` calls `printf()` twice, so two edges are included. Thus, the call graph is a multi-graph.

- **Undefined functions:** Calls to undefined functions are included in the graph. For example, in Figure 1, `printf()` is undefined, but calls to it are included.

- **Unreachable functions:** If functions are unreachable from `main()`, then the call graph will have multiple roots. For example, in Figure 1, `h()` is unreachable. The edge counts presented in Section 4 include *only reachable* functions. Programmers may leave functions for debugging and testing in the source code. These functions are unreachable and should not be counted as part of the program proper. Therefore, they are excluded.

Therefore, if the program presented earlier in Figure 1 were used in the experiments, its call graph would have an edge count of five.

4. Results

4.1. Experimental data

For each program, we built its call graph using each of the three points-to algorithms discussed in Section 2.2. In turn, for each algorithm, we counted the number of edges in the graph constructed using the unfiltered points-to data, using weak type signature filtering, and using strong type signature filtering. We also computed the number of edges in the "actual" or "true" call graph through hand examination of the code. To the best of our knowledge, this manually calculated graph is correct and more precise than any known points-to analysis could yield. For example, `gcc` uses multi-dimensional arrays of function pointers, where one index is (almost) always a constant. In our construction, we were able to take advantage of the fact that one index is either a constant (or a variable whose value is determined to be a constant) and separate the individual array dimensions, something that a pointer analysis would be unlikely to do.

The numeric data is shown in Table 3. Entries in boldface indicate where strong type signature filtering may be unsafe. If the number of edges in the filtered graph is less than the number of edges in the true graph, then filtering is obviously unsafe. This fact is most apparent for `gcc`. However, even if the edge count is greater, filtering may still be unsafe. Filtering tries to remove "false" edges from the graph, but may erroneously also remove true edges. To ensure safety, the actual call graph must be a subgraph of the filtered graph. For programs other than those indicated, we determined that filtering was safe by performing this subgraph check.

Figure 6 shows the effect of weak type signature filtering on call graph size. The graph shows the number of graph edges normalized as the percentage of the unfiltered

Program	Actual	One-level flow strong	One-level flow weak	One-level flow none	Steensgaard strong	Steensgaard weak	Steensgaard none	Address-taken strong	Address-taken weak	Address-taken none
gzip	307	307	307	307	307	307	307	307	378	378
diff	620	620	620	620	620	620	620	620	625	648
grep	712	712	712	741	712	727	780	712	736	794
find	1116	1116	1116	1116	1116	1716	1765	1116	2316	2964
ammp	1545	1545	1545	1545	1545	1545	1545	1556	1556	1953
m4	**1034**	**1033**	1034	1034	**1033**	1034	1034	**1036**	1047	1206
unzip	1351	1352	1651	2257	1352	1651	2257	1352	1969	2886
less	1688	1688	1688	1688	1688	1688	1688	1688	1701	1731
mesa	1673	13915	77831	171297	13915	77831	171297	13915	77889	171681
burlap	**4133**	**4268**	5466	6632	**4268**	5466	6632	**4273**	5473	6646
vortex	**6847**	**6645**	6895	7126	**6645**	6895	7126	**6657**	7112	7296
perlbmk	9060	9312	16438	18467	9312	16438	18467	9342	27948	33105
gcc	**24783**	**24119**	34329	40588	**24119**	34331	40884	**24628**	38806	54859

Table 3. Number of call graph edges. Boldface entries indicate where strong filtering may be unsafe.

address-taken data, i.e., the graph with the largest number of edges. The outlined bars show the unfiltered data for each algorithm. For some applications, weak filtering has little effect on the results from Steensgaard's and the one-level flow algorithms. This is because the graphs generated by these algorithms are already quite close to the actual call graph, so there is little room for improvement. However, it generally improves the naive address-taken scheme by a greater factor, thus closing the gap with the other more sophisticated algorithms.

Figure 7 shows the effect of strong type signature filtering on call graph size. For the majority of the applications, the resulting call graphs are now almost identical to the true call graphs. Only, `mesa` still shows a considerable gap, due to the fact that most of its function pointers are in structures and none of the analyses differentiate structure fields (i.e., the structure is treated as a single object). Our results are twofold. First, strong type filtering can dramatically improve the results of points-to analyses, with regard to function pointers and call graph size. Second, strong type filtering can overcome the simplicity of algorithms such as the address-taken scheme.

Consequently, type information appears to be of greater benefit to the accurate determination of points-to sets than the complexity and sophistication of the algorithm, at least for function pointers. Although this result has been shown for type safe languages such as Modula-3 [7] and for call graph construction in type safe languages such as Java [23], using type information to improve the results of points-to analyses of C programs as they pertain to call graph construction has not generally been considered.

4.2. Safety considerations

In Table 3, four of the applications, `m4`, `burlap`, `vortex`, and `gcc`, are indicated as programs for which strong type filtering is unsafe. However, for all four programs, the call graph constructed using strong prototype filtering is at least as accurate as the unfiltered graph, and is usually better. A closer examination of the applications yielded some interesting discoveries. For `m4`, the program itself is incorrect. A function that can be used during debugging (for dumping the contents of the symbol table) is not passed the correct number of arguments. Similar problems exist in `gcc`, where extra arguments are passed to functions or where too few arguments are passed, but the functions called actually do not use the missing arguments, so no run-time errors occur. We found it surprising that programs as "robust" and old as `m4` and `gcc` actually contain errors, although they may be "harmless".

However, for `gcc`, the errors described do not account for the entire difference between the actual and type-filtered call graphs. As previously discussed, the actual call graphs were determined by hand examination of the code. This was particularly difficult for `gcc`, given its size and complexity. After a thorough examination of the source, we found that the hand-constructed graph was actually too conservative. In the program, there are several places where a function in a dispatch table (cf. Figure 2) is called. However, based on the number of arguments passed in the call, only certain functions could be called and operate without error. (This was not a case where the missing arguments were unused by the called function.) Presumably the index (a machine opcode) to the table selects a correct subset of the functions that have a related operation, such as moving data, since `gcc` works correctly. However, we had no way of knowing

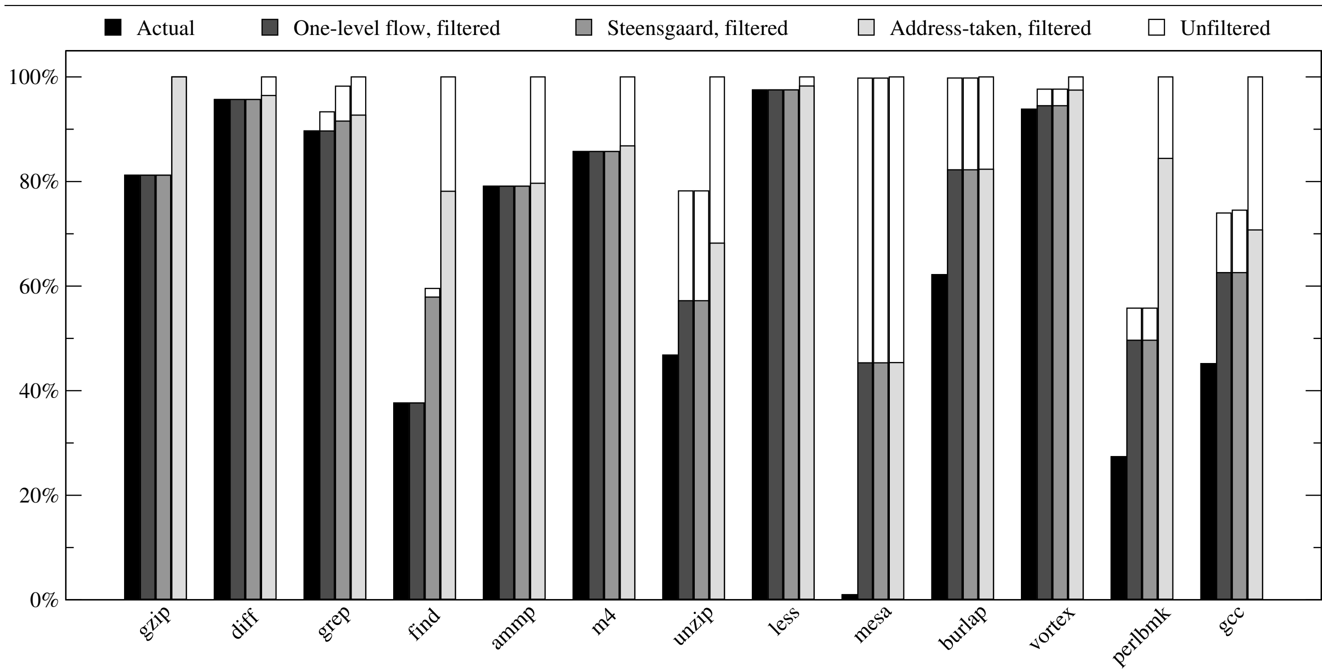

Figure 6. Effect of weak type signature filtering on call graph size.

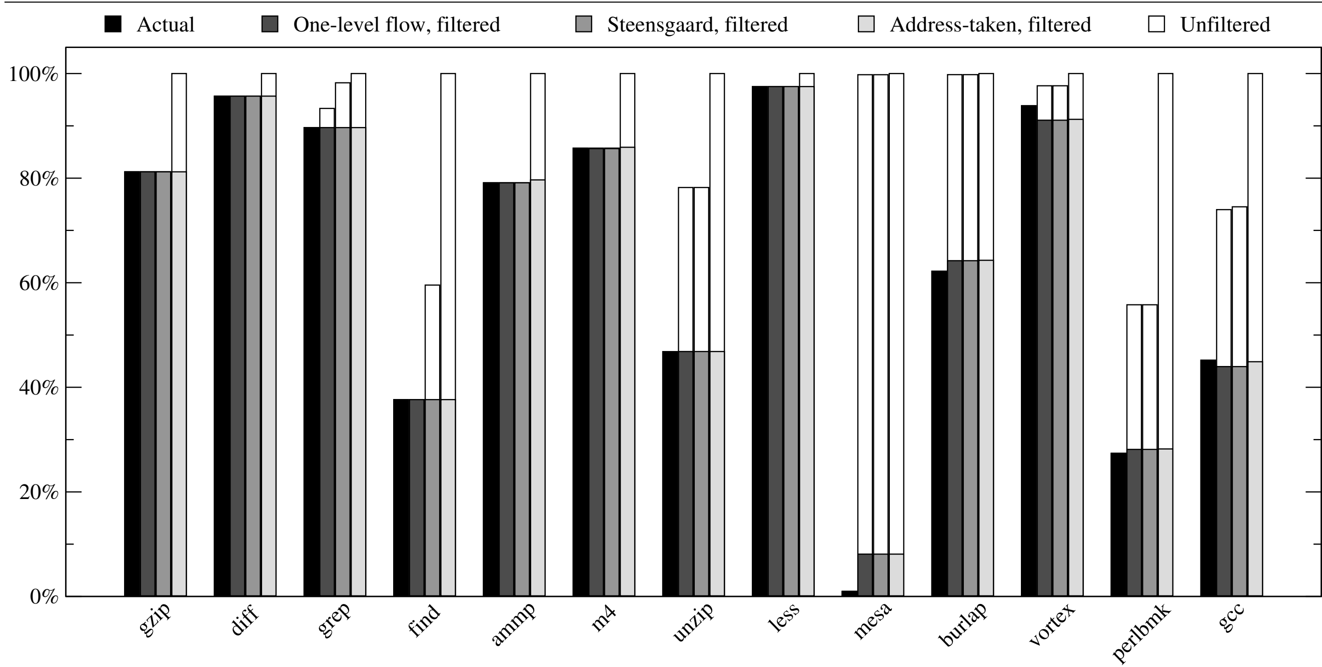

Figure 7. Effect of strong type signature filtering on call graph size.

this during our examination of the code. Only after examining the results from strong type signature filtering did we discover this fact.

For `burlap`, there was one call site in which additional arguments were (harmlessly) passed to the called function. For `vortex`, strong prototype filtering is actually inappropriate because the program does not conform to ANSI conventions. We determined that the program is written in an object-oriented style using inheritance and polymorphism. In C, these can be achieved simply by ensuring that the first fields of two structure types correspond. A pointer to the "derived" class can be passed wherever a pointer to the "base" class is expected, since the derived class has the same initial structure as that of the base class. However, such a call clearly violates the rules for strong prototype filtering, since the two pointer types are unassignable.

5. Related work

The use of type information for function pointers to improve program slicing [24] is discussed in [4]; however, the effect on the call graph itself is not discussed. Mock et al. [14] discuss the use of dynamic points-to data in program slicing and report that accurate points-to data for function pointers is more important than accurate points-to data for other variables. Dynamic function pointer data could be used to inexpensively construct a call graph for comparison with graphs obtained using type signature filtering. If an edge is present in the dynamic call graph but absent in the filtered graph, then filtering is obviously unsafe.

Both [25] and [8] discuss context-sensitive alias and points-to algorithms for C programs. Since the algorithms are context-sensitive, they must resolve the points-to sets for function pointers "on-the-fly" while computing the points-to data for other variables. The latter work discusses the use of type information and declares it unsound (which we do not dispute).

Type filtering in C++ is discussed in [19], and [12] discusses points-to analyses for Java. For object-oriented languages such as C++ and Java, the class hierarchy can be used to help resolve the set of called functions for virtual methods invoked through dynamic dispatch (i.e., through a function pointer). Finally, [7] discusses the use of type information to compute alias information in Modula-3, a type-safe language.

6. Conclusion

6.1. Discussion

Software systems are too difficult to understand without the aid of tools. The call graph is a useful representation of a program that can greatly aid understanding. An accurate call graph is also necessary before any subsequent interprocedural data-flow analysis can be performed. However, call graph extraction of a C program is difficult if the program uses function pointers. A pointer analysis must first be performed to compute the points-to sets for the function pointers. Although many practical points-to algorithms are available, the results are typically less than satisfactory. The resulting call graph contains too many "false" edges to be generally useful.

We have developed a technique called type signature filtering for improving the results for function pointers from points-to analyses. Our technique is implemented in a separate pass over the points-to sets, once pointer analysis is complete, making it reusable across many different pointer analyses. Although the use of type information may be unsafe for some C programs, for many programs using type information for function pointers is safe.

Our results showed that by filtering the sets for function pointers, an accurate call graph can be obtained. For the majority of programs in our test suite, the call graph constructed using the filtered sets was virtually identical to the actual call graph that was obtained by hand. Furthermore, filtering was so effective that the results from a naive points-to algorithm were made equal to those from more sophisticated and complex algorithms. For deriving an accurate call graph, type information is therefore more important than the intelligence of the underlying pointer analysis.

For some programs, filtering can be unsafe. However, a close approximation of the true call graph can still be constructed. If the program is not written in strict accordance with ANSI standards, a weaker version of the filtering rules can be used. For those programs where filtering was unsafe, we were able to gain insight into design and implementation of the programs. We believe that the use of type information is an attractive method for improving the results from pointer analyses in the areas of system understanding and maintenance.

6.2. Future work

Strong filtering is unsafe on many programs due to the restriction on the number of actual and formal arguments. In contrast, weak type signature filtering is generally safe and does not have this restriction. Therefore, a version of strong filtering that relaxes this restriction may be quite effective and yet still safe for most programs.

Although lexical filtering was useful as a compact way of specifying the points-to sets for the actual call graph, it is only useful if naming conventions are used. Furthermore, the tool user must be familiar enough with the program to know those conventions. Therefore, for the purposes of program understanding, lexical filtering is likely not generally

useful. It may however be useful for other purposes, such as programmer-directed optimizations during compilation.

Finally, since type signature filtering was so effective at improving call graph extraction for the programs in our test suite, using type information to filter all pointers seems attractive. Such filtering would obviously be unsafe for most C programs, but could be used to reflect on the effectiveness of points-to algorithms. Furthermore, a pointer analysis for C could be developed that assumes type safety, or at least that the type system is violated only in predictable, stylized ways, rather than assuming that using type information is completely unsafe.

7. Acknowledgments

Thanks to Bill Griswold, Markus Mock, and Van Nguyen for their helpful comments, and to Manuvir Das for providing his implementation of the One-Level Flow algorithm for use as a starting point for our implementation.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Department of Computer Science, May 1994.

[2] D. C. Atkinson. *The Design and Implementation of Practical and Task-Oriented Whole-Program Analysis Tools*. PhD thesis, University of California, San Diego, Department of Computer Science & Engineering, Apr. 1999.

[3] D. C. Atkinson. Call graph extraction in the presence of function pointers. In *Proc. 2002 Int. Conf. on Softw. Eng. Res. Prac.*, pages 579–584, Las Vegas, NV, June 2002. CSREA Press.

[4] D. C. Atkinson and W. G. Griswold. Effective whole-program analysis in the presence of pointers. In *Proc. 6th ACM Int. Symp. on Found. Softw. Eng.*, pages 46–55, Lake Buena Vista, FL, Nov. 1998. ACM Press.

[5] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proc. 2001 Int. Conf. on Softw. Maint.*, pages 52–61, Florence, Italy, Nov. 2001. IEEE Comput. Soc. Press.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Proc. 2000 ACM Conf. on Program. Lang. Des. Impl.*, pages 35–46, Vancouver, BC, June 2000. ACM Press.

[7] A. Diwan, K. S. McKinley, and J. F. B. Moss. Type-based alias analysis. In *Proc. 1998 ACM Conf. on Program. Lang. Des. Impl.*, pages 106–117, Montreal, Que., June 1998. ACM Press.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. 1994 ACM Conf. on Program. Lang. Des. Impl.*, pages 20–24, Orlando, FL, June 1994. ACM Press.

[9] S. C. Johnson. Lint: A C program checker. Computer Science Technical Report 65, AT&T Bell Laboratories, Murray Hill, NJ, July 1977.

[10] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1st edition, 1998.

[11] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. 7th Eur. Softw. Eng. Conf. and 7th ACM Symp. on Found. Softw. Eng.*, pages 199–215, Toulouse, France, Sept. 1999. Springer-Verlag.

[12] D. Liang and M. J. Harrold. Evaluating and extending flow-insensitive and context-insensitive points-to analyses for Java. In *Proc. 2001 ACM Work. on Program Anal. Softw. Tools Eng.*, pages 73–79, Snowbird, UT, June 2001. ACM Press.

[13] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Proc. 2nd IEEE Int. Work. on Source Code Anal. Manipulation*, pages 155–162, Montreal, Que., Oct. 2002. IEEE Comput. Soc. Press.

[14] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Improving program slicing using dynamic points-to data. In *Proc. 10th ACM Symp. on Found. Softw. Eng.*, pages 71–80, Charleston, SC, Nov. 2002. ACM Press.

[15] G. J. Myers. *The Art of Software Testing*. Wiley, New York, NY, 1979.

[16] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, Strategic Planning and Economic Analysis Group, Gaithersburg, MD, May 2002.

[17] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proc. 4th ACM Symp. on Found. Softw. Eng.*, pages 93–105, San Francisco, CA, Oct. 1996. ACM Press.

[18] H. D. Pande, W. A. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Trans. Softw. Eng.*, 20(5):385–403, May 1994.

[19] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for optimizing compilers. In *Proc. 1996 ACM Conf. on Program. Lang. Des. Impl.*, pages 54–67, Philadelphia, PA, May 1996. ACM Press.

[20] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. 24th ACM Symp. on Princ. Program. Lang.*, pages 1–14, Paris, France, Jan. 1997. ACM Press.

[21] R. M. Stallman. *GCC Reference Manual*. Free Software Foundation, Cambridge, MA, 1991.

[22] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. 23rd ACM Symp. on Princ. Program. Lang.*, pages 32–41, St. Petersburg Beach, FL, Jan. 1996. ACM Press.

[23] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 2000 ACM Conf. on Object-Oriented Program. Syst. Lang. Appl.*, pages 281–293, Minneapolis, MN, Oct. 2000. ACM Press.

[24] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, SE-10(4):352–357, July 1984.

[25] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. 1995 ACM Conf. on Program. Lang. Des. Impl.*, pages 1–12, La Jolla, CA, June 1995. ACM Press.