Level by Level: Making Flow- and Context-Sensitive Pointer Analysis Scalable for Millions of Lines of Code

Hongtao Yu [*] [$] Jingling Xue [†] Wei Huo [*] [$] Xiaobing Feng [*] Zhaoqing Zhang [*]

[*]Institute of Computing Technology
Chinese Academy of Sciences, China
{htyu, huowei, fxb, zqzhang}@ict.ac.cn

[$]Graduate School
Chinese Academy of Sciences, China

[†]School of Computer Science and Engineering
University of New South Wales, Australia
jingling@cse.unsw.edu.au

Abstract

We present a practical and scalable method for flow- and context-sensitive (FSCS) pointer analysis for C programs. Our method analyzes the pointers in a program level by level in terms of their *points-to levels*, allowing the points-to relations of the pointers at a particular level to be discovered based on the points-to relations of the pointers at this level and higher levels. This level-by-level strategy can enhance the scalability of the FSCS pointer analysis in two fundamental ways, by enabling (1) fast and accurate flow-sensitive analysis on full sparse SSA form using a flow-insensitive algorithm and (2) fast and accurate context-sensitive analysis using a full transfer function and a meet function for each procedure. Our level-by-level algorithm, LevPA, gives rises to (1) a precise and compact SSA representation for subsequent program analysis and optimization tasks and (2) a flow- and context-sensitive MAY / MUST mod (modification) set and read set for each procedure. Our preliminary results show that LevPA can analyze some programs with over a million lines of C code in minutes, faster than the state-of-the-art FSCS methods.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Languages, Performance

Keywords Pointer Analysis, Alias Analysis

1. Introduction

Pointer analysis is the basis of most other static program analyses and many compiler optimizations, especially for C programs. The motivation to enhance the scalability of flow- and context-sensitive (FSCS) pointer analysis is that the increased precision thus obtained is crucial for many security-related program analysis and verification tasks and may also open up new opportunities for future compiler optimizations in the multi-core era. However, none of the existing FSCS pointer analysis algorithms [10, 13, 14, 17, 20, 25, 28] can analyze beyond a million of lines of code efficiently. In this paper, we introduce a FSCS (and also field-sensitive) pointer analysis that can analyze some of these programs in minutes.

Flow-sensitive pointer analysis provides the points-to information in the form of points-to sets [1] or graphs [3], which is often used by def-use analysis. Conversely, def-use analysis provides the def/use information useful for flow-sensitive pointer analysis. For example, assignments to a pointer through dereferences of other pointers can alter the points-to information of that pointer. To facilitate def-use analysis, the static single assignment (SSA) form [8] is widely used. As a result, many flow-sensitive analysis problems can be sped up on SSA by using flow-insensitive algorithms.

In this paper, we present a FSCS pointer analysis, LevPA, that analyzes the pointers in a C program level by level according to their points-to levels. This level-by-level strategy may significantly enhance the scalability of the FSCS pointer analysis. On one hand, the time and space efficiency of flow-sensitive pointer analysis can be improved by applying a flow-insensitive algorithm on full-sparse SSA. On the other hand, the precise and compact SSA form enables precise alias relations to be exploited to eliminate spurious def/use points, sharpening the precision of def-use analysis.

The feasibility of binding full-sparse SSA and pointer analysis together comes from a simple observation. If all the def (definition) and use sites (either direct or indirect) of a pointer variable have been determined, we can build the SSA form for all accesses to the pointer and then analyze them flow-sensitively (by using a flow-insensitive algorithm). A variable can be referenced directly or indirectly through dereferences of another pointer. If we want to know all the indirect references of a variable, we need to analyze all the pointers possibly pointing to it first. To guarantee this, we assign a property, *points-to level*, to each variable and analyze the variables in a program in decreasing order of their levels.

Hardekopf and Lin [11] have recently proposed a flow-sensitive pointer analysis on SSA. However, their method is semi-sparse in the sense that it only builds the SSA form for the *top-level* pointer variables, i.e., those that cannot be referenced indirectly, in a program. In contrast, LevPA is fully sparse. By going beyond just the *top-level* pointer variables, LevPA reaps the full benefits of performing pointer analysis on SSA for all pointers at all levels.

Context-sensitive pointer analysis also benefits from a level-by-level approach. The key in achieving context-sensitivity is to obtain the output of a procedure according to a given input. Some methods accomplish this by distinguishing different calling paths for a procedure, and consequently, have to analyze the entire program on an exploded call graph [24, 28]. Instead, LevPA builds a full transfer function for a procedure and applies it in all its calling contexts.

A transfer function for a procedure is a description of the relations between its *formal-out* parameters and its *formal-in* parameters. The *formal-in* parameters of a procedure $proc$ include not only the declared formal parameters but also the global variables, escaped local variables (into this procedure) and dynamic allocated objects whose values may be accessed by $proc$ or the procedures that $proc$ invokes. Similarly, the *formal-out* parameters of a procedure $proc$ include not only its return value but also the global variables, escaped local variables (from this procedure) and dynamic allocated objects whose values may be modified by $proc$ or the procedures that $proc$ invokes.

If the higher-level formal-in parameters of a procedure have been analyzed, the pointers of a particular level in the procedure body can be analyzed succinctly. Therefore, we make use of the points-to sets of formal-in parameters at higher levels (and possibly at the same level due to the presence of points-to cycles) to distinguish the calling contexts for a procedure and encode them in its transfer function. Our transfer functions can precisely describe the modification side effects of procedures while keeping their space requirements and function application costs to a minimum. This is mostly achieved by representing for the first time the points-to relations with context-sensitive conditions using BDDs [2].

We do the same for the interprocedural read side effects for a procedure by using a so-called meet function to describe the read side effects of the procedure on all its formal-ins.

In summary, the paper contributes a fully flow- and context-sensitive, and also field-sensitive pointer analysis that

- is the first to carry out a level-by-level FSCS pointer analysis that is different from the summary-based algorithm in [14];
- performs a full-sparse flow-sensitive analysis on SSA flow-insensitively with significantly reduced time and space costs;
- performs a context-sensitive analysis efficiently with a precise full transfer function and a meet function for each procedure;
- yields flow- and context-sensitive interprocedural MAY/MUST modification and read side effects on a compact SSA form; and
- analyzes million lines of code in minutes, faster than the state-of-the art FSCS pointer analysis algorithms.

The rest of the paper is organized as follows. Section 2 introduces the LevPA framework and describes how to compute the points-to levels of variables. Section 3 discusses the pointer analysis performed by LevPA for a particular level. Section 4 presents and discusses our experimental results. Section 5 introduces the related work, and finally, Section 6 concludes the paper.

2. The LevPA Framework

Our method can cover full C features. As in [1], it suffices to consider only four types of assignments: (1) $x = y$, (2) $x = \&y$, (3) $*x = y$, and (4) $x = *y$. Arrays are treated as monolithic scalar objects. Heap objects are modeled by representing an allocation site at a program point loc by a statement of the form $p = \&alloc_{loc}$. A memory deallocation statement for p is replaced by $p = NULL$. All memory operations on structs are flattened into memory operations on scalar fields. Function pointers are handled as in [5]. Pointer arithmetic operations are handled by assigning the union of the points-to sets of all pointer operands in a pointer-related assignment to the resulting pointer [22]. Type casting is handled by inferring the locations accessed by the pointer being cast.

Suppose we have computed a property, *points-to level*, for every abstract memory location, such as a scalar variable or a dynamic allocated object (with all such abstract locations being called variables henceforth). The points-to level of a variable v, denoted *ptl*(v), satisfies the two conditions stated below.

Condition 1. *If a variable x is possibly pointed to by a pointer y during an execution of the program, then* ptl$(x) \leqslant$ ptl(y).

Condition 2. *If a variable x is possibly modified by assigning the value of y* to *x during an execution of the program, through direct or indirect references to x and y, then* ptl$(x) \leqslant$ ptl(y). *(The modification can happen due to (1) $x = y$, (2) $*p = y$, where p may point to x or (3) $x = *q$, where q may point to y.)*

Algorithm 1 The LevPA pointer analysis.

```
 1: Compute the points-to levels for all variables;
 2: Build the procedure call graph, denoted $PCG$;
 3: Reduce $PCG$ to a SCC-DAG, denoted $AVG$;
 4: repeat
 5:     for $lev$ from highest to lowest do
 6:         Bottom-up_analysis($AVG$, $lev$)
 7:         Top-down_analysis($AVG$, $lev$)
 8:     end for
 9:     Update $PCG$ and $AVG$ due to resolved indirect calls;
10: until $PCG$ does not change;
```

Our LevPA framework is summarized in Algorithm 1. We start by computing the points-to levels for all variables in a program. We then build its procedure call graph (PCG), where no function pointer is resolved yet. To handle recursive calls, PCG is partitioned into strongly connected components (SCC) to form a directed acyclic graph (a SCC-DAG), denoted AVG. The procedures in the same SCC form a recursion cycle. We then start analyzing the pointers of the same level together from highest to lowest. When working at a level, only the points-to sets of the pointers at this level are computed. The points-to sets of higher-levels pointers are used while the pointers at the lower level are ignored. Conditions 1 and 2 guarantee that whenever a variable is analyzed, all the other variables that may have an impact on its value either have been analyzed earlier or are being analyzed at the same time. The analysis at each level typically entails traversing AVG twice, first bottom-up (by Algorithm 3) and then top-down (by Algorithm 6), but iterations (not shown in Algorithm 1 explicitly) may be required as described in the next paragraph. During the bottom-up phase, we construct the (extended) SSA form for the pointers of the current level and perform the pointer inference to compute the points-to set for every pointer at the current level that may be possibly expressed in terms of the points-to sets of some formal-ins. In addition, we also build a full transfer function (and also a meet function) for each procedure. During the top-down phase, we propagate the points-to sets of the formal-in pointers of the level being analyzed to their use sites, and at the same time, expand the dereferences for the pointers at the current level to prepare for the analysis of the pointers at the next level (and lower levels). In actuality, the pointer dereference expansion performed signals the beginning of the SSA form construction for the pointers at the next and lower levels but this step can be moved into the bottom-up phase at the level below.

The pointer analysis at a level may involve two types of iterations. First, on detecting some cyclic points-to relations during the pointer dereference expansion performed in the top-down phase for a level, the pointers at this level are analyzed iteratively until their points-to sets are fully resolved (as discussed in Section 3.1.6). Second, in the presence of recursion, iterations are required to analyze the pointers in the procedures in each recursion cycle in PCG. In addition, in the presence of function pointers, PCG is built incrementally, causing iterations to be performed in order to accommodate both the modification and read side effects introduced by the newly resolved procedure calls on the program being analyzed.

As shown in Algorithm 2, we compute the points-to level of a variable based on the points-to graph built by a Steensgaard-styled pointer analysis [21, 22, 26] (line 1), which runs in almost linear time. Steensgaard-styled pointer analysis is an equivalence-based analysis. If there is an assignment between two variables x and y, both must point to the same object in the underlying points-

Algorithm 2 Computing points-to levels.

```
1: Perform the Steensgaard-styled pointer analysis;
2: Add pair-wise points-to edges for all predecessors of a node in
   the points-to graph thus obtained;
3: Reduce the points-to graph to a SCC-DAG;
4: Set ptl($r$) $= 0$ for every leaf node $r$;
5: for each non-leaf node $r$ such that the points-to levels of all its
   successors have already been computed do
6:     Let $s_1, \dots, s_n$ be all the successors of $r$;
7:     Set ptl($r$) $= \max\{$ptl($s_1$)$, \dots,$ ptl($s_n$)$\} + 1$;
8: end for
```

```
int  obj, t;

main()
{
   L1:   int **x, **y;
   L2:   int *a, *b, *c, *d, *e;
   L3:   x = &a;     y = &b;
   L4:   foo(x, y);
   L5:   *b = 5;
   L6:   if (t)   { x = &c;   y = &e; }
   L7:   else     { x = &d;   y = &d; }
   L8:   c = &t;
   L9:   foo(x, y);
  L10:   *e = 10;
}

void  foo(int **p, int **q)
{
  L11:   int *tmp = *q;
  L12:   *p = tmp;
  L13:   tmp = &obj;
  L14:   *q = tmp;
}
```

Figure 1. A motivating example.

to graph. In line 2, we ensure conservatively that Condition 2 is always satisfied. Otherwise, if x and y point to the same object in the points-to graph when y participates in a cycle that does not contain x, then *ptl*(x) $>$ *ptl*(y) would be possible. In this case, an assignment like $x = y$ would be rendered unanalyzable. In lines 3 – 8, the points-to level of a variable is computed as the longest distance from the node containing the variable to a leaf node.

Theorem 1. *Conditions 1 and 2 are satisfied by the points-to levels of variables computed by Algorithm 2 for a program.*

Proof. If a variable x is possibly pointed to by a pointer y, there must be a points-to edge from the node representing y to the node representing x in the Steensgaard-styled points-to graph. After line 3, x and y may be in the same SCC. In any case, *ptl*(x) $\leqslant$ *ptl*(y) always holds. So Condition 1 is guaranteed to be satisfied. Due to line 2, if a variable x is possibly modified by assigning the value of y to x, then x and y must be in the same SCC. Again *ptl*(x) $\leqslant$ *ptl*(y) holds. Condition 2 is satisfied, too. □

Our motivating example is given in Figure 1. For convenience, all assignments have already been put into the form supported in the LevPA framework. By applying Algorithm 2, we find that the pointers are organized in three levels: *ptl*(x) = *ptl*(y) = *ptl*(p) = *ptl*(q) = 2, *ptl*(a) = *ptl*(b) = *ptl*(c) = *ptl*(d) = *ptl*(e) = *ptl*(tmp) = 1 and *ptl*(t) = *ptl*(obj) = 0. LevPA will compute the points-to sets first for the pointers in $\{x, y, p, q\}$, then the pointers in $\{a, b, c, d, e, tmp\}$, and finally, the pointers in $\{t, obj\}$.

Algorithm 3 Bottom-up analysis.

```
 1: procedure Bottom-up_analysis($AVG$, $lev$)
 2: begin
 3: for each node $scc$ in reverse topological order of $AVG$ do
 4:     for each procedure $proc$ of $scc$ do
 5:         (a) Create_$\mu$_$\chi$_for_callsites($proc$, $lev$);
 6:         (b) Build the extended SSA form;
 7:         (c) Perform pointer inference;
 8:     end for
 9: end for
10: end
```

3. Analyzing a Level

Analyzing a level amounts to computing the points-to sets for the pointers at this level. LevPA proceeds in two phases, first bottom-up and then top-down. Both phases are inter-related. The bottom-up phase determines the points-to set for a pointer at the level being analyzed possibly in terms of the points-to sets of some formal-ins. All these points-to sets will be fully resolved subsequently in the top-down phase by propagating the points-to sets of formals-in to their use sites (with the actual parameters of a procedure call being bound to their corresponding formal parameters).

In this section, we focus on discussing how to analyze the pointers of a specific level on the assumption that the pointers of higher levels have already been analyzed. We first look at the bottom-up phase and then the top-down phase.

3.1 Bottom-Up Analysis

Algorithm 3 gives the key steps performed for a given level, lev. In this phase, we process the nodes in AVG in reverse topological order. Step 3(a) collects the flow- and context-sensitive read and modification side effects of all call sites in a procedure in terms of the μ and χ operators, respectively [7]. The μ and χ operators for the dereferencing operations on the pointers of level lev+1 are introduced earlier during the top-down analysis for lev+1. Step 3(b) builds the extended SSA form [7, 8] for the pointers of level lev. Step 3(c) performs the pointer inference to compute the points-to set for every pointer of level lev, which may be expressed in terms of some formal-ins at lev or higher. In this last step, we also obtain a full transfer function for each procedure that encodes its flow- and context-sensitive MAY/MUST modification side effects for its formal-outs and a meet function that gives its flow- and context-sensitive read side effects for its formal-ins.

In Algorithm 3, a node scc may represent multiple procedures contained in a recursion cycle. As a result, the bottom-up analysis for each procedure may have to be done iteratively using a work list in a demand-driven fashion. Whenever the transfer or meet function of a procedure has changed, the procedure is inserted into the work list, causing more iterations for its callers and indirect callers until a fixed point has been reached (i.e., when the work list is empty).

Section 3.1.1 introduces the full transfer functions used for specifying the interprocedural MAY/MUST modification side effects of procedures. Section 3.1.2 introduces the meet functions used for specifying the interprocedural read side effects of procedures. Section 3.1.3 introduces the extended SSA form used in the LevPA framework. Sections 3.1.4 - 3.1.6 describe Steps 3(a) - 3(c) of Algorithm 3, respectively, and illustrate them by examples.

3.1.1 Full Transfer Functions

To obtain high scalability while maintaining context-sensitivity, we build a single full transfer function for a procedure that can describe the modification side effect of the procedure independently of its inputs. Wilson and Lam [25] use partial transfer functions (PTFs) in their flow- and context-sensitive pointer analysis. PTFs are built for a procedure according to different alias inputs. This may result in analyzing a procedure more than once. For example, in Figure 1, the call sites L4 and L9 provide different alias inputs to foo. The formal parameters p and q do not alias with each other at L4 but may both point to d at L9. They thus analyze foo to build two different PTFs. Our method may also analyze a procedure multiple times, once for each level. However, the analysis at one level does not overlap with the analysis of another level. Furthermore, their parameterized representations of PTFs may result in precision loss. In Figure 1, their method employs so-called extended parameters 1_p and 1_q to represent all the variables pointed to by p and q, respectively. As 1_p and 1_q are aliases at L9, foo is regarded as having the same side effect on 1_p and 1_q, leading to the spurious points-to relation that c may point to obj after L9. Perhaps the major advantage of PTFs is that they summarize the side effects of a procedure only for those aliases that may actually occur in the program. However, by proceeding level by level, LevPA also eliminates unrealized alias relations according to calling contexts. Once the full transfer function for a procedure is available, we can apply it to different call sites accurately and efficiently.

We do not use full calling paths to distinguish calling contexts of a procedure when working at a particular level lev. Instead, we use the points-to sets of formal-in parameters at higher levels (and possibly at lev due to the existence of points-to cycles) to distinguish the calling contexts for the pointer accesses to the pointers at level lev and encode them into the transfer function for the procedure. We have designed a new points-to representation that not only describes the objects pointed to but also under which conditions the objects are pointed to. These context conditions are used to distinguish the calling contexts of a procedure so that its transfer function can be used in any calling context.

Definition 1 (**Points-to Set**). *Given a variable p of level lev, its points-to set, $Ptr(p)$ is $\{\langle v, M\rangle \mid v$ is an abstract memory location and $M \in \{\text{may}, \text{must}\}\}$. For convenience, we write $p \Rightarrow v$ ($p \rightarrow v$) to highlight the fact that p must (may) point to v.*

Definition 2 (**Context Condition**). *When working at a level lev, a context condition $\mathbb{C}(c_1, \ldots, c_k)$ is a Boolean function such that c_i evaluates to true (false) if the points-to relation that it represents for a pointer at lev or a higher level evaluates to true (false).*

As one of the contributions in this work, context conditions are implemented using BDDs [2], thereby greatly reducing the costs for representing and applying transfer functions. With BDDs, we can not only compactly represent context conditions but also enable Boolean operations to be evaluated efficiently. For example, Figure 2 shows how the context condition $\mathbb{C} = (p \rightarrow a \wedge q \rightarrow a) \vee p \rightarrow b$ is represented by a BDD. Each variable node in the BDD represents a points-to relation. We allocate a unique id for each points-to relation by organizing all points-to relations of all levels in a vector. This vector is filled up incrementally during the level-by-level analysis. The unique id of a points-to relation is just the index of the vector. For example, if only the points-to relation $p \rightarrow b$ holds at a call site, we can evaluate the context condition by writing $\mathbb{C}|_{x1=0,x2=0,x3=1}$ to see whether $\mathbb{C}$ holds at the call site or not. (The formal parameters of a procedure in $\mathbb{C}$ will be mapped to their corresponding actual parameters at the call site.)

Due to the inter-phase dependency between the top-down and bottom-up phases conducted at a level lev, the points-to-set $Ptr(p)$ of a variable p may not be explicitly computed until only after both phases are finished. Specifically, $Ptr(p)$ can be deduced from the following two sets given in Definitions 3 and 4, respectively.

Definition 3 (**Local Points-to Set**). *Given a variable p of level lev, $Loc(p)$ yields a so-called points-to set that is computed explicitly during the bottom-up phase and that may be included in $Ptr(p)$. It is recorded as a map $\{\langle\, v, \mathbb{C}(c_1, \ldots, c_k)\,\rangle \mid v$ is an abstract memory location and $\mathbb{C}(c_1, \ldots, c_k)$ is a context condition$\}$, meaning that p may/must point to v if and only if $\mathbb{C}(c_1, \ldots, c_k)$ holds. (Whether p may or must point to v at a particular point depends on the objects possibly pointed to by p, given by $Loc(p)$ and $Dep(p)$, at the point.)*

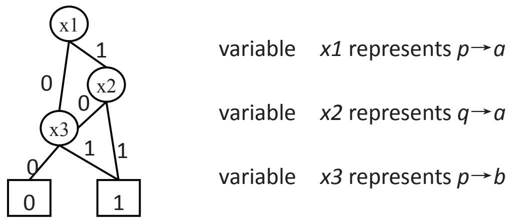
x1
x2
x3
0
1
0
0
1
1
0
0
1
variable x1 represents p→a
variable x2 represents q→a
variable x3 represents p→b

Figure 2. The BDD for $\mathbb{C} = (p \rightarrow a \wedge q \rightarrow a) \vee p \rightarrow b$.

For example, if `x = &y` is analyzed during the bottom-up phase at a level lev, then $\langle \texttt{y}, \textit{true}\rangle$ will be included in $Loc(x)$.

Definition 4 (**Dependence Set**). *Given a variable p of level lev, the dependence set $Dep(p)$ specifies the set of formal-ins f whose points-to sets may be included in $Ptr(p)$, i.e., are dependent on by $Ptr(p)$. It is is recorded as a map $\{\langle\, q, \mathbb{C}(c_1, \ldots, c_k)\,\rangle \mid q$ is a formal-in parameter of level lev and $\mathbb{C}(c_1, \ldots, c_k)$ is a context condition$\}$, meaning that for every $(q, \mathbb{C}(c_1, \ldots, c_k)) \in Dep(p)$, $Ptr(p)$ includes $Ptr(q)$ if and only if $\mathbb{C}(c_1, \ldots, c_k)$ holds.*

The dependence set $Dep(p)$ of a variable p at a level lev is used to record the data dependency between p and some formal-in parameters at the same level, since the points-to sets of p and the formal-ins will have to be determined together during the top-down phase. For example, a pointer may point to whatever a formal parameter points to when both are analyzed at the same level. The points-to set of the pointer can be determined as soon as the points-to set of the formal-in parameter is (propagated top-down).

The transfer function of a procedure is a combination of the transfer functions of all its formal-out parameters.

Definition 5 (**Transfer Function**). *Given a formal-out v at level lev of a procedure $proc$, its transfer function $Trans(proc, v)$ is a quadruple $\langle Loc(v), Dep(v), \mathbb{C}(c_1, \ldots, c_k), M\rangle$, where $\mathbb{C}(c_1, \ldots, c_k)$ is a context condition and $M \in \{\text{may}, \text{must}\}$, meaning that v may (must) be modified at a call site invoking $proc$ if $M =$ "may" ($M =$ "must") provided that $\mathbb{C}(c_1, \ldots, c_k)$ (with all formal parameters of $proc$ being mapped to their actual parameters) holds at the call site. The transfer function $Trans(proc, lev)$ of $proc$ at level lev is a combination of the individual transfer functions $Trans(proc, v)$ for all formal-out parameters v at lev.*

Let us understand the transfer functions thus defined using the example given in Figure 1. We start with the bottom-up analysis at level 2 first. The procedure foo does not modify any variable of level 2. So its transfer function at level 2 is empty:

$$Trans(foo, p) \;=\; Trans(foo, q) \;=\; \{\} \tag{1}$$

The procedure $main$ modifies x and y, but these two variables are local. So its transfer function at level 2 is also empty:

$$Trans(main, x) \;=\; Trans(main, y) \;=\; \{\} \tag{2}$$

A top-down analysis that follows immediately propagates the points-to sets of the actuals x and y of foo to their corresponding formal-ins p and q, respectively. In this case, LevPA finds that

$$\begin{array}{rcl} Ptr(p) & = & \{\langle a, \text{must}\rangle, \langle c, \text{may}\rangle, \langle d, \text{may}\rangle\} \\ Ptr(q) & = & \{\langle b, \text{must}\rangle, \langle d, \text{may}\rangle, \langle e, \text{may}\rangle\} \end{array} \tag{3}$$

Next, we start the bottom-up analysis for a, b, c, d, e and tmp at level 1, where the first five variables are formal-outs (and also formal-ins) of foo. The transfer function $\mathit{Trans}(foo, 1)$ is thus a combination of the following five individual transfer functions:

$$\begin{array}{ll}\mathit{Trans}(foo, a) = \langle \{\}, \{\langle b, q \Rightarrow b\rangle, \langle d, q \rightarrow d\rangle, \langle e, q \rightarrow e\}, & \\ \qquad p \Rightarrow a, \mathit{must}\rangle & \\ \mathit{Trans}(foo, c) = \langle \{\}, \{\langle b, q \Rightarrow b\rangle, \langle d, q \rightarrow d\rangle, \langle e, q \rightarrow e\}, & \\ \qquad p \rightarrow c, \mathit{may}\rangle & \\ \mathit{Trans}(foo, b) = \langle \{\langle obj, q \Rightarrow b\rangle\}, \{\}, q \Rightarrow b, \mathit{must}\rangle & (4) \\ \mathit{Trans}(foo, e) = \langle \{\langle obj, q \rightarrow e\rangle\}, \{\langle e, q \rightarrow e\rangle\}, q \rightarrow e, \mathit{may}\rangle & \\ \mathit{Trans}(foo, d) = \langle \{\langle obj, q \rightarrow d\rangle\}, \{\langle b, p \rightarrow d \wedge q \Rightarrow b\rangle, & \\ \qquad \langle d, p \rightarrow d\rangle, \langle e, p \rightarrow d \wedge q \rightarrow e\rangle\}, & \\ \qquad p \rightarrow d \vee q \rightarrow d, \mathit{may}\rangle & \end{array}$$

As can be observed from the transfer functions given above, $\mathit{Trans}(foo, a)$ and $\mathit{Trans}(foo, c)$ are structurally identical, and similarly for $\mathit{Trans}(foo, b)$ and $\mathit{Trans}(foo, e)$. When analyzing a procedure at a level lev during the top-down phase, LevPA tries to allocate a common parameterized space to a set of its formal-out parameters at the level below (i.e., $lev - 1$) to merge their transfer functions by merging the side effects on them. Thus, $Trans(foo, 1)$ is simplified to be a combination of the following three transfer functions:

$$\begin{array}{ll}\mathit{Trans}(foo, V^p) = \langle \{\}, \{\langle V^q, \mathit{true}\rangle\}, \mathit{true}, \mathit{must}\rangle & \\ \mathit{Trans}(foo, V^q) = \langle \{\langle obj, \mathit{true}\rangle\}, \{\}, \mathit{true}, \mathit{must}\rangle & (5) \\ \mathit{Trans}(foo, d) \;\; = \langle \{\langle obj, q \rightarrow d\rangle\}, \{\langle V^q, \mathit{true}\rangle, \langle d, p \rightarrow d\rangle\}, & \\ \qquad p \rightarrow d \vee q \rightarrow d, \mathit{may}\rangle & \end{array}$$

where the formal-out parameters a and c are parameterized by V^p, and b and e by V^q but d is not parameterized (during the top-down analysis at level 2). Unlike [25], the way we merge the side effects on formal-outs in a procedure by using a parameterized space (in Algorithm 7) never loses precision because the formal-ins parameterized together have exactly the same def/use points in the procedure except they may differ in their MAY/MUST modification effects. (This is why d is not parameterized as either part of V^p or V^q.) Such differences are distinguished at a calling context when the transfer function of the procedure is applied (lines 36 and 39 of Algorithm 4).

3.1.2 Meet Functions

We also need to define a meet function for a procedure that merges the inputs to the procedure at different calling contexts, specifying essentially its interprocedural read side effects.

Definition 6 (Meet Function). *Given a formal-in parameter v of level lev read (referenced) in a procedure $proc$, its meet function $\mathit{Meet}(proc,\ v)$ is a tuple $\langle Ptr(v), \mathbb{C}(c_1, \ldots, c_k)\rangle$, meaning that v (or the corresponding actual parameter of v if v is a formal parameter of $proc$) may/must be read at a call site invoking $proc$ only when $\mathbb{C}(c_1, \ldots, c_k)$ (with the formal parameters of $proc$ being mapped to their actual parameters) holds at the call site. The meet function $Meet(proc, lev)$ of $proc$ is a combination of all such individual meet functions $Meet(proc, v, lev)$ for all its formal-ins v at lev.*

The level-wise meet functions for our example are:

$$\begin{array}{ll}\mathit{Meet}(foo, p) \;\;\; = \langle \{\langle a, must\rangle, \langle c, may\rangle, \langle d, may\rangle\}, \mathit{true}\rangle & \\ \mathit{Meet}(foo, q) \;\;\; = \langle \{\langle b, must\rangle, \langle d, may\rangle, \langle e, may\rangle\}, \mathit{true}\rangle & (6) \\ \mathit{Meet}(foo, V^q) = \langle \{\}, \mathit{true}\rangle & \\ \mathit{Meet}(foo, d) \;\;\; = \langle \{\}, q \rightarrow d\rangle & \end{array}$$

In each meet function, the pointed-to objects (if any) are the read side effects. Note that V^p is not read (referenced) in foo.

3.1.3 Extended SSA Form

The SSA form on which our pointer analysis operates is an extended SSA form [7] that can effectively represent aliases and indirect memory operations in the SSA. We employ and further extend the μ and χ operators to precisely characterize aliasing effects.

In the extended SSA form [7], the μ and χ operators are introduced to specify the aliasing effects for indirect memory operations and call statements. A μ operator for an indirect memory operation is used to specify which variables may be read by the operation. In $\mu(v_i)$, μ takes as its operand the version i of v that may be read and produces no result. In our extension, we append a context condition $\mathbb{C}(c_1, \ldots, c_k)$ to μ to indicate the calling context that the variable can be read. Thus, a μ operator has the form $\mu(v_i, \mathbb{C}(c_1, \ldots, c_k))$.

A χ operator for an operation is used to model which variables the operation may modify. The operand of a χ operation is the last version of a variable and its result is the version after this potential definition. So χ links up the use-def edges through a may-definition. We add a context condition $\mathbb{C}(c_1, \ldots, c_k)$ to χ to model under which calling context the variable can be modified. We also add a MAY/MUST modification field $M \in \{\mathit{may}, \mathit{must}\}$ to χ to distinguish between the two types of modifications of a variable. So a χ operation has the form $v_{i+1} = \chi(v_i, \mathbb{C}(c_1, \ldots, c_k), M)$.

The μ and χ operators for the variables read and modified at a call site are created in Step 3(a) of Algorithm 3. The μ and χ operators for the variables read and modified by a pointer dereferencing operation is created during the top-down phase (in Algorithm 6).

Property 1 (Context Condition for a Meet Function). *The context condition of a meet function $\mathit{Meet}(proc, p)$ for a formal-in p of $proc$ at level lev is a disjunction of the context conditions of all its use sites, including all its μ statements.*

Property 2 (Context Condition for a Transfer Function). *The context condition of a transfer function $\mathit{Trans}(proc, p)$ for a formal-out p of $proc$ at level lev is a disjunction of the context conditions of all its def sites, including all its χ statements.*

3.1.4 Step 3(a): Create the μ and χ Lists for a Call site

For every call site in a procedure $proc$, the μ and χ lists are created at the call site for all variables of level lev that may be read (referenced) and modified, respectively, by all the procedures that may be invoked at the call site, as shown in Algorithm 4. In lines 5 – 22, all the variables read at a call site in $proc$ are appended to the μ list of the call site. In lines 23 – 46, all the variables modified at a call site in $proc$ are appended to the χ list of the call site.

For the sake of time and space efficiency (as discussed earlier in Section 3.1.1), some parameterized spaces may be created for lev during the top-down phase at the preceding level, i.e., at $lev + 1$ (as shown in Algorithms 6 and 7). Such parameterized spaces are handled by the if statements in lines 10 and 29. In line 6, we need to know the context condition of the meet function for each callee. This is available at this phase but the associated points-to set, which is not used here, is not known until after the top-down analysis at the same level is finished (Property 1). In line 24, the transfer function of each callee is known since it has just been built in the bottom-up phase for lev (Property 2). In lines 9 and 28, $\mathbb{C}''$ is simplified from $\mathbb{C}'(c_1, \ldots, c_k)$ to include only the points-to relations that hold at the entry of $proc$. It is then used to build the context conditions required for the μ and χ operators created. When creating a parameterized space for a set of formal-ins, their *may* and *must* fields are also "merged" due to the fact they are collectively represented by the MAY/MUST field of the parameterized space (lines 32 – 36 in Algorithm 6). As a result, in lines 31 – 35, the MAY/MUST field for a parameterized space is refined at a calling context. In lines 37 and 40, the meet operator $\sqcap$ on $\{\mathit{may}, \mathit{must}\}$

```
f()
{
    L1:  p1 = &a;
    L2:  g1();
             p2=χ(p1, true, may)
             ~~q2=χ(q1, true, may)~~
             r2=χ(r1, true, may)

             μ(r2, true)
    L3:  g2();
             ~~s2=χ(s1, true, may)~~

    L4:  g3();
             ~~s3=χ(s2, true, may)~~
             t2=χ(t1, true, may)

    L5:  g4();
             t3=χ(t2, true, must)
}
```

Figure 3. χ optimization (with redundant ones striken through).

has the standard meaning: *may* $\sqcap$ e $=$ e $\sqcap$ *may* $=$ *may*, for $e \in \{may, must\}$, and *must* $\sqcap$ *must* $=$ *must*.

However, creating many μ and χ variables this way at each call site can sometimes introduce many constraints to be resolved at the pointer inference stage performed in Step 3(c). Some of these variables at a call site may not directly impact the points-to relations of the caller if they are not accessed in any way in the body of the caller; they only serve to transfer the modification or read side effects upwards through the caller. In this case, we can directly deduce their modification and read side effects on these variables from the transfer and meet functions at each call site.

In our implementation, we only create explicitly χ operators at all call sites for a variable v in a procedure $proc$ if one of the following three conditions is satisfied:

W1. v may be read or modified by some non-call statement(s) in the body of $proc$, explicitly or implicitly;

W2. v may be modified at a call site and may also be read at another call site that may or may not be different; and

W3. v must be modified at a call site that must be called by $proc$.

In the case of μ operations, there are two conditions instead:

R1. v may be modified by some non-call statement(s) in the body of $proc$, explicitly or implicitly; and

R2. v may be modified at a call site and may also be read at another call site that may or may not be different.

This optimization looks simple but computationally significant. By eliminating redundant μ and χ operators this way, we have observed a ten-fold analysis time reduction in some benchmarks.

We use the example given in Figure 3 to illustrate the three conditions for the χ optimization. The χ for q can be removed since it is possibly modified by $g1$ but not anywhere else. However, due to Condition W1, the χ for p must be kept. Note that s is possibly modified by $g2$ and $g3$. However, the points-to set of s at the exit of f is the union of the points-to sets of s_2 and s_3 since neither of the two definitions can be killed. In this case, the χ operations are not created at L3 and L4. Instead, *Trans*(f, s) can be directly deduced from *Trans*$(g2, s)$ and *Trans*$(g3, s)$ without losing any precision. However, due to Condition W2, r is possibly modified by $g1$ and possibly read by $g2$. The χ for L2 and the μ for L3 must be created in order not to miss any points-to relations. Finally, t is definitely modified by $g4$. Due to Condition W3, we cannot merge the side

Algorithm 4 Creating μ and χ for the call sites in a procedure.

```
procedure Create_μ_χ_for_callsites($proc$, $lev$)
begin
for each $callsite$ of $proc$ do
    for each $callee$ of $callsite$ do
        for each formal-in $f$ of $callee$, where $ptl(f) = lev$ do
            Let $\mathbb{C}'(c_1, \ldots, c_k)$ be the context condition $\mathbb{C}(c_1, \ldots, c_k)$ of $Meet(callee, f)$ with all the formal parameters of $callee$ being replaced by their corresponding actual parameters at $callsite$;
            Let $c'_i$ be 1 if $c_i$ holds at $callsite$ and 0 otherwise;
            if $\mathbb{C}'(c'_1, \ldots, c'_k)$ evaluates to true then
                Let $\mathbb{C}''$ include all and only the points-to relations in $\mathbb{C}'(c_1, \ldots, c_k)$ that hold at the entry of $proc$;
                if $f$ is a parameterized space $V^p$ then
                    Map $p$ to an actual parameter $q$ at $callsite$;
                    for each $\langle v, \mathbb{C}_v \rangle$ of $Loc(q)$ do
                        Insert $\mu(v, \mathbb{C}'' \wedge \mathbb{C}_v)$ to $\mu$ list of $callsite$;
                    end for
                    for each $\langle w, \mathbb{C}_w \rangle$ of $Dep(q)$ do
                        Insert $\mu(V^w, \mathbb{C}'' \wedge \mathbb{C}_w)$ to $\mu$ list of $callsite$;
                    end for
                else if $f$ is not a formal parameter of $callee$ then
                    Insert $\mu(f, \mathbb{C}'')$ to $\mu$ list of $callsite$;
                end if
            end if
        end for
        for each formal-out $f$ of $callee$, where $ptl(f) = lev$ do
            Let the transfer function $Trans(callee, f)$ = $\langle Loc(f), Dep(f), \mathbb{C}(c_1, \ldots, c_k), M \rangle$ be given;
            Let $\mathbb{C}'(c_1, \ldots, c_k)$ be obtained from $\mathbb{C}(c_1, \ldots, c_k)$ with all the formal parameters of $callee$ being replaced by their corresponding actual parameters at $callsite$;
            Let $c'_i$ be 1 if $c_i$ holds at $callsite$ and 0 otherwise;
            if $\mathbb{C}'(c'_1, \ldots, c'_k)$ evaluates to true then
                Let $\mathbb{C}''$ include all and only the points-to relations in $\mathbb{C}'(c_1, \ldots, c_k)$ that hold at the entry of $proc$;
                if $f$ is a parameterized space $V^p$ then
                    Map $p$ to an actual parameter $q$ at $callsite$;
                    if $(|Loc(q)| == 1$ && $Dep(q) == \{\})$ || $(Loc(q) == \{\}$ && $|Dep(q)| == 1)$ then
                        $M'$ = "must";
                    else
                        $M'$ = "may";
                    end if
                    for each $\langle v, \mathbb{C}_v \rangle$ of $Loc(q)$ do
                        Insert $\chi(v, \mathbb{C}'' \wedge \mathbb{C}_v, M \sqcap M')$ to $\chi$ list of $callsite$;
                    end for
                    for each $\langle w, \mathbb{C}_w \rangle$ of $Dep(q)$ do
                        Insert $\chi(V^w, \mathbb{C}'' \wedge \mathbb{C}_w, M \sqcap M')$ to $\chi$ list of $callsite$;
                    end for
                else if $f$ is not the return vale of $callee$ then
                    Insert $\chi(f, \mathbb{C}'', M)$ to $\chi$ list of $callsite$;
                end if
            end if
        end for
    end for
end for
end
```

```
        μ(b1, true)
L4:   foo(x, y);
        a2=χ(a1, true, must)
        b2=χ(b1, true, must)

        μ(d1, true)
        μ(e1, true)
L9:   foo(x, y);
        c2=χ(c1, true, may)
        d2=χ(d1, true, may)
        e2=χ(e1, true, may)
```

Figure 4. $main$ with the μ and χ operations introduced for its two call sites at level 2. Other statements are not shown.

effects of t_2 and t_3 since t_3 must kill the definition of t_2. So the χ operations for t must be kept.

3.1.5 Step 3(b): Build the Extended SSA Form

In applying the SSA creation algorithm described in [8], the variable operands of μ and χ are treated as uses and the results of χ as additional assignments. The variables in the μ and χ operations are then renamed together with the rest of the program variables.

3.1.6 Step 3(c): Pointer Inference

After the extended SSA form has been created, we perform a flow-sensitive pointer analysis on SSA using a flow-insensitive algorithm. Each SSA variable is treated as an independent variable. The flow-insensitive algorithm used is set-constraint-based, like the Andersen-styled pointer analysis [1]. Therefore, there are two stages: constraint generation and constraint resolution.

Constraint Generation In this first stage, we set up a constraint system for the relevant statements including those with μ, χ and ϕ operators, as shown in Table 1. Rule Init does the initialization for a formal-in parameter, assuming that its first version in the SSA form is 0. Rules Base and Simple are self-explanatory. Rules Mu and Chi are applicable to pointer dereferencing operations. In particular, a Mu constraint is introduced for a read access while a Chi constraint for a write access. The operator $\supseteq_{\mathbb{C}}$ is the conditional set inclusion. Phi applies to a ϕ operation in the standard SSA form.

When encountering a call statement for a callee, the transfer function of the callee is applied by calling Algorithm 5 to generate the constraints required at the call site.

Consider the program given in Figure 1. Suppose that we have already analyzed level 2. We are now working on the pointers at level 1, a, b, c, d, e, and tmp, during the bottom-up phase. Suppose that we have just finished analyzing foo. Its transfer function *Trans*$(foo, 1)$ is a combination of the three individual transfer functions *Trans*(foo, V^p), *Trans*(foo, V^q) and *Trans*(foo, d) given earlier in (5). During the bottom-up analysis of $main$ at level 1, we have created the μ and χ lists for its two call sites, L4 and L9, as shown in Figure 4. To generate the constraints at the two call sites, *Trans*$(foo, 1)$ is applied at L4 and L9, respectively, by calling Algorithm 5. At L4, $a_2 \supseteq b_1$ and $b_2 \supseteq \{obj\}$ are generated. At L9, $c_2 \supseteq d_1$, $c_2 \supseteq e_1$, $c_2 \supseteq c_1$, $e_2 \supseteq \{obj\}$, $e_2 \supseteq e_1$, $d_2 \supseteq \{obj\}$, $d_2 \supseteq e_1$ and $d_2 \supseteq d_1$ are generated.

Constraint Resolution In this second stage, we obtain the points-to relations by computing the transitive closure of the constraint graph representing the constraints generated during the constraint generation stage. When propagating the value from a node to a successor, a guarded set union operation $\supseteq_{\mathbb{C}}$ is used. So $\supseteq$ is a special case of $\supseteq_{\mathbb{C}}$ with its context condition being *true*.

Some operations on local points-to sets are introduced below.

Algorithm 5 Applying a full transfer function at a call site.

```
 1: procedure Apply_FTF($callsite, lev$)
 2: begin
 3: for each $v_m = \chi(v_n, \mathbb{C}_v, M)$ generated for $callsite$ do
 4:   for each $callee$ of $callsite$ do
 5:     Map $v$ to a formal-out parameter $f$ of $callee$;
 6:     Let $Trans(callee, f) = \langle Loc(f), Dep(f), \mathbb{C}_f, M\rangle$;
 7:     for each $\langle p, \mathbb{C}(c_1,\dots,c_k)\rangle$ of $Loc(f)$ do
 8:       Let $\mathbb{C}'(c_1,\dots,c_k)$ be obtained from $\mathbb{C}(c_1,\dots,c_k)$
          with all the formal parameters of $callee$ being replaced
          by their corresponding actual parameters at $callsite$;
 9:       Let $c'_i$ be 1 if $c_i$ holds at $callsite$ and 0 otherwise;
10:       if $\mathbb{C}'(c'_1,\dots,c'_k)$ evaluates to true then
11:         Let $\mathbb{C}''$ include all and only the points-to relations
            in $\mathbb{C}'(c_1,\dots,c_k)$ that hold at the entry of the caller;
12:         Generate a constraint $v_m \supseteq_{\mathbb{C}''} \{p\}$
13:       end if
14:     end for
15:     for each $\langle q, \mathbb{C}(c_1,\dots,c_k)\rangle$ of $Dep(f)$ do
16:       Let $\mathbb{C}'(c_1,\dots,c_k)$ be obtained from $\mathbb{C}(c_1,\dots,c_k)$
          with all the formal parameters of $callee$ being replaced
          by their corresponding actual parameters at $callsite$;
17:       Let $c'_i$ be 1 if $c_i$ holds at $callsite$ and 0 otherwise;
18:       if $\mathbb{C}'(c'_1,\dots,c'_k)$ evaluates to true then
19:         Let $\mathbb{C}''$ include all and only the points-to relations
            in $\mathbb{C}'(c_1,\dots,c_k)$ that hold at the entry of the caller;
20:         Map $q$ to a list of actual parameters, $actuals$
21:         for each $a_i$ in $actuals$ do
22:           Generate a constraint $v_m \supseteq_{\mathbb{C}''} a_i$
23:         end for
24:       end if
25:     end for
26:   end for
27:   if M == "may" then
28:     Generate a constraint $v_m \supseteq_{\mathbb{C}_v} v_n$
29:   else
30:     Generate a constraint $v_m \supseteq_{\sim\mathbb{C}_v} v_n$
31:   end if
32: end for
33: end
```

Definition 7 (Union for Local Points-to Sets). *Loc*$(p) \cup$ *Loc*(q) *is a new points-to set that contains* $\langle v, \mathbb{C}\rangle$ *if* $\langle v, \mathbb{C}\rangle$ *is either contained in Loc*(p) *or Loc*(q) *exclusively or satisfies the property that if* $\langle v, \mathbb{C}_1\rangle \in$ *Loc*(p) *and* $\langle v, \mathbb{C}_2\rangle \in$ *Loc*(q)*, then* $\mathbb{C} = \mathbb{C}_1 \vee \mathbb{C}_2$.

Definition 8 (Guarded Assignments for Local Points-to Sets). *Loc*$(p) \times \mathbb{C} = \{\langle v, \mathbb{C} \wedge \mathbb{C}'\rangle \mid \langle v, \mathbb{C}'\rangle \in$ *Loc*$(p)\}$.

These operations on dependence sets are similarly defined. During the resolution process, a cycle in the constraint graph needs to be resolved iteratively until a fixed point has been reached.

Transfer and Meet Functions For the meet function of a formal-in v of a procedure foo, *Meet*$(foo, v) = \langle$*Ptr*$(v), \mathbb{C}\rangle$, *Ptr*(v) is fully resolved in lines 8 – 19 in Algorithm 6 (but its points-to relations are determined during the bottom-up phase) and $\mathbb{C}$ is computed based on Property 1.

For the transfer function of a formal-out v of a procedure foo, *Trans*$(foo, v) = \langle$*Loc*(v), *Dep*$(v), \mathbb{C}, M\rangle$, *Loc*$(v)$ and *Dep*(v) are computed during the bottom-up phase and $\mathbb{C}$ by Property 2. The M field is computed by solving a constraint propagation problem together with the pointer inference. Each SSA variable is associated with a property, named $mod \in \{$*may*, *must*$\}$, to indicate how the

Rule	Statement	Constraint(s)	Meaning
Init	$a_1 = \chi(a_0, \textit{true}, \textit{must})$ $\textit{ptl}(a_1) = lev$	$a_1 \supseteq a_0$	$\textit{Loc}(a_1) = \{\}$ $\textit{Dep}(a_1) = \{\langle a, \textit{true}\rangle\}$
Base	$a_i = \&b$ $\textit{ptl}(a_i) = lev$	$a_i \supseteq \{b\}$	$\textit{Loc}(a_i) = \{\langle b, \textit{true}\rangle\}$ $\textit{Dep}(a_i) = \{\}$
Simple	$a_i = b_j$ $\textit{ptl}(a_i) = lev$	$a_i \supseteq b_j$	$\textit{Loc}(a_i) = \textit{Loc}(b_j)$ $\textit{Dep}(a_i) = \textit{Dep}(b_j)$
Mu	$\mu(v_k, \mathbb{C}_v)$ $a_i = *b_j$ $\textit{ptl}(a_i) = lev$	$a_i \supseteq_{\mathbb{C}_v} v_k$	$\textit{Loc}(a_i) = \textit{Loc}(a_i) \cup \textit{Loc}(v_k) \times \mathbb{C}_v$ $\textit{Dep}(a_i) = \textit{Dep}(a_i) \cup \textit{Dep}(v_k) \times \mathbb{C}_v$
Chi	$*a_i = b_j$ $v_m = \chi(v_n, \mathbb{C}_v, M)$ $\textit{ptl}(v_m) = lev$	$v_m \supseteq_{\mathbb{C}_v} b_i$ if $M ==$ ”*may*” $v_m \supseteq_{\mathbb{C}_v} v_n$ else $v_m \supseteq_{\sim\mathbb{C}_v} v_n$	$\textit{Loc}(v_m) = \textit{Loc}(b_j) \times \mathbb{C}_v$ $\textit{Dep}(v_m) = \textit{Dep}(b_j) \times \mathbb{C}_v$ if $M ==$ ”*may*” $\textit{Loc}(v_m) = \textit{Loc}(v_m) \cup \textit{Loc}(v_n) \times \mathbb{C}_v$ $\textit{Dep}(v_m) = \textit{Dep}(v_m) \cup \textit{Dep}(v_n) \times \mathbb{C}_v$ else $\textit{Loc}(v_m) = \textit{Loc}(v_m) \cup \textit{Loc}(v_n) \times (\sim \mathbb{C}_v)$ $\textit{Dep}(v_m) = \textit{Dep}(v_m) \cup \textit{Dep}(v_n) \times (\sim \mathbb{C}_v)$
Phi	$a_i = \phi(a_j, a_k)$ $\textit{ptl}(a_i) = lev$	$a_i \supseteq a_j$ $a_i \supseteq a_k$	$\textit{Loc}(a_i) = \textit{Loc}(a_j) \cup \textit{Loc}(a_k)$ $\textit{Dep}(a_i) = \textit{Dep}(a_j) \cup \textit{Dep}(a_k)$
Call	$callsite\ c$	Call Apply_FTF(c, lev) given in Algorithm 5	

Table 1. Constraint generation for the pointer inference at level lev.

variable is defined. Initially, for an SSA variable defined by a direct (Base) assignment or a χ (Chi) assignment whose M field is *must*, its mod is initialized to be "*must*". For every other SSA variable, its mod is initialized to be "*may*". When resolving the points-to constraints, we propagate the value of mod from node to node along the constraint edges corresponding to ϕ assignments in the constraint graph. The left-hand side variable of a ϕ operation is *must*-defined if and only if all its operands are.

3.2 Top-Down Analysis

The top-down analysis traverses ACG and processes each of its nodes in topological order, as shown in Algorithm 6. We propagate the points-to sets of formal-ins to their use sites at each call site with actual parameters being bound to formal parameters (lines 6 – 20). Again, due to the presence of recursion cycles, the computation of $\textit{Ptr}(f)$ for a formal-in may have to be carried out iteratively. Recall that the points-to set of a pointer computed during pointer inference in the bottom-up phase may depend on the points-to sets of some formal-ins. As a result of this points-to set propagation, the points-to sets of all pointers at the level being analyzed are fully resolved. In addition, we expand the pointer dereferences of the variables at the level being analyzed by inserting the μ or χ operators for them to expose the def/use points for the pointed-to variables so that they can be analyzed at the next level and lower levels (lines 24 – 54).

For the program in Figure 1, we perform the top-down analysis for level 2 immediately after the bottom-up analysis for this level is finished. In the top-down phase analyzing $main$, we know that at L4, x must point to a and y must point to b. At L9, x may point to $\{c, d\}$ and y may point to $\{d, e\}$. We propagate the points-to sets of x and y to p and q, respectively, so that p points to $\{a, c, d\}$ and q points to $\{b, d, e\}$ as given in (3). Since $main$ has no pointer dereferences, we proceed to analyze foo in the top-down phase. By expanding the pointer dereferences $*p$ and $*q$, we obtain the code in Figure 5 with the newly introduced μ and χ operators, which are used in analyzing the pointers at the next level, i.e., level 1.

```
void foo( int **p, int **q)
{
          μ(b, q ⇒ b)
          μ(d, q → d)
          μ(e, q → e)
L11:   tmp1 = *q1;

L12:   *p1 = tmp1;
          a=χ(a, p ⇒ a, must)
          c=χ(c, p → c, may)
          d=χ(d, p → d, may)

L13:   tmp2 = &obj;

L14:   *q1 = tmp2;
          b=χ(b, q ⇒ b, must)
          d=χ(d, q → d, may)
          e=χ(e, q → e, may)
}
```

Figure 5. foo with the μ and χ operations introduced during top-down analysis at level 1 (without using parameterized spaces).

Let us revisit the notion of parameterized spaces discussed earlier. Many variables accessed in a procedure do not explicitly appear in its body since they only appear implicitly in some μ or χ operators, either through pointer dereferences or call statements. If we use a unique variable to represent the variables that have the same def-use chains, we can save a lot of space and reduce the analysis time as well. For the program in Figure 5, a and c have the same def/use points, and similarly for b and e. In a program, a formal-in parameter may point-to many variables at different call sites. So a dereference of a formal-in parameter may produce a lot of μ or χ operators, resulting in space pressure. We merge the side effects on such formal-ins by using a unique variable, called a parameterized

Algorithm 6 Top-Down Analysis.

```
1:  procedure Top-down_analysis($AVG$, $lev$)
2:  begin
3:  for each node $scc$ in topological order of $AVG$ do
4:      for each procedure $proc$ of $scc$ do
5:          for each $callsite$ of $proc$ do
6:              for each actual $v$ or $\mu(v, \mathbb{C})$ associated with $callsite$,
                where $ptl(v) = lev$ do
7:                  for each $callee$ of $callsite$ do
8:                      Map $v$ to a formal-in $f$ of $callee$;
9:                      if $|Loc(v)| == 1$ && $Dep(v) == \{\}$ then
10:                         $M$ == "must";
11:                     else
12:                         $M$ == "may";
13:                     end if
14:                     for each $\langle p, \mathbb{C}_p \rangle$ of $Loc(v)$ do
15:                         $Ptr(f)$ = $Ptr(f) \cup \{\langle p, M \rangle\}$;
16:                     end for
17:                     for each $\langle p, \mathbb{C}_p \rangle$ of $Dep(v)$ do
18:                         $Ptr(f)$ = $Ptr(f) \cup Ptr(p)$;
19:                     end for
20:                 end for
21:             end for
22:         end for
23:         ComVars = Alloc_Parameterized_Spaces($proc$, $lev$);
24:         for each assignment $S :=_{def} a = *b$ of $proc$ do
25:             if $ptl(b) == lev$ then
26:                 for each $\langle v, \mathbb{C} \rangle$ of $Loc(b)$ do
27:                     Insert $\mu(v, \mathbb{C})$ to $\mu$ list of $S$
28:                 end for
29:                 for each $\langle p, \mathbb{C} \rangle$ of $Dep(b)$ do
30:                     Insert $\mu(V^p, \mathbb{C})$ to $\mu$ list of $S$
31:                     for each $\langle v, \mathbb{C}', M \rangle$ of $ComVars(p)$ do
32:                         Insert $\mu(v, \mathbb{C} \wedge \mathbb{C}')$ to $\mu$ list of $S$
33:                     end for
34:                 end for
35:             end if
36:         end for
37:         for each assignment $S :=_{def} *b = a$ of $proc$ do
38:             if $ptl(b) == lev$ then
39:                 if ($|Loc(b)| == 1$ && $Dep(b) == \{\}$) || ($Loc(b)$
                    $== \{\}$ && $|Dep(b)| == 1$) then
40:                     $M'$ = "must"
41:                 else
42:                     $M'$ = "may"
43:                 end if
44:                 for each $\langle v, \mathbb{C} \rangle$ of $Loc(b)$ do
45:                     Insert $\chi(v, \mathbb{C}, M')$ to $\chi$ list of $S$
46:                 end for
47:                 for each $\langle p, \mathbb{C} \rangle$ of $Dep(b)$ do
48:                     Insert $\chi(V^p, \mathbb{C}, M')$ to $\chi$ list of $S$
49:                     for each $\langle v, \mathbb{C}', M \rangle$ of $ComVars(p)$ do
50:                         Insert $\chi(v, \mathbb{C} \wedge \mathbb{C}', M \sqcap M')$ to $\chi$ list of $S$
51:                     end for
52:                 end for
53:             end if
54:         end for
55:     end for
56: end for
57: end
```

Algorithm 7 Allocating Parameterized Spaces (for $lev - 1$).

```
1:  procedure Alloc_Parameterized_Spaces($proc$, $lev$);
2:  begin
3:  for each formal-in $p$ of $proc$, where $ptl(p) = lev$ do
4:      Let $Meet(proc, lev) = \langle Ptr(p), \mathbb{C}_p \rangle$,
5:      Let $V^p$ be a parameterized space representing a subset of
        the pointed-to objects in $Ptr(p)$ such that if $v$ is explicitly
        accessed in $proc$, where $\langle v, M \rangle \in Ptr(p)$, then $v \notin V^p$;
6:  end for
7:  Refine all parameterized spaces thus obtained so that they are
    pair-wise disjoint and as large as possible;
8:  for each formal-in $p$ of $proc$, where $ptl(p) = lev$ do
9:      for each $\langle v, M \rangle \in Ptr(p)$ such that $v \notin V^p$ do
10:         Let $\mathbb{C}_v$ be $q \Rightarrow v(q \rightarrow v)$ if $M$ is "must" ("may");
11:         $ComVars(p)$ = $ComVars(p) \cup \{\langle v, \mathbb{C}_v, M \rangle\}$
12:     end for
13: end for
14: end
```

space, to represent the dereference of a formal-in parameter. This is done by calling Algorithm 7 in line 23 of Algorithm 6. However, care must be taken to avoid losing any precision. If two formal-in parameters may point-to a common variable v, then v must not be parameterized (unlike [9, 23]). These unparameterized formal-ins are collected in *ComVars* in Algorithm 7. If a variable v appears in the procedure body directly, then v is not represented by any parameterized space. By using parameterized spaces, the code in Figure 5 becomes as shown in Figure 6.

```
void foo( int **p, int **q)
{
              μ(V^q, true)
              μ(d, q → d)
L11:    tmp = *q;

L12:    *p = tmp;
              V^p=χ(V^p, true, must)
              d=χ(d, may, p → d)

L13:    tmp = &obj;

L14:    *q = tmp;
              V^q=χ(V^q, true, must)
              d=χ(d, may, q → d)
}
```

Figure 6. Code of foo in Figure 5 using parameterized spaces.

Finally, the pointer analysis at a level lev may have to be performed iteratively in the presence of points-to cycles formed by some pointers at lev. During pointer dereferencing, if a pointed-to variable introduced as an operand of a μ or χ operation happens to be at lev, a points-to cycle has been detected. Whenever this happens, the pointer analysis for the same level is repeated so that more points-to relations for the pointers at lev may be discovered, resulting in potentially more μ and χ statements to be introduced. This iterative process stops as soon as all pointed-to variables discovered in the last iteration are at a lower level than lev.

4. Experiments

We have implemented our LevPA algorithm in the Open64 compiler using the BDD library cudd-2.4.2. Our current implementation consists of over 20,000 lines of C++ code. We have measured

Benchmark	KLOC	#Pointers	#Callsites	#Indirect Callsites	#Recursion Cycles	#Proc in Largest Recursion Cycle	#Points-to Relations in BDDs
icecast-2.3.1	22	1618	877	40	14	1	350
sendmail	115	31004	19578	364	40	28	176640
httpd	128	20162	8992	270	23	6	4360
445.gombk	197	16076	10078	44	26	22	17433
wine-0.9.24	1905	336591	393689	24376	264	113	159149
wireshark-1.2.2	2383	333654	245278	2230	123	30	75899

Table 2. Benchmark characteristics.

Benchmark	LevPA: 64 Bit / 32 Bit (secs)				Mem (MB)	Bootstrapping [14]
	Points-to Levels	Recursion	Function Pointers	Total		
icecast-2.3.1	0.30 / 0.40	0.07 / 0.10	0.26 / 0.68	2.18 / 5.73	30	- / 29
sendmail	2.91 / 7.15	4.27 / 11.00	22.26 / 35.30	72.63 / 143.68	568	- / 939
httpd	0.23 / 0.53	0.50 / 1.70	2.30 / 5.39	16.32 / 35.42	136	- / 161
445.gombk	1.93 / 4.47	0.30 / 0.69	3.72 / 6.67	21.37 / 40.78	691	-
wine-0.9.24	45.79 / 120.75	15.10 / 27.32	35.66 / 76.8	502.29 / 891.16	2526	-
wireshark-1.2.2	17.86 / 52.31	11.97 / 24.63	17.36 / 56.26	366.63 / 845.23	2288	-

Table 3. Analysis statistics.

its performance by using the six benchmarks listed in Table 2. The first three benchmarks are taken from Kahlon's paper [14] in order to compare the efficiency between the two methods. To our best knowledge, Kahlon's method is one of the latest FSCS pointer analysis techniques. Of these three benchmarks, `icecast`, `sendmail` and `httpd`, `httpd` is the largest used in his experiments. To evaluate the ultimate performance of our method on larger benchmarks, we have also selected three more benchmarks, `gombk`, `wine` and `wireshark`. The characteristics of these benchmarks are summarized in Table 2, including the numbers of lines, pointers, call sites, indirect call sites, recursion cycles and procedures contained in the largest recursion cycle (Columns 2 – 7). The last column gives the number of points-to relations used for building the context conditions in terms of BBDs (as discussed in Section 3.1.1).

We conducted our experiments on two computer platforms: an Intel 64-bit 2.66GHz Xeon system with 16GB RAM and an Intel 3.0GHz Pentium 4 with 2GB RAM. To compare with Kahlon's method, we have selected our 32-bit Intel 3.0GHz Pentium4 with 2GB RAM purposely to analyze the first three benchmarks given in Table 2 since Kahlon conducted his experiments on a slightly faster Intel 3.2GHz Pentium4 system with the same amount of RAM as our 32-bit system. Table 3 gives the analysis times of the three benchmarks by LevPA. In the last column, the times for the first three benchmarks consumed by Kahlon's method are taken directly from his paper [14]. For all the three benchmarks, our method is a few times faster.

Looking again at Table 3, Column 5 gives the times taken by LevPA for analyzing all the six benchmarks on each platform. In addition, Columns 2 – 4 give the times consumed by some internal phases of our method. In particular, Column 2 gives the time spent computing the points-to levels for each benchmark on both platforms, including the time elapsed in the Steensgaard-styled pointer analysis. Column 3 gives the time due to the iterative analysis for handling recursive calls for each benchmark. Column 4 gives the time taken due to the iterative analysis required for resolving function pointers. Due to the existence of recursion or indirect calls, we need to re-analyze pointers iteratively. As discussed in Section 3.1, we have adopted a demand-driven strategy to re-analyze the procedures in a cycle only when they may need to be analyzed. Of the six benchmarks, `wireshark` has more than two million lines of code. To our knowledge, this paper is the first to run a FSCS pointer analysis on benchmarks of this scale in minutes.

This level of scalability of LevPA is attributed to the facts that LevPA conducts its pointer analysis on the full-sparse SSA form, level by level, by avoiding redundant μ and χ operators at call sites and pointer dereferences and by making use of BDDs to produce a full transfer function and a meet function for a procedure that are precise as well as efficiently applicable to all calling contexts.

5. Related Work

There are many flow- and context-sensitive pointer analysis techniques reported during the last two decades, such as [4, 6, 9, 10, 14–17, 20, 25, 28] and some references therein.

Landi and Ryder [16] give a method that performs an iterative dataflow analysis on the CFG of a program while maintaining an alias relation set at each program point. Their method is not fully flow-sensitive because it cannot perform indirect strong updates. When encountering a procedure call, it enters into the body of a callee and recomputes the alias set at its exit. So this method can be classified a cloned-based context-sensitive analysis.

Choi *et al.* [6] present a method that performs an iterative dataflow analysis on a Sparse Evaluation Graph (SEG) rather on a CFG. SEG is a simplified CFG with the nodes that do not manipulate pointer information omitted. Their interprocedural algorithm is summary-based but not fully context-sensitive because it only considers one level of calling contexts.

Emani *et al.*'s method [9] is an interval analysis. The algorithm has an exponential time complexity since it is clone-based.

As discussed earlier in Section 3.1.1, Wilson and Lam [25] use partial transfer functions to summarize the behavior of already analyzed procedures. Unlike full transfer functions, partial transfer functions describe the output of a procedure based on a specific input. Their method is an iterative dataflow analysis and is fully flow-sensitive. It can scale up to 20 KLOC of C code.

Chatterjee *et al.* [9] use unknown initial values for parameters and global variables so that the summaries about a procedure can be computed by flow-sensitive alias analysis. They use a two-phase interprocedural analysis framework to compute flow- and context-sensitive alias information. Their analysis is not sparse since it needs to maintain a points-to graph at each program point. In addition, their strategy for handling function pointers is conservative. They use an over-approximated call graph. For an indirect call site, all functions with names taken and identical signatures are considered as possible callees.

Zhu [28] proposes a flow- and context-sensitive pointer analysis by using BDDs. His analysis is symbolic but not fully flow-sensitive because the kill information can only come from direct assignments. His method can scale to 200 KLOC of C code.

Zheng and Yew [27] is the first to propose the idea of performing the FSCS pointer analysis level by level. However, their method relies on type declarations to determine points-to levels and does not handle recursive data structures well. In addition, it does not use SSA form to optimize the analysis, and uses a different form of context-sensitivity.

Kahlon [14] proposes a bootstrapping algorithm that partitions a program into small subsets and uses a divide-and-conquer strategy to concurrently find solutions in these subsets. A summary-based approach is used for scalable context-sensitive alias analysis. His work is also based on the level-by-level strategy but his definition of points-to level (called depth) does not go all the way as ours. Furthermore, his approach to analyzing a level, by first computing flow- and context-insensitive alias summary and then analyzing pointers flow- and context-sensitively differs from ours. His algorithm can scale up to 128 KLOC of C code.

Kang *et al.* [15] propose a bottom-up flow- and context-sensitive pointer analysis. It is based on a new modular pointer analysis domain called the update history that can abstract memory states of a procedure independently of the information on aliases between memory locations and keep the information on the order of side effects performed. However, their method is not very efficient since for a 20 KLOC program, the elapsed analysis time is 213 seconds.

Chase *et al.* [3] present a flow-sensitive but context-insensitive pointer analysis. There are no experimental results reported. Tok *et al.* [23] give a similar method that also binds the def-use analysis and the pointer analysis together. Their method can scale up to 70 KLOC of C code. Hasti *et al.* [12], Lundberg *et al.* [18], Naeem *et al.* [19] and Hardekopf *et al.* [11] use SSA form to optimize a flow-sensitive but context-insensitive pointer analysis. Of these methods, Hardekopf and Lin's [11] is semi-sparse because in the sense tha it only builds the SSA form for the top-level pointer variables.

6. Conclusion

We have proposed a practical and scalable flow- and context-sensitive (FSCS) pointer analysis for C programs. We analyze the pointers level by level according to their points-to levels on the points-to graph in order to perform fast and accurate full sparse flow-sensitive analysis in a flow-insensitive style. This level-by-level strategy also facilitates a fast and accurate summary-based context-sensitive analysis. Our pointer analysis results in 1) a precise and compact SSA form for subsequent program analyses and optimizations, and 2) a flow- and context-sensitive mod/ref set for each procedure. We have implemented our algorithm in Open64 and our preliminary results show our new approach can analyze some large benchmarks with over a million lines of C code in minutes.

Acknowledgments

Thanks to Zhaowei Ding who for building the whole environment for our experiments. This work is supported in part by a Chinese National Basic Research Grant (2005CB321602), a Chinese National High Technology Research and Development Grant (2008AA01Z115), a Chinese National Science and Technology Major Project (2009ZX01036-001-002), a NSFC Innovation Research Group Project (60921002), a Chinese National Science Fund for Distinguished Young Scholars (60925009), and an Australian Research Council Grant (DP0987236).

References

[1] ANDERSEN, L. O. Program analysis and specialization for the c programming language, 1994.

[2] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35*, 8 (1986), 677–691.

[3] CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), ACM, pp. 296–310.

[4] CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. Relevant context inference. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1999), ACM, pp. 133–146.

[5] CHENG, B.-C., AND HWU, W.-M. W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM, pp. 57–69.

[6] CHOI, J.-D., BURKE, M., AND CARINI, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1993), ACM, pp. 232–245.

[7] CHOW, F. C., CHAN, S., LIU, S.-M., LO, R., AND STREICH, M. Effective representation of aliases and indirect memory operations in ssa form. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction* (London, UK, 1996), Springer-Verlag, pp. 253–267.

[8] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, K. F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13* (1991), 451–490.

[9] EMAMI, M., GHIYA, R., AND HENDREN, L. J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation* (New York, NY, USA, 1994), ACM, pp. 242–256.

[10] HACKETT, B., AND AIKEN, A. How is aliasing used in systems software? In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), ACM, pp. 69–80.

[11] HARDEKOPF, B., AND LIN, C. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), ACM, pp. 226–238.

[12] HASTI, R., AND HORWITZ, S. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings*

of the ACM SIGPLAN 1998 conference on Programming language design and implementation (New York, NY, USA, 1998), ACM, pp. 97–105.

[13] HIND, M., BURKE, M., CARINI, P., AND CHOI, J.-D. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst. 21*, 4 (1999), 848–894.

[14] KAHLON, V. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 249–259.

[15] KANG, H.-G., AND HAN, T. A bottom-up pointer analysis using the update history. *Inf. Softw. Technol. 51*, 4 (2009), 691–707.

[16] LANDI, W., AND RYDER, B. G. A safe approximate algorithm for interprocedural aliasing. *SIGPLAN Not. 27*, 7 (1992), 235–248.

[17] LIVSHITS, V. B., AND LAM, M. S. Tracking pointers with path and context sensitivity for bug detection in c programs. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2003), ACM, pp. 317–326.

[18] LUNDBERG, J., AND LÖWE, W. A scalable flow-sensitive points-to analysis. In *Compiler Construction - Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. h.c. Gerhard Goos* (2007), Springer Verlag. accepted.

[19] NAEEM, N. A., AND LHOTÁK, O. Efficient alias set analysis using ssa form. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management* (New York, NY, USA, 2009), ACM, pp. 79–88.

[20] RYDER, B. G., LANDI, W. A., STOCKS, P. A., ZHANG, S., AND ALTUCHER, R. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst. 23*, 2 (2001), 105–186.

[21] STEENSGAARD, B. Points-to analysis by type inference of programs with structures and unions. In *CC '96: Proceedings of the 6th International Conference on Compiler Construction* (London, UK, 1996), Springer-Verlag, pp. 136–150.

[22] STEENSGAARD, B. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), ACM, pp. 32–41.

[23] TOK, T., GUYER, S., AND LIN, C. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *Compiler construction: 15th international conference, CC 2006, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006: proceedings* (2006), Springer-Verlag New York Inc, p. 17.

[24] WHALEY, J., AND LAM, M. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation* (2004), ACM New York, NY, USA, pp. 131–144.

[25] WILSON, R. P., AND LAM, M. S. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (New York, NY, USA, 1995), ACM, pp. 1–12.

[26] YU, H., AND ZHANG, Z. An Aggressive field-sensitive unification-based pointer analysis. *Chinese Journal of Computers 32*, 9 (2009).

[27] ZHENG, B., AND CHUNG YEW, P. A hierarchical approach to context-sensitive interprocedural alias analysis, 1999.

[28] ZHU, J. Towards scalable flow and context sensitive pointer analysis. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference* (New York, NY, USA, 2005), ACM, pp. 831–836.