Complexity of Single Level Function Pointer Aliasing Analysis*

Sean Zhang Barbara G. Ryder

Department of Computer Science
Rutgers University, Piscataway, NJ 08855
email: {xxzhang, ryder}@cs.rutgers.edu

Abstract

We present a definition of the function pointer aliasing problem for single level function pointers, according to a *new* approximation of possible program execution for interprocedural analyses in the presence of calls through function pointers. We have classified the complexity of the problem as either polynomial or NP-hard, with respect to various program constructs affecting function pointer aliasing. We present our problem classification and give brief proofs for a polynomial case and a NP-hard case.

1 Introduction

Interprocedural data flow analyses have wide applicability to compiler optimization, data dependence analysis, data-flow-based testing, interprocedural pointer aliasing, interprocedural def-use chains, interprocedural program slicing and parallelization. Current techniques assume there are no procedure calls through function pointers as in C or procedure parameters[1] as in FORTRAN. Presence of such calls in a program would make most interprocedural analyzers inapplicable.

The problem with calls through function pointers is that control flow at these calls is not known at compile-time and has to be resolved for any interprocedural analyses. The precision of these analyses depends to a great extent on the precision of the resolution. One approximation would be to assume each such call could invoke *any* of the procedures, which is the target of a function pointer *anywhere* in the program (for instance, in C, this means any of those procedures, whose names have been either passed as actual arguments for function pointer formal parameters, or used in assignments for function pointers, or returned by procedures). More precise resolution requires interprocedural analysis.

A related problem is how calls through function pointers should be handled by interprocedural analyses. One way would be to treat each of these calls as an arbitrary branch to *any* of the procedures that could be called. The problem with this approach is that unfeasible paths, which do not correspond to real program executions, may be considered by the analyses; therefore imprecision may be introduced.

*The research reported here was supported, in part, by funds from NSF grants CCR-92-08632 and CCR-90-23628.

[1] These parameters are like single level function pointers as formal parameters in C.

In this paper, we consider single level function pointers[2] in a language like C. First, we define paths in a graphical program representation, that are considered *statically executable* for interprocedural analyses in the presence of calls through function pointers. We believe this definition leads to a static approximation of possible program execution, when there are calls through function pointers. The problem of determining if a function pointer points to a procedure at a program point along such a path, is called the *function pointer may aliasing problem*. We then classify the problem as either *polynomial-time* or *NP-hard*, with respect to the presence or absence of various program constructs affecting function pointer aliasing, such as global function pointers, assignments for function pointers, calls through function pointers and procedures returning function pointers.

Our results show that in the presence of global function pointers, calls through function pointers and assignments for function pointers, the aliasing problem is NP-hard. In the absence of global function pointers, the following two program constructs are significant to the problem: more than one function pointer used in calls through function pointers and procedures returning function pointers. The problem is NP-hard with both constructs present, but becomes polynomial without either construct.

To our knowledge, this is the *first* attempt to determine the theoretical complexity of handling function pointers in static analysis. This work enables us to identify sources of difficulties in the function pointer aliasing analysis and sources of approximations in existing algorithms. Ultimately, these findings will help us to come up with practical aliasing algorithms for function pointers and to tackle the problem of interprocedural data flow analyses in the presence of calls through function pointers.

Related Work Ryder [15] presented the first algorithm for call graph construction in the presence of calls through procedure parameters in FORTRAN. Callahan et. al. [2] extended it to handle recursion. Both algorithms propagate *sets* of pairs, each made of a procedure parameter and a procedure name, on *incomplete* call graphs and construct the final, precise call graphs. Hall and Kennedy [8] simplified the algorithm by propagating *single pairs* instead of sets and thus introduced imprecision into the call graphs constructed. None of these considered assignments for procedure parameters because they are not allowed in FORTRAN. Lakhotia [10] gave a more general algorithm that handled assignments for what he called *procedure variables*. His algorithm propagated single pairs, each consisting of a procedure variable and a procedure name, on program dependence graphs; similar to that of Hall and Kennedy, it is not precise. Weihl[17] proved that the problem of determining possible values for procedure-valued variables without considering other aliasing is P-space hard.

A related area of research is the work done on analyzing aliasing induced by pointers other than function pointers [4, 9, 12, 13]. Since function pointers are just a special kind of pointers, these methods have been adapted to handle function pointers [1, 6, 7], but none of these analyses explicitly define their precision and none are precise according to our definition of the problem.

There has also been some work on using static analysis to resolve function invocations through variables for functional languages like Scheme [16] or to resolve dynamic method invocations for object-oriented languages like SELF and C^{++} [3, 14].

None of these related work has addressed the theoretical difficulty of static analysis for imperative languages with calls through procedure parameters or function pointers.

[2] We show in [18] that multiple level function pointers are more difficult to handle.

Overview This paper is organized as follows. Section 2 presents the program representation and Section 3 defines the problem. The classification of the problem is reported in Section 4. Finally, Section 5 summarizes our results.

2 Program Representation

We are assuming an imperative language such as C, in which pointers to functions[3], called *function pointers*, can be assigned, used to invoke functions that they point to, passed as actuals in procedure calls, and returned by procedures. It is assumed that functions or procedures can not be created dynamically; in other words, function pointers can only point to functions defined in a program at compile-time. For procedure calls, call-by-value parameter passing is employed.

In this paper, we only consider single level function pointers; that is, only one level of dereference is necessary to access procedures to which they point. In C, for example, the function pointer fp declared as $void\ (*fp)()$ is of single level, whereas arrays of function pointers or pointers to structures with function pointer fields need two levels of dereference to access corresponding procedures. There are three kinds of single level function pointers: global, local and formal parameter. The first kind can be used in any procedure and the last two kinds can only be used in the procedure, in which they are declared.

We allow type casting *between function pointers*[4] in assignments for function pointers, return statements and procedure calls. General casting between other pointers and function pointers, however, is prohibited. With this limited form of type casting, a function pointer can point to a procedure of any type signature and a procedure can return a function pointer of any type signature (e.g., return a function pointer that points to the procedure itself).

We represent a program by an *extended interprocedural control flow graph* (EICFG). The EICFG contains an entry and an exit node for each procedure, a call and a return node for each call to a known procedure or each call through a function pointer, and one node for any other statement. We use $entry_{main}$ to represent the entry node for procedure *main*. For a procedure call that returns a function pointer[5], there is a node in the EICFG, which represents both a call and an assignment that assigns the returned function pointer to a variable. A special variable ret_{proc} is used in each procedure *proc*, which returns a function pointer, to save its returned function pointer; each return statement of *proc* in the source program, becomes an assignment for ret_{proc} in the EICFG representation. The EICFG contains all the control flow edges for each procedure and additional edges for procedure calls. For a call to a known procedure, there is an edge from the call node to the entry node of that procedure and an edge from the exit node of that procedure to the corresponding return node. For a call through a function pointer, there are edges from the call node to the entry node of *any* procedure, whose name is either used in assignments for function pointers, or passed as an actual parameter, or returned by a procedure, and whose formal parameters match the actual arguments of the call in their number. There are edges from exit nodes of those procedures to the corresponding return node.

[3]The term *function* and *procedure* are used interchangeably in this paper.

[4]Procedure names can be considered as special function pointers.

[5]In C, a calling procedure may or may not use the function pointer returned by a procedure. By *a procedure call returning a function pointer*, we mean that the calling procedure does use the returned function pointer.

It is theoretically difficult to exclude from static analysis, paths in the EICFG to any call through a function pointer which does not point to any procedure. To handle these static paths, if any, we introduce a special procedure NIL. Each global function pointer is initialized to point to NIL at the entry node of procedure *main*; each local function pointer of a procedure is initially made pointing to NIL at the entry node of the procedure. At each call through a function pointer, there is an edge from the call node to the entry node of procedure NIL and an edge from the exit node of NIL to the corresponding return node. If a function pointer does not point to any procedure at a call site along a path, NIL is considered to be called. The procedure NIL returns a function pointer that points to NIL.

It is worth pointing out that although a limited form of type casting is allowed in source programs, no casting appears in the EICFG representation. This is possible because the type casting does not have any other effect on function pointer aliasing besides making the function pointers involved agree on a same type signature.

Figure 1 shows an example C program and its EICFG. The special procedure NIL is not explicitly shown in the figure.

3 Problem Definition

In this section, we first define paths in the EICFG that should be considered by static analysis in the presence of calls through function pointers, and then define the function pointer may aliasing problem.

Definition 3.1 A path $P = n_1...n_{i-1}n_i$ in the EICFG, where n_1 is the entry node of a procedure, is a *realizable path* if either there is no return node on P or for each return node n_l $(1 < l \leq i)$ on P, such that n_{l-1} is an exit node and n_j $(1 < j < l-1)$ is its corresponding entry node, the following are true:

- The subpath $n_j...n_{l-1}$ of P is a realizable path.
- n_{j-1} on P is the call node corresponding to the return node n_l.

Intuitively, a realizable path has matching call and return nodes. The above definition is same as that of a realizable path in [11, 12]. Realizable paths starting from $entry_{main}$ would be considered potentially executable for static analysis if there are no calls through function pointers. In the presence of these calls, however, realizability does not totally capture the notion of potentially executable; we define *statically executable paths* to correct this deficiency.

Definition 3.2 A path $P = entry_{main}v_1...v_{i-1}v_i$ in the EICFG is a *statically executable path* if

- it is a realizable path, and
- at each call on P invoked through a function pointer, that function pointer points to the procedure being called on P.

By definition, any realizable path is statically executable if it does not contain any call through a function pointer. Only statically executable paths should be considered for data flow analysis in the presence of calls through function pointers.

Now, we define the *function pointer may aliasing problem* for single level function pointers. With single level function pointers, a function pointer alias exists if a function pointer points to a

```
typedef void (*PF)();

PF g;

main ()
{
  PF f1, f2;
  if (_)
  {   f1 = a;
      f2 = (PF) c;  }
  else
  {   f1 = b;
      f2 = d;   }
  f1(f2);
}

void a(f)
PF f;
{
  g = f();
}

void b(f)
PF f;
{
  f();
}

PF c()
{
  return (PF) c;
}

void d()
{
  g = d;
}
```

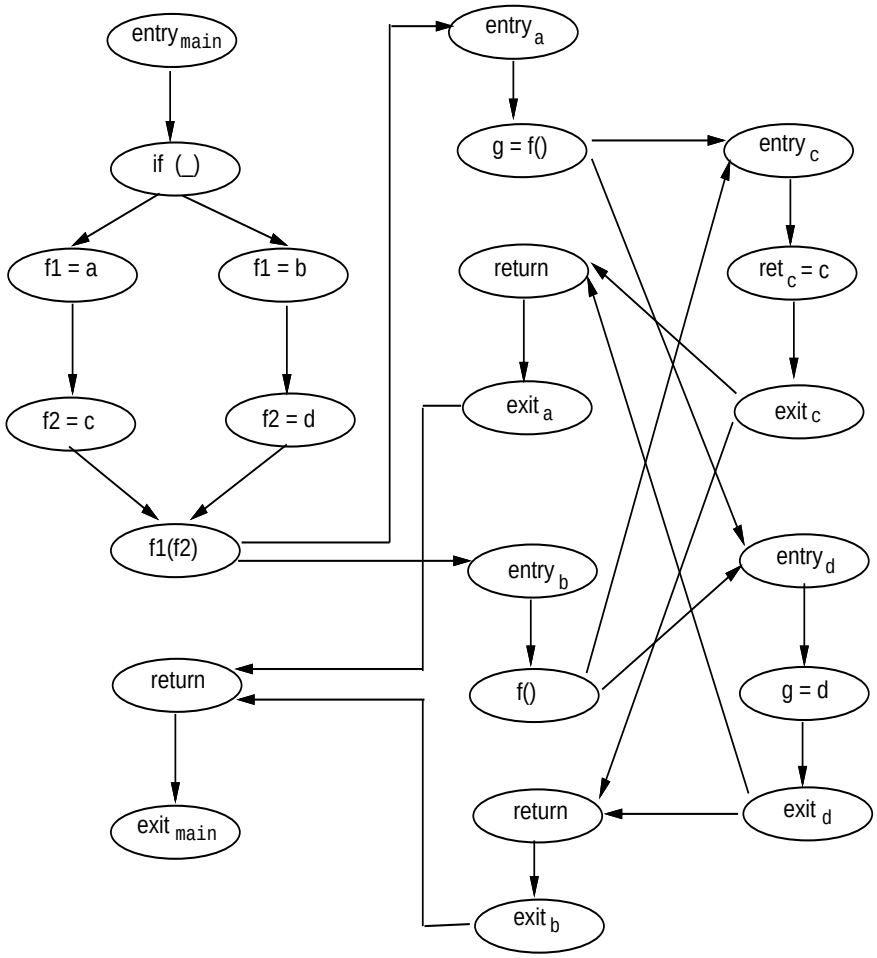

Figure 1: An Example Program and its EICFG

procedure, and thus can be represented by a pair consisting of the function pointer and the name of the procedure, e.g., $< fp , proc >$ [6], where fp is a function pointer and $proc$ is a procedure name.

Definition 3.3 In the presence of single level function pointers, the precise solution for the function pointer may aliasing problem is:

$$\left\{ [n, < fp , proc >] \;\middle|\; \begin{array}{l} \text{there is a statically executable path } entry_{main} v_1 ... v_i n \text{ in} \\ \text{the EICFG, such that the function pointer } fp \text{ points to} \\ \text{procedure } proc \text{ at the } top \text{ of } n \text{ on the path} \end{array} \right\}$$

Note that we associate aliasing information with the top of each node in the EICFG. In the rest of this paper, whenever we say a function pointer alias holds at a node, we mean it holds at the top of that node. We are solving an *interprocedural* problem if procedure calls are present; otherwise, we are solving an *intraprocedural* problem.

In our theoretical classification, we use the following assumption about reachability in the EICFG:

Assumption 3.1 Any statically executable path in the EICFG from $entry_{main}$ to the entry node of a procedure $proc$ can be extended into a statically executable path to any node in $proc$.

We say a function pointer fp is *referenced* in a call through a function pointer, if fp is the function pointer invoked in the call (e.g., $fp(...)$) or fp is an argument in the call (e.g., $fp_1(..., fp, ...)$).

Definition 3.4 For a given program, k denotes the maximum number of *distinct* function pointers referenced in any call through a function pointer in the program.

If there is a call through a function pointer in the program, then $k \geq 1$. For our classification, we use the following assumption about k:

Assumption 3.2 k can be treated as a constant, independent of program size [7].

In the rest of this paper, we will use *direct calls* to refer to calls of known procedures, *indirect calls* for calls through function pointers and *procedure calls* for both kinds of calls.

4 Classification of the Problem

We have studied the theoretical difficulty of solving the function pointer may aliasing problem for single level function pointers in the absence or presence of the following program constructs: global function pointers, assignments for function pointers, procedure calls, one or more function pointers referenced in indirect calls, and procedure calls returning function pointers. A summary of our results is provided in Figure 2.

[6] This notation is only used for single level function pointers. A more general representation would be $< *fp , *proc >$ [11, 12].

[7] This is analogous to similar assumption made about the maximum number of formal parameters of any procedure [5].

Program Constructs	Function Pointer May Aliasing in the absence of Single Level Global Function Pointers	Function Pointer May Aliasing in the presence of Single Level Global Function Pointers
• assignments for function pointers • no indirect calls • procedure calls returning function pointers	Polynomial	Polynomial
• assignments for function pointers • procedure calls	–	NP-hard [18]
• assignments for function pointers • procedure calls • one or more function pointers referenced in any direct call (i.e., $k \geq 1$) • no procedure calls returning function pointers	Polynomial [18]	NP-hard
• assignments for function pointers • procedure calls • only one function pointer referenced in any indirect call (i.e., $k = 1$) • procedure calls returning function pointers	Polynomial (Section 4.1)	NP-hard
• assignments for function pointers • procedure calls • more than one function pointer referenced in indirect calls (i.e., $k > 1$) • procedure calls returning function pointers	NP-hard (Section 4.2)	NP-hard

Figure 2: Classification of the Aliasing Problem for Single Level Function Pointers

With only direct calls, statically executable paths are simply realizable paths; the problem can be solved precisely by adapting the algorithm by Landi and Ryder [11, 12] for pointer-induced aliasing. With indirect calls and assignments for function pointers, the presence of global function pointers immediately makes tracking of statically executable paths NP-hard; the proof of this can be found in [18]. In the absence of global function pointers, the following two program constructs are significant to the problem: more than one function pointer referenced in indirect calls (i.e., $k > 1$) and procedure calls returning function pointers. The problem is NP-hard if both constructs are present; we prove this in Section 4.2. In the absence of either construct, the problem becomes polynomial. The explanation for the case with no more than one function pointer referenced in any indirect call (i.e., $k = 1$), is given in Section 4.1. The intuition behind the case with no procedure calls returning function pointers, is that only information about k or less function pointer aliases holding simultaneously at program points is necessary for determining precise function pointer may aliases; details are in [18]. The call graph construction problem for FORTRAN [2, 8, 15] is a special case of the latter, where no assignments for function pointers are allowed.

4.1 At Most One Function Pointer Referenced in any Indirect Call

In this section, we consider the function pointer may aliasing problem when assignments for function pointers, procedure calls, procedure calls returning function pointers are allowed and at most one function pointer can be referenced in any indirect call (i.e., $k = 1$). In this context, a procedure that does not return a function pointer, has no effect on function pointer aliasing at its call sites. A procedure that does return a function pointer, only affects aliases of the function pointer that is assigned its returned value in the calling procedure.

To account for the effects of procedures being called and avoid paths in the EICFG that are not realizable, a two-phase approach is employed, similar to [11, 12]. In the first phase, we solve the problem of *conditional function pointer may aliasing*; that is, we answer the question:

> If there is a statically executable path from $entry_{main}$ to the entry node of the procedure containing n, such that a set $\mathcal{A}$ of aliases for function pointer formal parameters of that procedure holds at the entry node, can the path be extended into a statically executable path to n, such that a function pointer fp points to procedure p at n on the extended path ?

In this phase, edges from call nodes to entry nodes of procedures are ignored and the effects of called procedures returning function pointers are incorporated at return nodes in calling procedures. In the second phase, we use the solutions for conditional function pointer may aliasing to solve for the actual function pointer may aliases. In this phase, edges from exit nodes to return nodes are ignored and the effects of procedure calls on function pointer aliasing are accounted for at entry nodes of called procedures.

Similar to [11, 12], we prove that in this context, we need only consider sets $\mathcal{A}$ of function pointer aliases such that $|\mathcal{A}| \leq 1$. The following lemma states this.

Lemma 4.1 In the presence of single level function pointers with no more than one function pointer referenced in any indirect call, and in the absence of global function pointers, if there is a statically executable path $P = entry_{main}...n_1n_2...n_in$ in the EICFG, where n_1 is the entry node of the procedure containing n, such that a function pointer alias $\mathcal{PF}$ holds at n on P, then one of the following is true:

- Any statically executable path from $entry_{main}$ to n_1 in the EICFG can be extended into a statically executable path to n, such that $\mathcal{PF}$ holds at n on the extended path.
- There is an alias $\mathcal{AF}$ for a function pointer formal parameter of the procedure, such that (i) $\mathcal{AF}$ holds at n_1 on P, and (ii) any statically executable path from $entry_{main}$ to n_1 in the EICFG, on which $\mathcal{AF}$ holds at n_1, can be extended into a statically executable path to n, such that $\mathcal{PF}$ holds at n on the extended path.

The proof of this Lemma is by induction on the length of the subpath $n_1 n_2 ... n_i n$ and a case analysis on n_i [18].

Computing Conditional Function Pointer May Aliases. Given an EICFG, conditional function pointer may aliases are computed on a *function pointer alias graph* (FPAG). Each node in the FPAG is of the form: [n, $\mathcal{AF}$, $\mathcal{PF}$], where n is an EICFG node, $\mathcal{AF}$ is an *assumed function pointer alias*, which is either an alias for a function pointer formal parameter or $\emptyset$, and $\mathcal{PF}$ is either a function pointer alias or a special alias γ. The alias γ is used to denote reachability information from $entry_{main}$ and is considered to hold at any node on a statically executable path from $entry_{main}$ to that node.

We use the *f-holds* relation to represent conditional aliasing information. *f-holds*[n, $\mathcal{AF}$, $\mathcal{PF}$] is *true iff* any statically executable path from $entry_{main}$ to the entry node of the procedure containing n, such that $\mathcal{AF}$ holds at the entry node[8], can be extended into a statically executable path to n, such that $\mathcal{PF}$ holds at n on the extended path. By this definition, *f-holds*[n, $\mathcal{AF}$, $\mathcal{AF}$] is *true* for any entry node n; *f-holds*[n, $\mathcal{AF}$, γ] is *true* for any node n in the EICFG because of Assumption 3.1.

Edges in the FPAG represent dependences between *f-holds* relation. Because of the two-phase approach, there are no interprocedural edges in the FPAG. Let n be a node in the EICFG, which is neither a call node nor an exit node. s is any successor node of n. We show below edges from [n, $\mathcal{AF}$, $\mathcal{PF}$] to other nodes in the FPAG by a case analysis of n:

- n is an entry node of a procedure.

  There is an edge from [n, $\mathcal{AF}$, $\mathcal{PF}$] to [s, $\mathcal{AF}$, $< fp , NIL >$] for any local function pointer fp of the procedure.

  If $\mathcal{PF}$ is not an alias of a local function pointer, there is an edge from [n, $\mathcal{AF}$, $\mathcal{PF}$] to [s, $\mathcal{AF}$, $\mathcal{PF}$] in the FPAG.

- n is an assignment node of the form: $fp = proc$, where fp is a function pointer and $proc$ is a procedure name.

  There is an edge from [n, $\mathcal{AF}$, $\mathcal{PF}$] to [s, $\mathcal{AF}$, $< fp , proc >$] in the FPAG.

  If $\mathcal{PF}$ is not an alias of fp, there is an edge from [n, $\mathcal{AF}$, $\mathcal{PF}$] to [s, $\mathcal{AF}$, $\mathcal{PF}$] in the FPAG.

- n is an assignment node of the form: $fp = fp_1$, where fp and fp_1 are function pointers.

  If $\mathcal{PF} = < fp_1 , p >$, where p is a procedure name, there is an edge from [n, $\mathcal{AF}$, $< fp_1 , p >$] to [s, $\mathcal{AF}$, $< fp , p >$] in the FPAG.

  If $\mathcal{PF}$ is not an alias of fp, there is an edge from [n, $\mathcal{AF}$, $\mathcal{PF}$] to [s, $\mathcal{AF}$, $\mathcal{PF}$] in the FPAG.

[8] $\emptyset$ means no alias is assumed; it is considered to hold at a procedure entry node on any statically executable path from $entry_{main}$ to that entry node.

- n is any other node.

  There is an edge from $[n, \mathcal{AF}, \mathcal{PF}]$ to $[s, \mathcal{AF}, \mathcal{PF}]$ in the FPAG.

By construction of the FPAG, *f-holds*$[n, \mathcal{AF}, \mathcal{PF}]$, where n is neither an entry node nor a return node, is *true* if *f-holds* relation for one of the predecessor nodes of $[n, \mathcal{AF}, \mathcal{PF}]$ in the FPAG is *true*. That is, *f-holds* is computed by the following equation:

$$\textit{f-holds}[n, \mathcal{AF}, \mathcal{PF}] \quad = \quad \bigvee_{\ll [pred, \mathcal{AF}, \mathcal{PF}'] \ , \ [n, \mathcal{AF}, \mathcal{PF}] \gg \in \mathcal{E}_{FPAG}} (\textit{f-holds}[pred, \mathcal{AF}, \mathcal{PF}'])$$

Computing *f-holds* for a return node, on the other hand, is a bit complicated; it depends on whether or not the procedure being called has any effect on function pointer aliasing in the calling procedure. Let *return* be a return node in the EICFG and *call* be its corresponding call node. Figure 4 shows the four possible cases, each of which will be discussed below. If *call* is a procedure call that does not return a function pointer, then a function pointer alias holds at *return if* it holds at *call* on any statically executable path; this justifies *Case 1* in Figure 4.

Now consider if *call* is a procedure call that returns a function pointer. Let fp be the function pointer that is assigned the returned function pointer, that is, *call* is of the form: $fp = proc(...)$ or $fp = fp_1(...)$. If $\mathcal{PF}$ is not an alias of fp, then $\mathcal{PF}$ holds at *return if* it holds at *call* on any statically executable path; this implies *Case 2* in Figure 4. Now suppose $\mathcal{PF}$ is an alias $< fp \ , \ q >$, where q is a procedure name, that holds at *return*. Let *proc* be the procedure being called at *call*. Then *proc* must return a function pointer that points to q. Since the effects of *proc* on its returned function pointer have been summarized by *f-holds* relation at its exit node $exit_{proc}$, e.g., *f-holds*$[exit_{proc}, \mathcal{AF}', < ret_{proc} \ , \ q >]$ for some assumed function pointer alias $\mathcal{AF}'$, we need only consider what alias has to hold at *call*, so that $\mathcal{AF}'$ is made true at the entry node of *proc* along a statically executable path. We use function *backbind* for this purpose.

$backbind_{\ll call \ , \ entry_{proc} \gg}(\mathcal{AF}')$, where $entry_{proc}$ is the entry node of *proc*, can be one of three values: *true*, *false* or a function pointer alias. Let P be *any* statically executable path in the EICFG of the form: $entry_{main} ... call \ entry_{proc}$. Informally, if $backbind_{\ll call \ , \ entry_{proc} \gg}(\mathcal{AF}')$ is *true*, then $\mathcal{AF}'$ holds at $entry_{proc}$ on P; if it is *false*, then $\mathcal{AF}'$ does not hold at $entry_{proc}$ on P; if it is a function pointer alias, then that alias holds at *call iff* $\mathcal{AF}'$ holds at $entry_{proc}$ on P. The definition of *backbind* is given in Figure 3.

Now assume $< fp \ , \ q >$ holds at *return* corresponding to a *call* of the form: $fp = proc(...)$ or $fp = fp_1(...)$ on a statically executable path $P = entry_{main} ... call \ entry_{proc} ... exit_{proc} \ return$. Consider the two cases of *call*:

- *call* is of the form: $fp = proc(...)$.

  Because $< fp \ , \ q >$ holds at *return* on P, $< ret_{proc} \ , \ q >$ must hold at $exit_{proc}$. By Lemma 4.1, there is an assumed function pointer alias $\mathcal{AF}'$ such that $\mathcal{AF}'$ holds at $entry_{proc}$ on P and *f-holds*$[exit_{proc}, \mathcal{AF}', < ret_{proc} \ , \ q >]$ is *true*.

  Since $\mathcal{AF}'$ holds at $entry_{proc}$ on P, $backbind_{\ll call \ , \ entry_{proc} \gg}(\mathcal{AF}')$ is either *true* or a function pointer alias. In the first case, any statically executable path from $entry_{main}$ to $entry_{proc}$ through *call* makes $\mathcal{AF}'$ hold at $entry_{proc}$. In the second case, the function pointer alias must hold at *call* so that $\mathcal{AF}'$ holds at $entry_{proc}$ on P; by Lemma 4.1, there is an

$backbind_{\ll call\ ,\ entry_{proc} \gg}(\emptyset) = true$

$$backbind_{\ll call\ ,\ entry_{proc} \gg}(< f\ ,\ p >^{\dagger}) = \begin{cases} true & \text{the actual for formal } f \text{ at } call \text{ is procedure name } p \\ false & \text{the actual for formal } f \text{ at } call \text{ is a procedure name other than } p \\ < f_1\ ,\ p > & call \text{ is a direct call and the actual for } f \text{ at } call \text{ is a function pointer } f_1 \\ true & call \text{ is a call through } fp_1, \text{ the actual for } f \text{ at } call \text{ is } fp_1^{\ddagger} \text{ and } p = proc \\ false & call \text{ is a call through } fp_1, \text{ the actual for } f \text{ at } call \text{ is } fp_1 \text{ and } p \neq proc \end{cases}$$

† f is a function pointer formal parameter of $proc$ and p is a procedure name.
‡ fp_1 is the only function pointer that can be referenced in $call$ if it is a call through fp_1.

Figure 3: Definition of *backbind* function

assumed function pointer $\mathcal{AF}$ such that *f-holds*[*call*, $\mathcal{AF}$, $backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}')$] is *true*. *f-holds* is computed as in *Case 3* of Figure 4.

- *call* is of the form: $fp = fp_1(...)$.

  To make the call to *proc*, $< fp_1\ ,\ proc >$ must hold at *call* on P. By Lemma 4.1, there is an assumed function pointer alias $\mathcal{AF}$ such that *f-holds*[*call*, $\mathcal{AF}$, $< fp_1\ ,\ proc >$] is *true*.

  Because $< fp\ ,\ q >$ holds at *return* on P, $< ret_{proc}\ ,\ q >$ must hold at $exit_{proc}$. By Lemma 4.1, there is an assumed function pointer alias $\mathcal{AF}'$ such that $\mathcal{AF}'$ holds at $entry_{proc}$ on P and *f-holds*[$exit_{proc}$, $\mathcal{AF}'$, $< ret_{proc}\ ,\ q >$] is *true*.

  Since $\mathcal{AF}'$ holds at $entry_{proc}$ on P, $backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}')$ can not be $false$. Because *call* is an indirect call, by definition of *backbind* , $backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}')$ must be *true*, which means $\mathcal{AF}'$ holds at $entry_{proc}$ on any statically executable path from $entry_{main}$ to $entry_{proc}$ through *call*. *f-holds* is computed as in *Case 4* of Figure 4.

The *f-holds* relation is initially $false$ for any node in the FPAG and is computed by a fixed point calculation of the equations defined above. The calculation takes time polynomial in the program size, because the number of *f-holds* is polynomial in the size of the EICFG, the number of function pointers and the number of procedure names in the program, and each *f-holds* relation will change its value at most once (from $false$ to $true$).

- *Case 1:* *call* is a procedure call that does not return a function pointer.
  $f\text{-}holds[return, \mathcal{AF}, \mathcal{PF}] = f\text{-}holds[call, \mathcal{AF}, \mathcal{PF}]$
- *Case 2:* *call* is of the form: $fp = proc(...)$ or $fp = fp_1(...)$ and $\mathcal{PF}$ is not an alias of fp.
  $f\text{-}holds[return, \mathcal{AF}, \mathcal{PF}] = f\text{-}holds[call, \mathcal{AF}, \mathcal{PF}]$
- *Case 3:* *call* is of the form: $fp = proc(...)$ and $\mathcal{PF} = < fp\ ,\ q >$, where q is a procedure name.
  $f\text{-}holds[return, \mathcal{AF}, < fp\ ,\ q >] =$

$$\bigvee_{\text{any } \mathcal{AF}'} \left\{ \begin{array}{l} f\text{-}holds[exit_{proc}, \mathcal{AF}',\ < ret_{proc}\ ,\ q >\ ] \wedge \\ \left(\begin{array}{l} backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}') = true \vee \\ f\text{-}holds[call, \mathcal{AF}, backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}')\ ] \end{array} \right) \end{array} \right\}$$

- *Case 4:* *call* is of the form: $fp = fp_1(...)$ and $\mathcal{PF} = < fp\ ,\ q >$, where q is a procedure name.
  $f\text{-}holds[return, \mathcal{AF}, < fp\ ,\ q >] =$

$$\bigvee_{\text{any } proc,\ \text{any } \mathcal{AF}'} \left\{ \begin{array}{l} f\text{-}holds[call, \mathcal{AF}, < fp_1\ ,\ proc >] \wedge \\ f\text{-}holds[exit_{proc}, \mathcal{AF}',\ < ret_{proc}\ ,\ q >\ ] \wedge \\ backbind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{AF}') = true \end{array} \right\}$$

Figure 4: Computing *f-holds* relation for return nodes

Computing Function Pointer May Aliases. Actual function pointer may aliases are then computed by using the conditional aliasing information. To do this, we need to know which aliases hold at an entry node given that some aliases hold at a call node. We use function *bind* to model the effects of parameter bindings at a procedure call on a statically executable path.

Let *call* be a call node in the EICFG. *proc* is a procedure that could be called at *call* and $entry_{proc}$ is its entry node. Intuitively, if $\mathcal{PF}$-set is a set of function pointer aliases, each of which holds at *call* on some statically executable path from $entry_{main}$ to *call*, then each alias in $bind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{PF}\text{-set})$ holds at $entry_{proc}$ on a statically executable path from $entry_{main}$ to $entry_{proc}$ through *call*. The *bind* function can be defined by using *backbind* , as shown in Figure 5. Note that the special alias γ indicates that *proc* can be reached from $entry_{main}$ along a statically executable path.

For any node n in the EICFG, we use $f\text{-}alias(n)$ to denote the set of function pointer aliases that hold at n on statically executable paths from $entry_{main}$ to n. $f\text{-}alias(n)$ is initialized to be $\emptyset$ for any node n and is computed by a fixed point calculation of the following equations:

- $f\text{-}alias(entry_{main}) = \{\ \gamma\ \}$
- n is an entry node of a procedure other than *main*.

$$f\text{-}alias(n) = \bigvee_{\ll c\ ,\ n \gg\ \in\ \mathcal{E}_{EICFG}} (\ bind_{\ll c\ ,\ n \gg}(f\text{-}alias(c)))$$

- n is any other node.

- *Case 1:* *call* is a direct call to *proc* and $\mathcal{PF}$-set $= \emptyset$.
  $bind_{\ll call\ ,\ entry_{proc} \gg}(\emptyset) = \emptyset$
- *Case 2:* *call* is a direct call to *proc* and $\mathcal{PF}$-set $\neq \emptyset$.
  $bind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{PF}\text{-set}) =$

$$\left(\begin{array}{l} \{\ \gamma\ \} \cup \{\ < f\ ,\ p > \mid backbind_{\ll call\ ,\ entry_{proc} \gg}(< f\ ,\ p >) = true\ \} \cup \\ \\ \{\ < f\ ,\ p > \mid backbind_{\ll call\ ,\ entry_{proc} \gg}(< f\ ,\ p >) = < f_1\ ,\ p > \wedge < f_1\ ,\ p > \in \mathcal{PF}\text{-set}\ \} \end{array} \right)$$

- *Case 3:* *call* is a call through function pointer fp_1 and $< fp_1\ ,\ proc > \notin \mathcal{PF}$-set.
  $bind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{PF}\text{-set}) = \emptyset$
- *Case 4:* *call* is a call through function pointer fp_1 and $< fp_1\ ,\ proc > \in \mathcal{PF}$-set.
  $bind_{\ll call\ ,\ entry_{proc} \gg}(\mathcal{PF}\text{-set}) =$

$$\left(\{\ \gamma\ \} \cup \{\ < f\ ,\ p > \mid backbind_{\ll call\ ,\ entry_{proc} \gg}(< f\ ,\ p >) = true\ \} \right)$$

Figure 5: Definition of *bind* function

Let $entry_{proc}$ be the entry node of procedure *proc* containing n.

$$\textit{f-alias}(n) = \left\{ \mathcal{PF} \;\middle|\; \begin{array}{l} \left(\gamma \in \textit{f-alias}(entry_{proc}) \wedge \textit{f-holds}[n, \emptyset\ ,\ \mathcal{PF}] = true \right) \vee \\ \\ \left(\exists\ \mathcal{AF} \in \textit{f-alias}(entry_{proc}) \text{ such that } \textit{f-holds}[n, \mathcal{AF}, \mathcal{PF}] = true \right) \end{array} \right\}$$

With solutions for *f-holds* relation available, the fixed point calculation of *f-alias* takes time polynomial in the program size, because there is one *f-alias* set for each EICFG node and each set can change its value at most $\mathcal{O}(V * P)$ times, where V is the number of function pointers and P is the number of procedure names in the program.

Theorem 4.1 In the absence of global function pointers, and in the presence of assignments for function pointers, procedure calls, procedures returning function pointers and no more than one function pointer referenced in any indirect call (i.e., $k = 1$), the problem of determining precise function pointer may aliases for single level function pointers is polynomial.

We claim the calculation of *f-holds* and *f-alias* is a polynomial time algorithm for determining precise function pointer may aliases in this context. The proof that the algorithm is precise is by induction on path length and number of iterations of the fixed point calculations for *f-holds* and *f-alias* [18].

4.2 More Than One Function Pointer Referenced in Indirect Calls and Procedure Calls Returning Function Pointers

Theorem 4.2 In the absence of global function pointers, and in the presence of assignments for function

```
typedef void (*PF)();

PF true( PF (*arg)() )
{ return ((PF) arg); }      /* type casting */

PF false( PF (*arg)() )
{ return ((PF) false); }      /* type casting */

main()
{
        PF  (*c)();
        PF  (*v1)(), (*v̄1)(),  ... , (*vm)(), (*v̄m)();
  L1:
        if (-)  { v1 = true; v̄1 = false; }  else  { v1 = false; v̄1 = true; }
        if (-)  { v2 = true; v̄2 = false; }  else  { v2 = false; v̄2 = true; }
        ...

        if (-)  { vm = true; v̄m = false; }  else  { vm = false; v̄m = true; }

  L2:
        if (-)  c = l1,1†   else  if (-)  c = l1,2;  else  c = l1,3;

        if (-)  ((PF) c) = (c)(l2,1);
           else  if (-)  ((PF) c) = (c)(l2,2);
              else  ((PF) c) = (c)(l2,3);        /* type casting */

        if (-)  ((PF) c) = (c)(l3,1);
           else  if (-)  ((PF) c) = (c)(l3,2);
              else  ((PF) c) = (c)(l3,3);
        ...

        if (-)  ((PF) c) = (c)(ln,1);
           else  if (-)  ((PF) c) = (c)(ln,2);
              else  ((PF) c) = (c)(ln,3);

  L3:   /* the formula satisfiable iff  < c , true > true at L3 */
}
```

† $l_{i,j}$ is the literal it represents in the formula (i.e., v_l or $\overline{v_l}$ for some l).

Figure 6: Reduction of the 3-SAT Problem to the Function Pointer May Aliasing Problem

pointers, procedure calls, more than one function pointer referenced in indirect calls (i.e., $k > 1$), and procedure calls returning function pointers, the problem of determining precise function pointer may aliases for single level function pointers is NP-hard.

We prove this theorem by reducing the 3-SAT problem for the formula $\bigwedge_{i=1}^{n}(l_{i,1} \vee l_{i,2} \vee l_{i,3})$ with propositional variables $\{v_1, v_2, ..., v_m\}$ to the problem of determining function pointer may aliases for the program shown in Figure 6. The program is written in C; it has more than one function pointer referenced in indirect calls, procedure calls returning function pointers, and type casting between function pointers. Its size is polynomial in the size of the formula.

We interpret the function pointer v_l $(1 \leq l \leq m)$ in the program pointing to procedure *true* or *false* as the propositional variable v_l in the formula being true or false respectively. Thus any path from L_1 to L_2 in the program corresponds to a truth assignment for the propositional variables and vice versa. If there is a truth assignment satisfying the formula, then for all $1 \leq i \leq n$, one of the literals $l_{i,1}$, $l_{i,2}$, $l_{i,3}$ points to *true* at L_2. Therefore, there is a path from L_2 to L_3, on which c is initially pointing to *true* and all subsequent indirect calls invoke procedure *true* with an argument pointing to *true*. The alias $< c$, *true* $>$ holds at L_3. On the other hand, if the formula is not satisfiable, for any path from L_1 to L_2 in the program, there exists i $(1 \leq i \leq n)$ such that the three literals $l_{i,1}$, $l_{i,2}$, $l_{i,3}$ all point to *false* at L_2. Therefore, on any path from L_2 to L_3, either an assignment for c or an indirect call to procedure *false* makes the function pointer c point to *false* thereafter. The alias $< c$, *true* $>$ does not hold at L_3.

In summary, the formula is satisfiable *iff* [L_3 , $< c$, *true* $>$] is in the function pointer aliasing solution for the program.

5 Conclusions

We have categorized the theoretical complexity of the function pointer may aliasing problem for single level function pointers with respect to various program constructs. An explanation of a polynomial time case and a proof of a NP-hard case are given. To our knowledge, this is the first theoretical classification of the difficulty of the problem. Future work includes design of a practical algorithm to handle function pointers and study of interprocedural analyses in the presence of indirect calls.

Acknowledgments We thank Bill Landi and Tom Marlowe for helpful discussions on this work. We also want to thank Hemant Pande for his help with the proof in Section 4.2 and the presentation of this paper.

References

[1] Rita Altucher. Personal communication. Feb. 1994.

[2] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transaction on Software Engineering*, 16(4):483–487, April 1990.

[3] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[4] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of te 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[5] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Technical Report 87-61, Department of Computer Science, Rice University, Houston,Texas, 1987.

[6] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[7] Rakesh Ghiya. Interprocedural analysis in the presence of function pointers. ACAPS Technical Memo 62, McGill University, School of Computer Science, December 1992.

[8] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, 1993.

[9] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Department of Computer Science, Cornell University, April 1990.

[10] Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of te 20th ACM Symposium on Principles of Programming Languages*, pages 273–284, January 1993.

[11] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*, pages 93–103, Jan. 1991.

[12] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM Symposium on Programming Language Design and Implementation*, June 1992.

[13] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: a clarification. *ACM SIGPLAN Notices*, 28(9):67–70, 1993.

[14] Hemant D. Pande and Barbara G. Ryder. Static type determination for C++. In *Proceedings of USENIX 6th C++ Technical Conference*, pages 85–97, April 1994. Also available as Technical Report 197-A, LCSR, Rutgers University.

[15] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transaction on Software Engineering*, 5(3):216–226, May 1979.

[16] Olin Shivers. Control flow analysis in scheme. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[17] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the 7th ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[18] Sean Zhang and Barbara G. Ryder. Complexity of interprocedural function pointer aliasing analysis. Technical report, Laboratory for Computer Science Research, Rutgers University, July 1994. In Preparation.