SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA

Yulei Sui [1], Sen Ye [1], Jingling Xue [1], Pen-Chung Yew [2]

[1] School of Computer Science and Engineering, UNSW, Australia

[2] Department of Computer Science and Engineering, University of Minnesota, USA

Abstract. We present a new SPAS (*S*calable *PA*th-*S*ensitive) framework for resolving points-to sets in C programs that exploits recent advances in pointer analysis. SPAS enables intraprocedural path-sensitivity to be obtained in flow-sensitive and context-sensitive (FSCS) techniques scalably, by using BDDs to manipulate program paths and by performing pointer analysis level-by-level on a full-sparse SSA representation similarly as the state-of-the-art LevPA (the FSCS version of SPAS). Compared with LevPA using all 27 C benchmarks in SPEC CPU2000 and CPU2006, SPAS incurs 18.42% increase in analysis time and 10.97% increase in memory usage on average, while guaranteeing that all points-to sets are obtained with non-decreasing precision.

1 Introduction

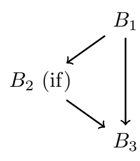
There have been great advances in pointer analysis performed flow-sensitively [9, 10, 14], context-sensitively [19, 16, 17] or with both combined [23, 11]. As reported recently [10, 11, 23], insensitive analysis can be leveraged to bootstrap sensitive analysis, thereby leading to significant improvements in scalability and precision. In particular, it is shown by LevPA [23] that flow-sensitive and context-sensitive (FSCS) analysis becomes substantially more scalable when performed on full-sparse SSA, level by level (in order of their decreasing *points-to levels*), with each level being analyzed with an inclusion-based flow-insensitive algorithm. However, little progress [20, 15] has been made when path-sensitivity is also considered. Exploiting recent advances in FSCS pointer analysis, we describe a SPAS (*S*calable *PA*th-*S*ensitive) framework that enables intraprocedural path-sensitivity to be obtained scalably for C programs on top of the state-of-the-art LevPA (the FSCS version of SPAS). SPAS obtains all points-to sets with non-decreasing precision by adding small analysis overhead in both time and space, as validated using SPEC CPU2000 and SPEC2006.

Equipped with path-sensitivity, a FSCS pointer analysis is equally as or more precise, as illustrated in Figure 1. With such path-sensitive precision, the quality of many software tools and techniques in program optimization, analysis and verification can be significantly improved. Examples include bug hunting [13], memory leak analysis [20] and software vulnerability detection [15, 21].

A major hindrance to path-sensitive pointer analysis is the lack of scalability. We tackle it by tracking program paths using Binary Decision Diagrams (BDDs) and by infusing path-sensitivity into FSCS pointer analysis seamlessly on full-sparse SSA. Our generalization is simple, drops easily on a full-sparse FSCS

```
int *q, v, w, z;
void main() {
     int **a, *f = &z;
     a = &f; q = &v;
     foo(a);
}
void foo(int **x) {
     int *g = &z;
     if (*) {
         x = &g; q = &w;
     }
     *x = q;
}
```

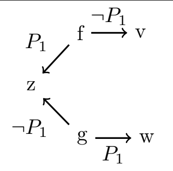
(a) Original code



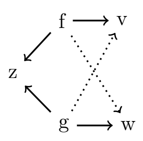
Path $P_1 : B_1 \to B_2 \to B_3$

Path $\neg P_1 : B_1 \to B_3$

(b) CFG of foo



(c) Points-to relations in *foo* **with** path-sensitivity (each points-to relation is labeled with the path along which it holds)



(d) Points-to relations in *foo* **without** path-sensitivity (the two depicted in **dashed arrows** are spurious)

Fig. 1. Effects of path-sensitivity on indirect updates at a store $*x{=}q$ in FSCS analysis.

pointer analyzer like LevPA [23], and retains the scalability of the underlying FSCS analysis (by adding small overhead in analysis time and memory usage).

Presently, SPAS captures path correlation only without ruling out infeasible paths. In summary, this paper makes the following contributions:

- SPAS is the first *full-sparse* path-sensitive FSCS pointer analysis;
- SPAS is the first to encode program paths using BDDS on full-sparse SSA;
- SPAS obtains (intraprocedural) path-sensitivity efficiently on full-sparse SSA, level by level, with each level being analyzed flow-insensitively; and
- SPAS has compatible performance as the state-of-art FSCS analyzer LevPA (the FSCS-version of SPAS). With both implemented in Open64, SPAS adds small average overhead (18.42% more in time and 10.97% more in space) for all the 27 C programs in SPEC CPU2000 and CPU2006, while ensuring that all points-to sets are obtained with non-decreasing precision. To the best of our knowledge, SPAS is the fastest path-sensitive FSCS pointer analysis for C reported in the literature.

2 Related Work

There is not much reported on performing whole-program pointer analysis flow-, context- and path-sensitively. We review some solutions using sparse SSA and BDDs and some limited amount of prior work on path-sensitive pointer analysis.

Sparsity Unlike iterative dataflow-based pointer analysis [12, 3, 7], SSA-based pointer analysis [9, 10, 23] is sparse and thus more scalable, as SSA allows points-to information to flow directly from variable definitions to their uses only.

In [9], Hardekopf and Lin presented a semi-sparse flow-sensitive analysis. By putting *top-level pointers* in SSA, their def-use information can be exposed directly. Lately [10], they generalized their work by making it full-sparse. This is done by using a flow-insensitive inclusion-based pointer analysis to compute the required def-use information in order to build SSA for all variables. However, their algorithms are not context-sensitive. Yu et al. [23] introduced LevPA for performing FSCS pointer analysis on full-sparse SSA. In LevPA, points-to resolution and SSA construction are performed together, level by level, in decreasing order of their points-to levels. Our SPAS framework is a scalable generalization of FSCS pointer analyzers like LevPA to obtain intraprocedural path-sensitivity.

BDDs Berndl et al. [1] proposed to use BDDs to encode points-to relations and also studied the impact of BDD variable ordering on analysis performance. Later, BDDs were also used to encode transfer functions [24] and contexts [19, 23] in context-sensitive analysis. In [21], Xie and Aiken discussed to use BDDs to represent program paths and simplify paths heuristically in SATURN, a SAT-based bug detection tool. In our SPAS pointer analyzer, BDDs are used to encode paths (and contexts) on full-sparse SSA using a standard BDD library. This enables the pointers to be resolved using a guarded inclusion-based insensitive analysis on full-sparse SSA, where the guards are contexts and program paths.

Path-Sensitivity We are unaware of any prior FSCS pointer analyzer supporting path-sensitivity on sparse SSA. In some bug-hunting tools like Prefix [2], path-sensitivity was exploited to look for bugs in some selected paths. A recent sound and complete generalization for SATURN [6] can compute various program properties using a SAT solver with even interprocedural path-sensitivity. However, it is not suited for whole-program points-to resolution.

SPAS and bug analysis aim to achieve different goals. Some bug detection tools rule out some infeasible paths based on branch conditions to reduce false positives but SPAS presently does not. However, SPAS can already provide more precise points-to information to make these tools more effective. Finally, Gutzmann et al. [8] discussed how to filter out spurious points-to relations flowing out of a branch node but their approach neither captures path correlation as SPAS does nor rules out infeasible paths.

3 The Basic Idea

SPAS analyzes all features of C by considering four types of assignments: $x = y$ (*copy*), $x = \&y$ (*address*), $*x = y$ (*store*) and $x = *y$ (*load*). SPAS is field-insensitive for arrays (by not distinguishing array elements) but field-sensitive for structs (by flattening and replacing them with separate variables, one for each field). SPAS names abstraction heap objects by their allocation sites.

The following sets and functions are used in some definitions given below.

- $\mathbb{O}$: set of abstract memory locations representing the variables in a program.
- $\mathcal{L} : \mathbb{O} \to \{0, \dots, L\}$: a *level map* giving each variable a points-to level. If q may be modified by operations on p (possibly indirectly), then $\mathcal{L}(p) \geq \mathcal{L}(q)$.
- $\mathbb{C}$: set of contexts represented as Boolean expressions over the set $\mathbb{O} \times \mathbb{O}$ of points-to relations. The notation $p \to o$ means that p may point to o.
- $\mathbb{D}$: set of paths represented as Boolean conditions over the set of *decision variables* (encoding the branches in a CFG to be introduced in Section 5.1).
- $\mathcal{C} : \mathbb{C} \times \mathbb{D}$: set of combined context and path conditions used to specify under which condition in $\mathcal{C}$ a pointer may point to a memory object in $\mathbb{O}$.

Definition 1 (Formal-Ins). *Given a method m, $\mathbb{F}^m_{\text{in}} \subseteq \mathbb{O}$ denotes the set of its* formal-ins*, i.e., its formal parameters and non-local variables accessed in m.*

Definition 2 (Formal-Outs). *Given a method m, $\mathbb{F}^m_{\text{out}} \subseteq \mathbb{O}$ denotes the set of its* formal-outs*, i.e., its* return parameter *(a special local variable of m containing its return value) and non-local variables that may be modified in m.*

In flow-sensitive analysis, we speak of the points-to sets of a pointer p at program points, which are identified by the versions of p at those points in SSA.

Definition 3 (Points-to Sets). *The points-to set of a pointer p at a program point, $\mathsf{PtrSet}(p) \subseteq \mathbb{O}$, is a set of locations in $\mathbb{O}$ possibly pointed to by p.*

SPAS achieves context-sensitivity by traversing the call graph of a program bidirectionally. During a bottom-up traversal, the points-to sets of a pointer p at a point may be related to those of some formal-ins in terms of points-to maps (Definition 4). During a top-down traversal, p is resolved once the points-to sets of all its dependent formal-ins have been found (Definition 3).

Definition 4 (Points-to Maps). *The points-to map of a pointer p at a program point in a method m is given by:*

$$\mathsf{PtrMap}(p) = (\mathsf{Loc}(p), \mathsf{Dep}(p)) \tag{1}$$

where $\mathsf{Loc}(p) \subseteq \mathbb{O} \times \mathcal{C}$ contains each tuple (v, C_v, P_v) such that p may point to v in context C_v along path P_v within method m and $\mathsf{Dep}(p) \subseteq \mathbb{F}^m_{\text{in}} \times \mathcal{C}$ contains each tuple (f, C_f, P_f) such that p may point to what formal-in f points to in context C_f along path P_f within method m.

SPAS is intraprocedurally path-sensitive. Thus, the transfer functions (*MOD* and *USE*) of a formal-out in a method m are predicated by m's calling contexts (without m's path conditions). Similarly, the path conditions in their specified side effects are also ignored (and hence, the $*$'s). To support strong updates, we distinguish MAY-DEFs and MUST-DEFs at stores.

Definition 5 (*MOD*). *The transfer function MOD of a formal-out $f_{\text{out}} \in \mathbb{F}^m_{\text{out}}$ in a method m describes its interprocedural modification side effect:*

$$MOD(m, f_{\text{out}}) = (\mathsf{Loc}(f_{\text{out}}), \mathsf{Dep}(f_{\text{out}}), C^{\text{may}}_{f_{\text{out}}}, C^{\text{must}}_{f_{\text{out}}}) \tag{2}$$

indicating that f_{out} may be modified, i.e., MAY-DEF'ed in context $C^{\text{may}}_{f_{\text{out}}} \in \mathbb{C}$ to point to either (a) v for each $(v, C_v, *) \in \mathsf{Loc}(f_{\text{out}})$ when C_v holds or (b) whatever f points to for each $(f, C_f, *) \in \mathsf{Dep}(f_{\text{out}})$ when C_f holds. If $C^{\text{must}}_{f_{\text{out}}} \in \mathbb{C}$ also holds, then the MAY-DEF is actually a MUST-DEF.

Definition 6 (*USE*). *The transfer function USE of a formal-in $f_{\text{in}} \in \mathbb{F}^m_{\text{in}}$ in a method m describes its MAY-USE, i.e., interprocedural read side effect:*

$$USE(m, f_{\text{in}}) = (\mathsf{PtrSet}(f_{\text{in}}), C_{f_{\text{in}}}) \tag{3}$$

indicating that what is pointed to by f_{in} may be read in context $C_{f_{\text{in}}} \in \mathbb{C}$. (If m is a formal parameter, it does not need a USE function as m is local.)

The basic idea behind SPAS is simple. The pointers are resolved in order of their decreasing points-to levels by maintaining the invariant stated below.

*Property 1 (***Level-Wise Invariant).** Just before level ℓ is analyzed, (a) all (direct and indirect) accesses to the pointers at higher levels are in SSA, with the indirect accesses via pointer dereferencing and calls being expressed using μ (MAY-USE) and χ (MAY/MUST-DEF) operations [4], (b) all pointers at higher levels have been soundly resolved, and (c) all indirect accesses made by dereferencing the pointers at level ℓ are exposed using μ and χ operations.

The analysis performed at level ℓ is to ensure that this invariant holds at the beginning of $\ell - 1$. This is done by traversing the call graph of a program first bottom-up and then top-down iteratively. During bottom-up analysis, SPAS analyzes each method m by (B1) building its SSA (doable as Property 1(c) holds for ℓ) and (B2) computing the points-to maps for its pointers at ℓ (Definition 4). Prior to (B1), SPAS inserts μ and χ operations for each of its call sites, c, to expose the MAY-USEs and MAY/MUST-DEFs made by c's callees to every pointer at ℓ. This is done by applying the callees' transfer functions at call site c. After (B2) is done, the transfer functions of method m are computed. During top-down analysis, SPAS (T1) resolves the points-to sets of the pointers at ℓ by propagating the dependent points-to sets to formal-ins (Definition 3) and (T2) annotates the dereferences to these pointers with μ and χ operations.

4 A Motivating Example

We describe how SPAS improves FSCS pointer analysis by refining Figure 1(d) into Figure 1(c). This program may look complex but it appears to be one of the smallest examples that we can come up with in order to illustrate all key aspects of SPAS. The variables are partitioned into three levels: $\{a, x\}$ at level 2, $\{q, f, g\}$ at level 1 and $\{v, w, z\}$ at level 0. We examine the top two levels only.

Level 2 The pointers a and x are considered. The input is Figure 1(a), for which Property 1 holds trivially at this level. The output is given in Figure 2(a).

– **Bottom-Up** When *foo* is analyzed, all accesses to x are first put in SSA. During points-to resolution, the points-to maps for its three definitions are found. In particular, x_0 is recorded to point to what formal-in x points to. By analyzing the ϕ node for x path-sensitively, x_2 is found to point to g in context true (i.e., any context) along path P_1 and what x points to in context true along $\neg P_1$. When *main* is analyzed, the points-to map of a_0 is found.
– **Top-Down** In *main*, a_0 has been resolved locally. Binding a_0 with formal-in x at the call site to *foo* reveals that $\mathsf{PtrSet}(x_0) = \{f\}$. When *foo* is processed, the MAY-DEF to f (g) via $*x_2$ is exposed by a χ operation, where context condition $x \to f$ (true) indicates that the MAY-DEF occurs when formal parameter x points to f (in any context). Like LevPA, SPAS uses points-to relations holding at a call site to represent and distinguish calling contexts.

Level 1 The pointers q, f and g are considered. The input is Figure 2(a), for which Property 1 holds at this level. The output is given in Figure 2(b).

– **Bottom-Up** When *foo* is analyzed, all accesses to the three pointers (including the two MAY-DEFs) are first put in SSA. Like x, $q_0 = q$ and $f_0 = f$ are inserted for the formal-ins q and f. Here, q is a global and f is an invisible [12] accessed via pointer dereferening. The points-to resolution for q is done similarly as x with the points-to maps obtained as shown. We now consider how store $*x_2 = q_2$ is analyzed together with its two MAY-DEFs to resolve f_1 and g_1. By capturing path correlation, SPAS deduces that f_1 points to whatever formal-in q does along path $\neg P_1$ and whatever formal-in f does

(a) After level 2 is analyzed

```
int *q, v, w, z;
void main() {
    int **a, *f = &z;
    a0 = &f; q = &v;
    foo(a0);
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)
    int *g = &z;
    if (*) {
        x1 = &g; q = &w;
    }
    x2 = φ(x0, x1);
   ┌──────────────────────────────┐
   │*x2 = q;                      │
   │  f = χ(f, x → f, ¬P1, MAY);  │
   │  g = χ(g, true, P1, MAY);    │
   └──────────────────────────────┘
}
```

main:
$\mathsf{PtrMap}(a_0) = (\{(f, \mathsf{true}, \mathsf{true})\}, \emptyset)$
foo:
$\mathsf{PtrMap}(x_0) = (\emptyset, \{(x, \mathsf{true}, \mathsf{true})\})$
$\mathsf{PtrMap}(x_1) = (\{(g, \mathsf{true}, P_1)\}, \emptyset)$
$\mathsf{PtrMap}(x_2) = (\{(g, \mathsf{true}, P_1)\}, \{(x, \mathsf{true}, \neg P_1)\})$
$\mathsf{PtrSet}(x_0) = \{f\}$

(b) After level 1 is analyzed

```
int *q, v, w, z;
void main() {
    q0 = q; // formal-in q identified as q0 (ver 0)
    int **a, *f0 = &z;
    a0 = &f; q1 = &v;
   ┌──────────────────────────────┐
   │ μ(q1, true, true);           │
   │foo(a0);                      │
   │ f1= χ(f0, true, true, MAY);  │
   └──────────────────────────────┘
}
void foo(int **x) {
    x0 = x; // formal-in x identified as x0 (ver 0)
    q0 = q; // formal-in q identified as q0 (ver 0)
    f0 = f; // formal-in f identified as f0 (ver 0)
    int *g0 = &z;
    if (*) {
        x1 = &g; q1 = &w;
    }
    x2 = φ(x0, x1);
    q2 = φ(q0, q1);
   ┌────────────────────────────────┐
   │*x2 = q2;                       │
   │  f1 = χ(f0, x → f, ¬P1, MAY);  │
   │  g1 = χ(g0, true, P1, MAY);    │
   └────────────────────────────────┘
}
```

main:
$\mathsf{PtrMap}(f_0) = (\{(z, \mathsf{true}, \mathsf{true})\}, \emptyset)$
$\mathsf{PtrMap}(q_1) = (\{(v, \mathsf{true}, \mathsf{true})\}, \emptyset)$
$\mathsf{PtrMap}(f_1) = (\{(z, \mathsf{true}, \mathsf{true}), (v, \mathsf{true}, \mathsf{true})\}, \emptyset)$
foo:
$\mathsf{PtrMap}(q_0) = (\emptyset, \{(q, \mathsf{true}, \mathsf{true})\})$
$\mathsf{PtrMap}(q_1) = (\{(w, \mathsf{true}, P_1)\}, \emptyset)$
$\mathsf{PtrMap}(q_2) = (\{(w, \mathsf{true}, P_1)\}, \{(q, \mathsf{true}, \neg P_1)\})$
$\mathsf{PtrMap}(f_0) = (\emptyset, \{(f, \mathsf{true}, \mathsf{true})\})$
$\mathsf{PtrMap}(f_1) = (\emptyset, \{(q, x \to f, \neg P_1), (f, x \to f, P_1)\})$
$\mathsf{PtrMap}(g_0) = (\{(z, \mathsf{true}, \mathsf{true})\}, \emptyset)$
$\mathsf{PtrMap}(g_1) = (\{(z, \mathsf{true}, \neg P_1), (w, \mathsf{true}, P_1)\}, \emptyset)$
$\mathsf{PtrSet}(f_0) = \{z\}$
$\mathsf{PtrSet}(q_0) = \{v\}$

Fig. 2. Level-wise SSA construction and path-sensitive pointer analysis for Figure 1(a).

along P_1 and that g_1 points to z along $\neg P_1$ and w along P_1. The transfer functions relevant for computing the side effects of the call to *foo* are:

$$\begin{aligned} MOD(foo, f) &= (\{\mathsf{Loc}(f_1), \mathsf{Dep}(f_1), x \to f, \mathsf{false}) \\ USE(foo, q) &= (\mathsf{PtrSet}(q_0), \mathsf{true}) \end{aligned} \tag{4}$$

When *main* is analyzed, a MAY-DEF for f is added since context $C_f^{\text{may}} = x \to f$ in $MOD(foo, f)$, once mapped to $a \to f$ at the call site, holds. However, this is not a MUST-DEF since $C_f^{\text{must}} = \mathsf{false}$. In addition, a MAY-USE for q is added. Then the SSA form is built. Finally, by performing the points-to resolution for f and q with the modification side effects of *foo* on f being accounted for, we obtain their points-to maps as shown.

– **Top-Down** When *main* is analyzed, its pointers at this level are already resolved. Propagating the points-to sets of q_1 and f_0 at the call site to *foo*, we find $\mathsf{PtrSet}(f_0) = \{z\}$ and $\mathsf{PtrSet}(q_0) = \{v\}$ in *foo*. When *foo* is processed next, all its pointers at this level can be resolved by Definition 4. As a result, the points-to relations for f_1 and g_1 are obtained as shown in Figure 1(c).

SPAS obtains such improved precision with a slight increase in analysis overhead. There are several reasons for SPAS to achieve this level of scalability:

Program Paths Manipulated as BDDs Like contexts [23], program paths are also represented and operated on using BDDs in a compact and canonical fashion, resulting in fast operations on program paths.

Full-Sparse SSA (at All Points-to Levels) As in LevPA [23], pointers are resolved, level by level, in order of their decreasing points-to levels. At the same time, the full-sparse SSA form is being built incrementally. The points-to relations of a pointer at a particular level cannot be propagated to lower-level pointers unless it has fully been resolved. Thus, the number of repropagations is reduced, leading to faster convergence for the points-to resolution.

Flow-Insensitive Points-to Resolution SSA is ideal for enabling sparse analysis because it makes def-use information explicit. Like contexts [23], program paths are also used to guard what points-to information can be propagated across an pointer assignment. Thus, our path-sensitive pointer analysis is sped up with a guarded inclusion-based flow-insensitive pointer analysis.

5 The SPAS Framework

SPAS is a summary-based FSCS pointer analyzer with intraprocedural path-sensitivity being supported. In particular, SPAS builds *MOD* and *USE* functions for each method and applies them to all its calling contexts.

Section 5.1 discusses how to encode program paths using BDDs. Section 5.2 examines the χ and μ operators added to the classic SSA form. To ease understanding, we introduce SPAS in two stages. In Section 5.3, we focus on capturing path correlation without performing strong updates. In Section 5.4, we discuss briefly but precisely how to extend it to perform path-sensitive strong updates.

5.1 Encoding Program Paths as BDDs

SPAS does not presently distinguish the paths inside a loop-induced cycle but can analyze the first few iterations of a loop path-sensitively via loop peeling.

All branch nodes are assumed to be binary. We use *decision variables* to encode branch nodes to express program paths. The edges and blocks in a CFG (with cycles collapsed) are associated with paths as follows. The path for the incoming edge of the entry block is initialized to true (representing the set of all paths). Let B be a block with n incoming edges associated with paths $P_1, \ldots, P_n$. The path for B is $P_1 \vee \cdots \vee P_n$. If B is a branch node encoded with decision variable Q, the paths for its two outgoing edges are $(P_1 \vee \cdots \vee P_n) \wedge Q$ and $(P_1 \vee \cdots \vee P_n) \wedge \neg Q$, respectively. Otherwise, its unique outgoing edge is $P_1 \vee \cdots \vee P_n$.

Our BDD encoding has three advantages. First, the number of BDD variables used is kept to a minimum. Second, it plays up the strengths of BDDs by exposing opportunities for path redundancy elimination. Third, the paths combined at a join node are effectively simplified (e.g., with $P_1 \vee \neg P_1$ being reduced into true), resulting in fast propagation of path conditions during points-to resolution.

5.2 Extended SSA Form

The classic SSA representation [5] is mainly useful for scalars without aliases. Following [23], we extend the classic SSA form by using the μ and χ operators [4] to make explicit all potential uses and definitions at loads/stores and call sites, as shown in Figure 2. A load or call site is annotated with a $\mu(v, C_v, P_v)$ operation to indicate a MAY-USE of v in context C_v along path P_v. A store or call site is annotated with a $v = \chi(v, C_v, P_v, M_v)$ operation to indicate a MAY-DEF (MUST-DEF) of v in context C_v along path P_v if M_v is MAY (MUST).

5.3 Capturing Path Correlation

This section focuses on capturing path correlation without strong updates. We consider functions with return statements in Section 5.4. As we do not distinguish MAY-DEFs and MUST-DEFs for now, the last entry $C^{\text{must}}_{\text{out}}$ in a *MOD* function (Definition 5) is ignored and the last entry in a χ operation is always a MAY. For Figure 2, all points-to maps are unchanged except for f_1 and g_1 in *foo*:

$$\begin{array}{l} \mathsf{PtrMap}(\mathrm{f}_1) = (\emptyset, \{(q, x \to f, \neg P_1), (f, x \to f, \mathsf{true})\}) \\ \mathsf{PtrMap}(\mathrm{g}_1) = (\{(z, \mathsf{true}, \mathsf{true}), (w, \mathsf{true}, P_1)\}, \emptyset) \end{array} \tag{5}$$

Without path-sensitive strong updates, the old points-to sets (i.e., $\mathsf{PtrMap}(f_0)$ for f_1 and $\mathsf{PtrMap}(g_0)$ for g_1 in Figure 2(b)) must be preserved along path "true" as above and cannot be killed path-sensitively as in Figure 2(b).

To account for the read and modification side effects at a call site, the binding between the actual and formal parameters is performed in the standard manner.

Definition 7 (Mappings of Formal-Ins and Formal-Outs). *For a formal-in $f_{\text{in}} \in \mathbb{F}^n_{\text{in}}$ of method n invoked at call site c, Callee2Caller$_{\text{in}}(c, n, f_{\text{in}})$ denotes the corresponding actual parameter of f_{in} at c if f_{in} is a formal parameter of n and f_{in} itself otherwise (i.e., if f_{in} is a nonlocal). For a formal-out $f_{\text{out}} \in \mathbb{F}^m_{\text{out}}$ of n invoked at c, Callee2Caller$_{\text{out}}(c, n, f_{\text{out}})$ denotes the variable at c that is assigned from f_{out} if f_{out} is a return parameter of n and f_{out} itself otherwise.*

In Figure 2, *Callee2Caller*$_{\text{in}}(c, foo, x) = a$, *Callee2Caller*$_{\text{in}}(c, foo, q) = q$, and *Callee2Caller*$_{\text{in}}(c, foo, f)$=*Callee2Caller*$_{\text{out}}(c, foo, f)$=$f$, where c is the call to *foo*.

Conceptually, SPAS proceeds in the following two sequential stages:

Stage 1. $\mathcal{L} =$ Partition($\mathbb{O}$) We compute $\mathcal{L}$ by using some fast flow-insensitive pointer analysis for the pointers in $\mathbb{O}$. For example, we can apply Steensgaard's algorithm [18] to obtain a points-to graph, merge all predecessors of each node, and finally, make the points-to graph acyclic by collapsing SCCs, as in [23]. The points-to level of a variable is its longest length over $\{0, \ldots, L\}$ to a sink node.

Stage 2. $\Delta_{-1} =$ Analyze($\mathcal{L}, \Delta_L, \mathbb{G}$) We build SSA and resolve pointers, level by level, from L to 0. Δ_L is the initial SSA that satisfies vacuously Property 1 for L and $\mathbb{G}$ is the initial call graph constructed when function pointers are not yet resolved. Analyze is restarted whenever new points-to relations are discovered for a function pointer. $\mathbb{G}$ is always a directed acyclic graph. In the presence of recursive calls, $\mathbb{G}$ is made acyclic by collapsing all SCCs. The analysis within each SCC is performed iteratively until a fixed-point is reached to obtain full context sensitivity for all the methods in the SCC. Once Analyze has run to completion, Δ_{-1} is the full-sparse SSA obtained that satisfies Property 1 for level -1 (excluding its Part(c)) and all pointers have been fully resolved.

When analyzing level ℓ, Analyze starts with Δ_ℓ, i.e., the SSA form that satisfies Property 1 for ℓ and ends with producing $\Delta_{\ell-1}$, i.e., the SSA form that

```
1  BottomUp (Method: $m$, Level $\ell$)
2    Step ①: $Add\_\mu\_\chi\_Callsites(m,\ell)$
3    Step ②: $Build\_SSA(m,\ell)$
4    Step ③: $Pointer\_Inference(m,\ell)$
5    Step ④: $Comp\_MOD\_USE\_Funs\ (m,\ell)$

6  ① $Add\_\mu\_\chi\_Callsites$(Method: $m$, Level: $\ell$)
7    for each call site $c$ in method $m$
8      Let $P_c$ be the path allocated to call site $c$
9      for each callee $n$ invoked at call site $c$
10       for each formal-out $f_{\text{out}} \in \mathbb{F}^n_{\text{out}}$ of $n$,
           $\mathcal{L}(f_{\text{out}}) = \ell$, that is not a return parameter
11         Let $MOD(n, f_{\text{out}}) = (*, *, C^{\text{may}}_{f_{\text{out}}})$
12         if $\overline{(C^{\text{may}}_{f_{\text{out}}}}$=$Callee2Caller_{\text{ctx}}(c, n, C^{\text{may}}_{f_{\text{out}}}))\neq$false
13           Add $f_{\text{out}} = \chi(f_{\text{out}}, \overline{C^{\text{may}}_{f_{\text{out}}}}, P_c, \text{MAY})$ for $c$
14       for each formal-in $f_{\text{in}} \in \mathbb{F}^n_{\text{in}}$ of $n$,
           $\mathcal{L}(f_{\text{in}}) = \ell$, that is not a formal parameter
15         Let $USE(n, f_{\text{in}}) = (*, C_{f_{\text{in}}})$
16         if $\overline{(C_{f_{\text{in}}}} = Callee2Caller_{\text{ctx}}(c, n, C_{f_{\text{in}}}))\neq$false
17           Add $\mu(f_{\text{in}}, \overline{C_{f_{\text{in}}}}, *)$ for $c$

18 ② $Build\_SSA$(Method: $m$, Level: $\ell$)
19   Apply the SSA construction algorithm [5]

20 ③ $Pointer\_Inference$(Method: $m$, Level: $\ell$)
21   Perform a guarded inclusion-based flow-insensitive
     pointer analysis using the rules in Table 1

22 ④ $Comp\_MOD\_USE\_Fun$(Method: $m$, Level: $\ell$)
23   See text (Section 5.3)

24 TopDown (Method: $m$, Level $\ell$)
25   Step ⑤: $Resolve\_PointsToSets(m,\ell)$
26   Step ⑥: $Add\_\mu\_\chi\_Derefs(m,\ell)$

27 ⑤ $Resolve\_PointsToSets$(Method: $m$, Level: $\ell$)
28   for each call site $c$ in method $m$
29     for each callee $n$ invoked at call site $c$
30       for each variable version $p_i$ in $m$, $\mathcal{L}(p) = \ell$,
           such that $p_i$ reaches call site $c$ and $p =$
           $Callee2Caller_{\text{in}}(c, n, f_{\text{in}})$, where $f_{\text{in}} \in \mathbb{F}^n_{\text{in}}$
31         for each $(v, C_v, P_v) \in \mathsf{Loc}(p_i)$
32           $\mathsf{PtrSet}(f_{\text{in}}) \cup= \{v\}$
33         for each $(f, C_f, P_f) \in \mathsf{Dep}(p_i)$
34           $\mathsf{PtrSet}(f_{\text{in}}) \cup= \mathsf{PtrSet}(f)$

35 ⑥ $Add\_\mu\_\chi\_Derefs$(Method $m$, Level: $\ell$)
36   for each store "$*p_i = \dots$" in $m$, $\mathcal{L}(p) = \ell$
37     Let $\mathsf{PtrMap}(p_i) = (\mathsf{Loc}(p_i), \mathsf{Dep}(p_i))$
38     for each $(v, C_v, P_v) \in \mathsf{Loc}(p_i)$
39       Add $v = \chi(v, C_v, P_v, \text{MAY})$
40     for each $(f_{\text{in}}, C_{f_{\text{in}}}, P_{f_{\text{in}}}) \in \mathsf{Dep}(p_i)$
41       for each $v \in \mathsf{PtrSet}(f_{\text{in}})$
42         Add $v = \chi(v, C_{f_{\text{in}}} \wedge f_{\text{in}} \to v, P_{f_{\text{in}}}, \text{MAY})$
43   for each load "$\dots = *p_i$" in $m$, $\mathcal{L}(p) = \ell$
44     Let $\mathsf{PtrMap}(p_i) = (\mathsf{Loc}(p_i), \mathsf{Dep}(p_i))$
45     for each $(v, C_v, P_v) \in \mathsf{Loc}(p_i)$
46       Add $\mu(v, C_v, P_v)$
47     for each $(f_{\text{in}}, C_{f_{\text{in}}}, P_{f_{\text{in}}}) \in \mathsf{Dep}(p_i)$
48       for each $v \in \mathsf{PtrSet}(f_{\text{in}})$
49         Add $\mu(v, C_{f_{\text{in}}} \wedge f_{\text{in}} \to v, P_{f_{\text{in}}}, \text{MAY})$
```

Fig. 3. Bottom-up and top-down analysis of method m at level ℓ.

satisfies Property 1 for $\ell - 1$. The call graph $\mathbb{G}$ is traversed twice, first bottom-up (reversal topologically) and then top-down (topologically). When points-to cycles are detected, level ℓ is re-reanalyzed until $\Delta_{\ell-1}$ is completely built. Thus, the contexts in a transfer function may comprise the points-to relations of some formal-ins discovered earlier at higher levels and the current level ℓ.

A context used in a callee is mapped to a caller in the standard manner by applying the context mapping introduced in Definition 8 below.

Definition 8 (Context Mapping). *Let C be a context used in a callee n invoked at a call site c in a method m. $Callee2Caller_{\text{ctx}}(c, n, C)$ denotes the mapping of C from callee n to call site c by performing a formal-to-actual parameter mapping. It is understood that every points-to relation in $Callee2Caller_{\text{ctx}}(c, n, C)$ that is not dependent on any of m's contexts is fully evaluated (to* true *or* false*).*

Figure 3 gives our algorithm for analyzing a method m at level ℓ. We describe the bottom-up phase first but both phases may have to be understood together.

To soundly capture path correlation, the path assigned to a variable at any of its definition site must not under-approximate the scope of its definition.

① BottomUp: *Add_μ_χ_Callsites* Due to Property 1(c), SPAS proceeds to expose the MAY-USEs and MAY-DEFs for each pointer at level ℓ that is accessed at a call site c. This is done by simply examining the context condition $C^{\text{may}}_{f_{\text{out}}}$ of $MOD(n, f_{\text{out}})$ (Definition 5) and the context condition $C_{f_{\text{in}}}$ of $USE(n, f_{\text{in}})$ (Definition 6) of each callee n invoked at c, which were computed earlier during the same bottom-up phase. In line 13, the path for a χ operation is safely over-approximated as P_c, i.e., the path of call site c, where f_{out} may be defined. As SPAS tracks path-sensitivity intraprocedurally, the path condition for a μ operation at a call site is irrelevant and thus marked with a '$*$'.

Let us see how the MAY-USE and MAY-DEF are added for the call site c_{foo} to *foo* in *main* in Figure 2(b), given the transfer functions of *foo* in (4). In line 12, $\overline{C_f^{\text{may}}} = \mathit{Callee2Caller}_{\text{ctx}}(c_{foo}, foo, C_f^{\text{may}}) = \mathsf{true}$ since by Definition 8, $C_f^{\text{may}} = x \to f$ is mapped to $a \to f$ at the call site, which is generated locally in *main*. So the MAY-DEF, $f = \chi(f, \mathsf{true}, \mathsf{true}, \text{MAY})$, is added. The MAY-USE, $\mu(q, \mathsf{true}, *)$, is added since $C_q = \mathsf{true}$ in $\mathit{USE}(foo, q)$.

② BottomUp: ***Build_SSA*** Once all MAY-USEs and MAY-DEFs are exposed for the pointers at level ℓ accessed, they can be put in SSA by applying the classic SSA construction algorithm [5], as illustrated in Figure 2.

③ BottomUp: ***Pointer_Inference*** Table 1 lists the seven rules for resolving the points-to maps for the pointers at level ℓ in a method m. The first six rules are illustrated in Figure 2 and the last partially when *Add_μ_χ_Callsites* is discussed.

The propagation of points-to information across an assignment may be guarded by both a context condition and a path condition. We define $\mathcal{P}(x) \times C_x \times P_x = \{(v, C_v \wedge C_x, P_v \wedge P_x) \mid (v, C_v, P_v) \in \mathcal{P}(x)\}$. $\mathcal{P}(x) \cup \mathcal{P}(y)$ includes all and only elements in $\mathcal{P}(x)$ and $\mathcal{P}(y)$ such that if $(v, C_v^x, P_v^x) \in \mathcal{P}(x)$ and $(v, C_v^y, P_v^y) \in \mathcal{P}(y)$, then both are merged as $(v, C_v^x \vee C_v^y, P_v^x \vee P_v^y)$.

Loc-Init is self-explanatory. As SPAS is intraprocedurally path-sensitive, the path $P_{p_i = \&a}$ is generated locally in method m. *Dep-Init* is applied to a copy of the form $p_0 = p$, where p is a formal-in of method m. Such copies are added at the entry of m for all its formal-ins. The path condition is over-approximated as true since p_0 may point to whatever p point to on entry of the method considered.

Assn applies to every other copy assignment. The points-to relations at the RHS are propagated to the LHS, guarded by the path of the assignment.

Rules *Mu* and *Chi* are also easy to understand. The context and path conditions in a χ or μ operation serve as the guards to enforce context- and path-sensitivity. According to the second constraint for *Chi*, the old points-to relations of v are weakly updated, i.e., simply propagated from v_t to v_s unchanged.

Let us consider Rule *Phi*. For each operand, we use the path along which its value flows into the result as the guard to propagate its points-to relations into the result. In a FSCS pointer analyzer that does not consider path-sensitivity, the two unguarded constraints $p_i \supseteq p_j$ and $p_i \supseteq p_k$ are generated. Applying these two would yield the two spurious points-to relations shown in Figure 1(d).

Finally, Rule *Call* is applied to a call site c in the standard manner. In lines C4 and C7, constraints are generated to propagate the points-to relations created by a callee n in $\mathsf{Loc}(f_{\text{out}})$ and $\mathsf{Dep}(f_{\text{out}})$ to v_s, guarded by the (mapped) context conditions C_o and $C_{f_{\text{in}}}$, respectively; but the paths created inside the callee are ignored (and hence, the $*$'s). In line C8, v is weakly updated as in Rule *Chi*.

④ BottomUp: ***Comp_MOD_USE_Funs*** For each formal-out $f_{\text{out}} \in \mathbb{F}_{\text{out}}^m$ at level ℓ, we write $f_{\text{out}}^{\max}$ for its last SSA version in method m. Let $\mathsf{PtrMap}(f_{\text{out}}^{\max}) =$

Rule	Statement	Constraints	Inference Operations
Loc-Init	$p_i = \&a$ (on path $P_{p_i=\&a}$)	$p_i \supseteq \{a\}$	$\mathsf{Loc}(p_i) = \{(a, \mathsf{true}, P_{p_i=\&a})\}$ $\mathsf{Dep}(p_i) = \emptyset$
Dep-Init	$p_0 = p$ (a formal-in)	$p_0 \supseteq p$	$\mathsf{Loc}(p_0) = \emptyset$ $\mathsf{Dep}(p_0) = \{(p, \mathsf{true}, \mathsf{true})\}$
Assn	$p_i = q_j$ (on path $P_{p_i=q_j}$)	$p_i \supseteq_{\mathsf{true} \times P_{p_i=q_j}} q_j$	$\mathcal{P}(p_i) = \mathcal{P}(q_j) \times \mathsf{true} \times P_{p_i=q_j}$
Mu	$\mu(v_k, C_{v_k}, P_{v_k})$ $p_i = *q_j$	$p_i \supseteq_{C_{v_k} \times P_{v_k}} v_k$	$\mathcal{P}(p_i) \cup= \mathcal{P}(v_k) \times C_{v_k} \times P_{v_k}$
Chi	$*p_i = q_j$ $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \text{MAY})$	$v_s \supseteq_{C_{v_s} \times P_{v_s}} q_j$ $v_s \supseteq v_t$	$\mathcal{P}(v_s) \cup= \mathcal{P}(q_j) \times C_{v_s} \times P_{v_s}$ $\mathcal{P}(v_s) \cup= \mathcal{P}(v_t)$
Phi	$p_i{=}\phi(p_j, p_k)$ ($P_{p_j^\phi}$ ($P_{p_k^\phi}$) is the path of the incoming edge along which the value of p_j (p_k) flows into p_i)	$p_i \supseteq_{\mathsf{true} \times P^\phi_{p_j}} p_j$ $p_i \supseteq_{\mathsf{true} \times P^\phi_{p_k}} p_k$	$\mathcal{P}(p_i) = \mathcal{P}(p_j) \times \mathsf{true} \times P^\phi_{p_j}$ $\mathcal{P}(p_i) = \mathcal{P}(p_k) \times \mathsf{true} \times P^\phi_{p_k}$
Call	call site c invoking callee n $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \text{MAY})$ (P_{v_s} is the path P_c of c inserted in line 13 in Figure 3)	C1 $v = \mathit{Callee2Caller}_{\text{out}}(c, n, f_{\text{out}})$ C2 Let $\mathit{MOD}(n, f_{\text{out}}) = (\mathsf{Loc}(f_{\text{out}}), \mathsf{Dep}(f_{\text{out}}), *)$ C3 **for** every $(o, C_o, *) \in \mathsf{Loc}(f_{\text{out}})$ C4 **Generate** $v_s \supseteq_{\mathit{Callee2Caller}_{\text{ctx}}(c,n,C_o) \times P_{v_s}} \{(o, \mathsf{true}, \mathsf{true})\}$ C5 **for** every $(f_{\text{in}}, C_{f_{\text{in}}}, *) \in \mathsf{Dep}(f_{\text{out}})$ C6 $w = \mathit{Callee2Caller}_{\text{in}}(c, n, f_{\text{in}})$ such that w_i reaching c C7 **Generate** $v_s \supseteq_{\mathit{Callee2Caller}_{\text{ctx}}(c,n,C_{f_{\text{in}}}) \times P_{v_s}} w_i$ C8 **Generate** $v_s \supseteq v_t$	

Table 1. Rules for resolving points-to maps $\mathsf{PtrMap}(x) = \{\mathsf{Loc}(x), \mathsf{Dep}(x)\}$ in method m for level ℓ. Each of the last five is applied once for $\mathbb{P} = \mathsf{Loc}$ and once for $\mathbb{P} = \mathsf{Dep}$.

$(\mathsf{Loc}(f^{\max}_{\text{out}}), \mathsf{Dep}(f^{\max}_{\text{out}}))$, which is already available. Then $\mathit{MOD}(m, f_{\text{out}})$ is defined to be $(\mathsf{Loc}(f^{\max}_{\text{out}}), \mathsf{Dep}(f^{\max}_{\text{out}}), C^{\text{may}}_{f_{\text{out}}})$, where $C^{\text{may}}_{f_{\text{out}}}$ is set as true if f_{out} is directly modified in method m and set otherwise as a disjunction of the context conditions in all its MAY-DEF sites, i.e., all χ operations of $f^{\max}_{\text{out}}$ in method m.

For a formal-in $f_{\text{in}} \in \mathbb{F}^m_{\text{in}}$ at level ℓ, we write f^0_{in} for its first SSA version in m. Thus, $\mathit{USE}(m, f_{\text{in}}) = (\mathsf{PtrSet}(f_{\text{in}}), C_{f_{\text{in}}})$, where $C_{f_{\text{in}}}$ is true if f^0_{in} is directly used in method m and otherwise as a disjunction of the context conditions at all MAY-USE sites, i.e., μ operations of $f^0_{f_{\text{in}}}$ in m. $\mathsf{PtrSet}(f_{\text{in}})$ is computed later by *Resolve_PointsToSets* and subsequently used in lines 41 and 48 of *Add_μ_χ_Derefs*.

In Figure 2(b), the *MOD* and *USE* functions of *foo* are given in (4).

⑤ TopDown: ***Resolve_PointsToSets*** The points-to sets of the pointers at level ℓ in method m can now be obtained by resolving all formal-ins (lines 32 and 34).

⑥ TopDown: *Add_μ_χ_Derefs* We annotate all dereferences to the pointers at level ℓ with MAY-USEs and MAY-DEFs. The points-to relations in $\mathsf{Loc}(p_i)$ are generated locally in method m and handled straightforwardly. To deal with those generated by m's callers in $\mathsf{Dep}(p_i)$ in lines 42 and 49, new context conditions are generated. If p_i points to v, because a formal-in f_{in} does, then $C_{f_{\text{in}}}$ is strengthened to include $f_{\text{in}} \to v$ to indicate the context condition under which the MAY-USE/MAY-DEF occurs (Figure 2).

SPAS soundly tracks path correlation on top of a FSCS pointer analyser.

Theorem 1. *$\mathsf{PtrSet}(p)$ contains all possible targets for p during any execution.*

Proof sketch. In a FSCS pointer analyser without considering path-sensitivity, the path/scope for a variable definition is taken as true. SPAS refines but never under-approximates it at a call site (lines 8 and 13 in Figure 3) and in Rules *Loc-Init*, *Dep-Init*, *Assn* and *Phi* (Table 1). So the soundness of SPAS follows from that of the underlying FSCS analyser, which preserves Property 1 level-wise.

The following theorem states a well-known fact about path-sensitivity.

Theorem 2. *Let $\mathsf{PtrSet}^{\mathsf{SPAS}}(p)$ ($\mathsf{PtrSet}^{\mathsf{LevPA}}(p)$) be the points-to set of p found by SPAS (a FSCS pointer analyser like LevPA). Then $\mathsf{PtrSet}^{\mathsf{SPAS}}(p) \subseteq \mathsf{PtrSet}^{\mathsf{LevPA}}(p)$.*

Proof sketch. Compared to LevPA, as argued in the proof of Theorem 1, the path condition at a variable definition site in SPAS is either the same or strengthened.

5.4 Supporting Strong Updates

In Table 1, a χ operation $v_s = \chi(v_t, C_{v_s}, P_{v_s}, \text{MAY})$ represents a MAY-DEF, where P_{v_s} safely overapproximates the scope where v_s is defined (Theorem 1). In Rules *Chi* and *Call*, $v_s \supseteq v_t$ is always used as only weak updates are allowed.

To support strong updates, we consider a χ operation, $v_s = \chi(v_t, C_{v_s}, P_{v_s}, M_{v_s})$, associated with a store "$*p_i = \dots$" residing on a path P_χ in method m. Let this χ operation be referred to as χ_{op}. In χ_{op}, $M_{v_s} \in \{\text{MAY}, \text{MAY}^+, \text{MUST}\}$. So MAY^+ is now identified as a special case of MAY introduced in Section 5.2. M_{v_s} is set as MUST when χ_{op} is a MUST-DEF. This is both context-sensitive and path-sensitive, meaning that p_i must point to v along path P_{v_s} in context C_{v_s}. However, in the other contexts, $p_i \to v$ may not hold, i.e., P_{v_s} may not be exact. M_{v_s} is MUST when $\mathsf{UniqueTarget}(m, p_i, v)$ is true, which is defined to hold when (a) p_i points to v uniquely whenever method m is invoked at a call site such that p_i points to v and (b) v is a concrete object in Singletons. Following [14], Singletons is the subset of locations in $\mathbb{O}$ with arrays, heap objects and locals inside recursion cycles being removed. M_{v_s} is set as MAY^+ or MAY when χ_{op} is a MAY-DEF, in which case, dereferencing p_i may yield more than one target. M_{v_s} is set as MAY^+ when $C_{v_s} = \mathsf{true}$ and P_{v_s} is exact, meaning that p_i must point to v whenever the program is executed along P_{v_s}. Otherwise, M_{v_s} is set as MAY, in which case, p_i may or may not point to v as described in Section 5.3.

Now, constraint $v_s \supseteq v_t$ used in Table 1 is augmented with the guards:

$$v_s \supseteq_{C_{\text{old}} \times P_{\text{old}}} v_t, \quad \text{where}(C_{\text{old}}, P_{\text{old}}) = \begin{cases} (\neg C_{v_s}, \mathsf{true}) \text{ if } M_{v_s} = \text{MUST} \\ (\mathsf{true}, P_\chi \wedge \neg P_{v_s}) \text{ if } M_{v_s} = \text{MAY}^+ \\ (\mathsf{true}, \mathsf{true}) \text{ if } M_{v_s} = \text{MAY} \end{cases} \quad (6)$$

The base version of our algorithm, shown in Figure 3, performs weak updates only as it treats MUST and MAY^+ conservatively as MAY. In our fully-fledged algorithm, SPAS obtains improved precision since *all-path strong updates*, much like Dead Code Elimination (DCE), are enabled when M_{v_s} is MUST. In this case, the old points-to relations of v_t at all incoming paths of the store $p_i = \dots$ are killed *if they are in the same context* C_{v_s}. In addition, SPAS improves analysis precision further since *some-path strong updates*, must like Partial DCE [22], are also performed when M_{v_s} is MAY^+. In this case, the old points-to relations of v_t *in any context* are killed along P_{v_s} but allowed to flow into v_s along $P_\chi \wedge \neg P_{v_s}$.

We only need to make small changes to our algorithm in Figure 3. Only the path conditions for the points-to relations of p_i established intraprocedurally in $\mathsf{Loc}(p_i)$ may be considered as being exact conservatively.

Line 8 Insert a MUST-DEF, $r = \chi(r, \mathsf{true}, P_c, \text{MUST})$, between lines 8 and 9 to handle the assignment of a return parameter in a function invoked at the call site c to a locally-defined pointer r in method m (Definition 2).

Line 11 Use $MOD(m, f_{\text{out}})$ given in Definition 5, where its fourth component $C_{\text{out}}^{\text{must}}$ is a conjunction of context conditions, one condition C_P for every possible path P from the entry to the exit of method m, such that C_P is true if f_{out} is directly modified on P and a disjunction of context conditions in all χ's representing MUST-DEFs on P otherwise.

Line 13 M_{v_s} is MUST if $\mathit{Callee2Caller}_{\text{ctx}}(c,n,C_{\text{out}}^{\text{must}})$ holds for every callee n checked in line 9 and MAY otherwise.

Line 39 M_{v_s} is MAY$^+$ if $v \in$ Singletons and p_i is not defined in a cycle in the CFG of method m (decided in *Pointer_Inference*), and MAY otherwise.

Line 42 M_{v_s} is MUST if UniqueTarget(m, p_i, v) holds and MAY otherwise. (This overwrites MAY$^+$ set for v in line 39 if UniqueTarget(m, p_i, v) holds.)

The points-to maps for f_1 and g_1 are thus refined from (5) to those in Figure 2.

Theorem 3. *With strong updates thus specified, Theorems 1 and 2 remain valid.*

Proof sketch. Follows simply from the definitions of MUST, MAY$^+$ and MAY.

6 Experimental Evaluation

We have implemented SPAS in the Open64 compiler (v4.2). We use the CUDD2.4.2 library for representing points-to relations, contexts and paths. As in [23], parameterised spaces are used to reduce analysis overhead and improve precision. We evaluate the scalability of SPAS in handling (intraprocedural) path-sensitivity by integrating it with a state-of-the-art FSCS pointer analyser, LevPA [23], which already performs all-path strong updates (for MUST-DEFs). Despite this, SPAS obtains points-to information with non-decreasing precision with improvements at stores/loads that are amenable to path-sensitive pointer analysis, at a small increase in analysis overhead. We have used all 27 C benchmarks from SPEC CPU2000 and CPU2006 and carried out our experiments on a 3.0GHz quad-core Intel Xeon system running Redhat Enterprise Linux 5 (kernel version is 2.6.18) with 16GB memory. Benchmarks `253.perlbmk` and `403.gcc` run out of memory under both analyzers and are thus excluded in further discussions.

6.1 Analysis Overhead

As shown in Table 2, SPAS uses 18.42% more time and 10.97% more memory than LevPA on average. To the best of our knowledge, SPAS is the fastest path-sensitive pointer analysis reported. Benchmarks `176.gcc` and `400.perlbench` are the most costly to analyze due to many iterations required for handling function pointers and recursion cycles. From the statistics in the last nine columns, we can see the extra analysis overhead incurred by SPAS. Column "D-Vars" gives the number of decision variables used to encode the paths in a program using BDDs, resulting in a slight increase in the total memory usage by SPAS for each benchmark (measured using the memory tracing tool available in Open64).

In the last eight columns, the analysis time for a benchmark is broken down into eight parts on computing points-to levels, generating paths (Section 5.1), and performing the six steps of SPAS in Figure 3. In 13 out of 25 benchmarks, Step 2 (*Build_SSA*) and Step 3 (*Pointer_Inference*) consume most of the analysis time in a benchmark. These are also the very steps where SPAS spends more analysis time than LevPA as it does extra work in handling program paths. For `186.crafty`, `188.ammp`, `401.bzip2` and `464.h264ref`, the analysis times under LevPA are small. SPAS adds relatively high overheads mainly in Steps 2 and 3.

Benchmark	Analysis Overhead				SPAS								
	Time (secs)		Memory (MBs)		D-Vars	Time Breakdown (secs)							
	LevPA	SPAS(%)	LevPA	SPAS(%)		Comp. Levels	Gen. Paths	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
164.gzip	0.42	9.52	20.97	9.31	269	0.09	0.02	0.07	0.22	0.02	0.00	0.04	0.00
175.vpr	1.11	12.00	55.78	9.62	639	0.34	0.03	0.09	0.50	0.15	0.01	0.08	0.04
176.gcc	1230.76	23.62	6576.05	9.91	16043	4.05	1.02	129.99	346.20	926.12	15.70	95.73	2.71
177.mesa	8.21	19.01	247.29	12.41	6242	2.84	0.25	0.48	2.90	0.83	0.11	0.23	2.13
179.art	0.08	0.00	5.28	10.51	47	0.03	0.00	0.00	0.03	0.01	0.00	0.00	0.01
181.mcf	0.13	0.00	5.74	7.52	34	0.03	0.00	0.00	0.05	0.01	0.00	0.01	0.03
183.equake	0.09	22.20	5.62	10.47	50	0.04	0.01	0.00	0.05	0.00	0.01	0.00	0.00
186.crafty	3.65	62.73	136.44	11.33	1517	0.55	0.06	0.48	3.06	1.55	0.11	0.07	0.06
188.ammp	2.28	35.53	58.94	5.93	804	0.03	0.09	1.11	1.54	0.03	0.06	0.20	0.03
197.parser	15.31	13.46	133.60	10.60	570	0.27	0.03	0.23	1.99	13.44	0.04	1.20	0.17
254.gap	21.71	12.16	440.50	4.90	7482	1.91	0.31	4.92	7.43	4.20	0.51	4.23	0.84
255.vortex	19.37	27.36	624.01	5.24	6019	1.91	0.33	4.88	8.69	3.67	0.44	4.30	0.45
256.bzip2	0.20	0.00	13.29	10.45	144	0.06	0.00	0.03	0.07	0.01	0.02	0.01	0.00
300.twolf	1.65	21.82	64.22	7.94	520	0.52	0.03	0.09	0.80	0.37	0.00	0.08	0.12
400.perlbench	971.20	24.75	4111.17	9.11	13218	2.98	0.87	105.84	277.65	680.99	13.01	125.60	4.67
401.bzip2	0.79	36.71	24.52	16.68	530	0.17	0.02	0.03	0.66	0.07	0.01	0.01	0.11
429.mcf	0.11	18.18	4.95	24.45	37	0.03	0.00	0.00	0.03	0.03	0.00	0.00	0.04
433.milc	0.87	0.00	45.05	10.23	469	0.32	0.02	0.17	0.19	0.08	0.01	0.04	0.04
445.gobmk	14.66	12.21	682.00	16.64	3680	1.45	0.23	3.14	5.83	2.95	0.22	2.26	0.37
456.hmmer	2.71	18.15	45.97	14.00	1673	0.86	0.05	0.12	1.30	0.50	0.03	0.10	0.86
458.sjeng	1.39	21.58	55.78	11.93	1060	0.27	0.06	0.24	0.65	0.34	0.01	0.10	0.02
462.libquantum	0.32	12.50	0.00	10.41	141	0.07	0.02	0.10	0.11	0.03	0.00	0.02	0.01
464.h264ref	5.77	39.86	247.29	11.06	2457	1.44	0.16	0.80	2.97	1.37	0.16	0.47	0.70
470.lbm	0.07	14.29	5.28	11.52	19	0.04	0.00	0.01	0.02	0.00	0.00	0.00	0.01
482.sphinx3	1.76	2.84	5.74	11.98	835	0.53	0.07	0.13	0.65	0.16	0.02	0.08	0.17

Table 2. Percentage increases of analysis overhead under SPAS w.r.t. LevPA.

6.2 Path-Sensitive Precision

Table 3 shows that SPAS can obtain more precise points-to sets than LevPA at certain loads/stores in most benchmarks. We consider only the loads/stores that reside beyond the first branch in the CFG of a method after all its SCCs (strongly connected components) have been collapsed. The pointers accessed indirectly at their associated χ and μ operations (MAY/MUST-DEFs and MAY-USEs) are the ones whose points-to information may be improved by SPAS.

We measure the number of χ's and μ's with improved points-to information in two ways, indicated by their *Traditional* and *Path-Sensitive* columns. Note that by Theorem 2, $\mathsf{PtrSet}(p)^{\mathsf{SPAS}} \subseteq \mathsf{PtrSet}(p)^{\mathsf{LevPA}}$ holds for any pointer p. With the traditional metric, the points-to set of p is said to be *more precise* under SPAS if $|\mathsf{PtrSet}(p)^{\mathsf{SPAS}}| < |\mathsf{PtrSet}(p)^{\mathsf{LevPA}}|$. With the path-sensitive metric, the path, i.e., the scope information governing each points-to target is also taken into account. For LevPA, the path guarding a χ or μ operation is always true. Thus, the points-to set of p is *more precise* under SPAS if either $|\mathsf{PtrSet}(p)^{\mathsf{SPAS}}| < |\mathsf{PtrSet}(p)^{\mathsf{LevPA}}|$ or the guarding path for a χ or μ operation is not true (i.e., more restricted).

As shown in Table 3, SPAS has improved points-to information in most benchmarks. Under "Traditional" for χ's, the percentage improvements range from 0% to 6.71% with an average of 2.61%. Under "Traditional" for μ's, the percentages are within 0 to 7.08% with an average of 2.49%. Under "Path-Sensitive", the improvements are more significant with an average of 42.38% for χ's and 41.07% for μ's. These results should be understood with some caveats. First, what SPAS is compared with is a state-of-the-art FSCS pointer analyser that already performs all-path strong updates. Second, SPAS obtains such improved path-sensitive pre-

Benchmark	χ's (MAY/MUST-DEFs at Stores)					μ's (MAY-USEs at Loads)				
	LevPA (Total)	SPAS (More Precise)				LevPA (Total)	SPAS (More Precise)			
		Traditional		Path-Sensitive			Traditional		Path-Sensitive	
		Total	%	Total	%		Total	%	Total	%
164.gzip	144	6	4.17	115	79.86	165	4	2.42	128	77.58
175.vpr	232	6	2.59	135	58.19	184	7	3.80	106	57.61
176.gcc	3710	132	3.56	1356	36.55	11843	506	4.27	4701	39.69
177.mesa	3780	16	0.42	1101	29.13	5200	60	1.15	1375	26.44
179.art	6	0	0.00	6	100.00	6	0	0.00	6	100.00
181.mcf	76	0	0.00	26	34.21	144	0	0.00	65	45.14
183.equake	29	0	0.00	2	6.90	55	0	0.00	5	9.09
186.crafty	343	23	6.71	291	84.84	1007	62	6.16	860	85.40
188.ammp	475	20	4.21	418	88.00	694	46	6.63	615	88.62
197.parser	374	14	3.74	296	79.14	403	19	4.71	260	64.52
254.gap	297	6	2.02	188	63.30	5466	32	0.59	3311	60.57
255.vortex	801	11	1.37	120	14.98	3651	36	0.99	469	12.85
256.bzip2	51	0	0.00	13	25.49	109	0	0.00	33	30.28
300.twolf	106	3	2.83	33	31.13	265	17	6.42	111	41.89
400.perlbench	2938	146	4.97	1045	35.57	17084	641	3.75	7027	41.13
401.bzip2	601	13	2.16	96	15.97	1380	11	0.80	184	13.33
429.mcf	77	3	3.90	29	37.66	162	2	1.23	45	27.78
433.milc	153	0	0.00	29	18.95	287	0	0.00	34	11.85
445.gobmk	1123	58	5.16	382	34.02	2957	139	4.70	1795	60.70
456.hmmer	568	37	6.51	305	53.70	1680	119	7.08	1239	73.75
458.sjeng	343	13	3.79	268	78.13	748	32	4.28	706	94.39
462.libquantum	5	0	0.00	0	0.00	7	0	0.00	0	0.00
464.h264ref	1994	68	3.41	477	23.92	7654	121	1.58	1522	19.89
470.lbm	8	0	0.00	0	0.00	13	0	0.00	0	0.00
482.sphinx3	220	8	3.64	66	30.00	782	13	1.66	208	26.60

Table 3. Percentages of variables at χ's and μ's with more accurate points-to sets.

cision at small analysis overhead. Finally, such improvement can be critical for some client applications (e.g., bug detection).

Let us look at some benchmarks in detail. In the case of `164.gzip`, `176.gcc` `197.parser`, `400.perlbench`, `429.mcf`,`464.h264ref`, `482.sphinx3`, `458.sjeng` the improvements are mostly over 3% for both χ's and μ's. Path-sensitive analysis provides little benefits for seven benchmarks: `179.art`,`181.mcf`,`183.equake`, `256.bzip2`, `433.milc`, `462.libquantum` and `470.lbm`. They are small programs with few pointers but mostly scalar-to-array assignments. For other benchmarks on scientific computations, such as `177.mesa`, and `255.vortex` , the improvements are below 2%. In contrast, benchmarks such as `186.crafty`, `445.gobmk` and `456.hmmer` exhibit much better precision improvements. They each have a relatively high number of decision variables, giving rise to more opportunities for capturing path correlation.

7 Conclusion

We have presented SPAS, a path-sensitive pointer analysis that extends a recent flow- and context-sensitive pointer analysis LevPA. Our experimental evaluation shows that SPAS incurs reasonable analysis overhead over LevPA (on average 18.42% increase in analysis time and 10.97% increase in memory usage) and computes more precise points-to information. Our results are expected to provide insights for developing client-driven pointer analysis techniques.

Acknowledgement

This work is supported the Australian Research Council Grant DP0987236.

References

1. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. *PLDI'03*, 38(5):114, 2003.
2. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE'00*, 30:775–802, June 2000.
3. R. Chatterjee, B.G. Ryder, and W.A. Landi. Relevant context inference. In *POPL'99*, page 146. ACM, 1999.
4. F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC'96*, pages 253–267, 1996.
5. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS'91*, 13(4):490, 1991.
6. I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI'08*, pages 270–280. ACM, 2008.
7. M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI'94*, pages 242–256, 1994.
8. T. Gutzmann, J. Lundberg, and W. Lowe. Towards path-sensitive points-to analysis. In *SCAM'07*, pages 59–68. IEEE, 2007.
9. B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL'09*, pages 226–238. ACM, 2009.
10. B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO'11*, 2011.
11. V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI'08*, pages 249–259. ACM, 2008.
12. William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. *PLDI'92*, 27(7):235–248, 1992.
13. Wei Le and Mary Lou Soffa. Refining buffer overflow detection via demand-driven path-sensitive analysis. In *PASTE'07*, pages 63–68. ACM, 2007.
14. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL'11*, pages 3–16, New York, NY, USA, 2011. ACM.
15. V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *FSE'03*, pages 317–326, 2003.
16. E.M. Nystrom, H.S. Kim, and W.W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. *SAS'04*, pages 165–180, 2004.
17. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL'11*, pages 17–30. ACM, 2011.
18. B. Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM New York, NY, USA, 1996.
19. J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*, pages 131–144. ACM, 2004.
20. Y. Xie and A. Aiken. Context-and path-sensitive memory leak detection. *FSE'05*, 30(5):125, 2005.
21. Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. *POPL'05*, 40(1):351–363, 2005.
22. Jingling Xue, Qiong Cai, and Lin Gao. Partial dead code elimination on predicated code regions. *SPE'06*, 36:1655–1685, 2006.
23. H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO'10*, pages 218–229. ACM, 2010.
24. J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI'04*, page 157. ACM, 2004.