Value-Flow-Based Demand-Driven Pointer Analysis for C and C++

Yulei Sui and Jingling Xue

Abstract—We present SUPA, a value-flow-based demand-driven flow- and context-sensitive pointer analysis with strong updates for C and C++ programs. SUPA enables computing points-to information via value-flow refinement, in environments with small time and memory budgets. We formulate SUPA by solving a graph-reachability problem on an inter-procedural value-flow graph representing a program's def-use chains, which are pre-computed efficiently but over-approximately. To answer a client query (a request for a variable's points-to set), SUPA reasons about the flow of values along the pre-computed def-use chains sparsely (rather than across all program points), by performing only the work necessary for the query (rather than analyzing the whole program). In particular, strong updates are performed to filter out spurious def-use chains through value-flow refinement as long as the total budget is not exhausted.

We have implemented SUPA on top of LLVM (4.0.0) together with a comprehensive micro-benchmark suite after a years-long effort (consisting of around 400 test cases, including hand-written ones and the ones extracted from real programs). We have evaluated SUPA by choosing uninitialized pointer detection and C++ virtual table resolution as two major clients, using 24 real-world programs including 18 open-source C programs and 6 large CPU2000/2006 C++ benchmarks. For uninitialized pointer client, SUPA achieves improved precision as the analysis budget increases, with its flow-sensitive (context-insensitive) analysis reaching 97.4% of that achieved by whole-program Sparse Flow-Sensitive analysis (SFS) by consuming about 0.18 seconds and 65KB of memory per query, on average (with a budget of at most 10000 value-flow edges per query). With context-sensitivity also considered, SUPA becomes more precise for some programs but also incurs more analysis times. To further demonstrate the effectiveness of SUPA, we have also evaluated SUPA in resolving C++ virtual tables by querying the function pointers at every virtual callsite. Compared to analysis without strong updates for heap objects, SUPA's demand-driven context-sensitive strong update analysis reduces 7.35% spurious virtual table targets with only 0.4 secs per query, on average.

Index Terms—strong updates, value flow, pointer analysis, flow sensitivity

1 INTRODUCTION

As one of the most fundamental static program analyses, pointer analysis serves as an important enabling technology for many clients, such as change impact analysis [1], bug detection [51, 59, 60], security analysis [4, 5], enforcing control-flow integrity [7, 15], compiler optimization [47, 50, 52], symbolic execution [8, 54] and type state analysis [17, 58].

The goal of pointer analysis is to compute an approximation of the set of abstract objects that a pointer can refer to. Flow-sensitivity is one of the key dimensions of precision in pointer analysis, especially for analyzing C and C++ programs. A pointer analysis is *flow-sensitive* if it respects control flow while a *flow-insensitive* one ignores the program execution order.

An important factor in flow-sensitive pointer analysis is *strong updates*, where stores overwrite, i.e., kill the previous contents of their abstract destination objects with new values [19, 29]. In the case of *weak updates*, these objects are assumed conservatively to also retain their old contents. A flow-sensitive analysis can strongly update an abstract object written at a store if and only if that object has exactly one concrete memory address, known as a singleton.

When augmented with (1) *field-sensitivity* which distinguishes fields of an aggregate object and (2) context-sensitivity which distinguishes different calling contexts of a method, flow-sensitive analysis can identify more singletons to perform strong updates on field and heap objects to further boots its precision.

This paper introduces SUPA, a query-based demand-driven pointer analysis for C and C++ by investigating how to perform strong updates effectively in a field-, flow- and context-sensitive analysis framework. By applying strong updates where needed based on clients' needs, SUPA can improve precision for pointer analysis without strong updates in both C and C++ programs.

Background and Challenges. Ideally, strong updates at stores should be performed to analyze all paths independently by solving a *meet-over-all-paths* (MOP) problem. However, even with branch conditions being ignored, this problem is intractable due to potentially unbounded number of paths that must be analyzed [26, 35].

Instead, traditional flow-sensitive pointer analysis (FS) for C [22, 24] computes the maximal-fixed-point solution (MFP) as an over-approximation of MOP by solving an iterative data-flow problem. Thus, the data-flow facts that reach a confluence point along different paths are merged. Improving on this, sparse flow-sensitive pointer analysis (SFS) [20, 30, 33, 61, 62] boosts the performance of FS in analyzing large C programs while maintaining the same strong updates done by FS. The basic idea is to first conduct a pre-analysis on the program to over-approximate its def-use chains and then perform FS by propagating the data-flow facts, i.e., points-to information sparsely along only the pre-computed def-use chains (aka value-flows) instead of all program points in the program's control-flow graph (CFG).

Recently, an approach [29] for performing strong updates in C programs is introduced. It sacrifices the precision of FS

- *Yulei Sui is with Centre for Artificial Intelligence (CAI) at University of Technology Sydney (UTS) and Jingling Xue is with University of New South Wales (UNSW).*
  E-mail: yulei.sui@uts.edu.au, jingling@cse.unsw.edu.au
- *This work is supported by Australia Research Council grants, DE170101081, DP170103956 and DP180104069.*

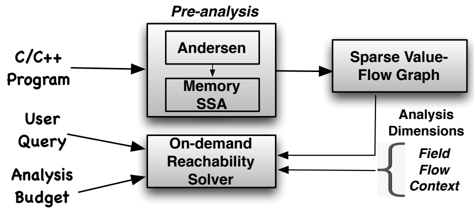


Fig. 1: SUPA

to gain efficiency by applying strong updates at stores where flow-sensitive singleton points-to sets are available but falls back to the flow-insensitive points-to information otherwise.

By nature, the challenge of pointer analysis is to make judicious tradeoffs between efficiency and precision. Almost all of the prior pointer analyses for C that consider some degree of flow-sensitivity are whole-program analyses. Precise ones are unscalable since they must typically consider both flow- and context-sensitivity (FSCS) in order to maximize the number of strong updates performed. In contrast, faster ones like [29] are less precise, due to both missing strong updates and propagating the points-to information flow-insensitively across the weakly-updated locations.

Insights. In practice, a client application of a pointer analysis may require only parts of the program to be analyzed, e.g., indirect call resolution for enforcing control-flow integrity [15]. In addition, some points-to queries may demand precise answers while others can be answered as precisely as possible with small time and memory budgets [18]. In all these cases, performing strong updates blindly across the entire program is cost-ineffective in achieving precision.

How do we develop precise and efficient pointer analyses that are focused and partial, paying closer attention to the parts of the programs relevant to on-demand queries? Most of the existing demand-driven analyses for C [21, 63, 66] and Java [31, 39, 41, 43, 57] are flow-insensitive and thus cannot perform strong updates to produce the precision needed by some clients. Elsewhere, advances in whole-program flow-sensitive analysis for C have exploited some form of sparsity to improve performance [20, 30, 33, 61, 62]. However, how to replicate this success for demand-driven flow-sensitive analysis is unclear.

Moreover, flow-sensitive analysis for C++, the OO incarnation of C, is less studied due to its complicated language features (i.e., imperative + low-level object-orientation). Compared to C, heap allocations are more frequently used in C++ with more indirect calls in the form of polymorphic virtual invocations. The extensive usage of C++ templates and STL containers cause compilers to generate a huge amount of low-level code (e.g., LLVM IR), which significantly complicates whole-program analysis. This makes demand-driven analysis more compelling if it can analyze the necessary program parts driven by client queries under tight analysis budgets.

Finally, it remains open as to whether sparse strong update analysis can be performed both flow- and context-sensitively on-demand for both C and C++ programs.

Our Solution. As illustrated in Figure 1, this paper introduces SUPA, a new demand-driven pointer analysis with strong updates for C/C++, designed to support flexible yet effective tradeoffs between efficiency and precision in answering client queries, in environments with user-defined time and memory budgets. The novelty behind SUPA lies in performing **S**trong **UP**date **A**nalysis precisely by refining imprecisely pre-computed value-flows away under user-defined budges (i.e., value-flows traversed). Given a points-to query, strong updates are performed by solving a graph-reachability problem on an inter-procedural value-flow graph that captures the def-use chains of the program obtained conservatively by a pre-analysis. Such over-approximated value-flows can be obtained by applying Andersen's analysis [3] (flow- and context-insensitively). SUPA conducts its reachability analysis on-demand sparsely along only the pre-computed value-flows rather than control-flows. In addition, SUPA filters out imprecise value-flows by performing strong updates field-, flow- and context-sensitively where needed with no loss of precision as long as the total analysis budget is sufficient. The precision of SUPA depends on the degree of value-flow refinement performed under a budget. The more spurious value-flows SUPA removes, the more precise the points-to facts are. This paper makes the following key contributions:

- We present SUPA, a demand-driven field-, flow- and context-sensitive pointer analysis that enables computing precise points-to information by refining away imprecisely precomputed value-flows. SUPA supports on-demand strong updates of field and heap objects for analyzing both C and C++ programs by facilitating efficiency and precision tradeoffs with user-defined analysis budgets.
- We have also released a comprehensive micro benchmark suite PTABEN for pointer analysis for C/C++ after a years-long effort. It consists of around 400 well-designed small tests, including hand-written ones and the ones extracted from real programs to cover as many analysis aspects as possible. PTABEN provides flexible and extendable interfaces for users to add their own tests for validating the correctness of different pointer analysis implementations.
- We evaluate SUPA with uninitialized pointer detection as a practical client by using a total of 18 open-source C programs. As the analysis budget increases, SUPA achieves improved precision, with its single-stage flow-sensitive analysis reaching 97.4% of that achieved by whole-program flow-sensitive analysis, by consuming about 0.18 seconds and 65KB of memory per query, on average (with a per-query budget of at most 10000 value-flow edges traversed). With context-sensitivity also being considered, more strong updates are observed at the expense of more analysis times.
- We also evaluate the effectiveness of SUPA in analyzing C++ programs with virtual table resolution as another major client by using 6 large SPEC2000/2006 C++ programs. Compared to pointer analysis without strong updates, SUPA's context-sensitive strong updates significantly reduces 7.35% spurious virtual table targets with only 0.4 secs per query, on average.

2 Program Representation

We describe how to represent a C/C++ program by a sparse value-flow graph to enable demand-driven pointer analysis via value-flow refinement. Section 2.1 introduces the part of LLVM-IR relevant to pointer analysis. Section 2.2 describes

TABLE 1: Domains and LLVM instructions used by our pointer analysis.

Analysis Domains			LLVM-like Instruction Set	
ℓ	$\in \mathcal{L}$	instruction labels	ADDROF	$p \ = \&o$
cnt, fld	$\in \mathcal{C}$	constants	COPY	$p \ = q$
s, idx	$\in \mathcal{S}$	stack virtual registers	PHI	$p \ = \phi(q, r)$
g	$\in \mathcal{G}$	global variables	FIELD	$p \ = \&q \rightarrow fld$
f	$\in \mathcal{F} \subseteq \mathcal{G}$	program functions	LOAD	$p \ = *q$
p, q, r, x, y	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	top-level variables	STORE	$*p = q$
o, a, b, c, d	$\in \mathcal{O}$	address-taken variables	CALL	$p \ = q(r_1, \ldots, r_n)$
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	program variables	FUNENTRY	$f(r_1, \ldots, r_n)$
			FUNEXIT	$ret_f \ p$

how to put top-level and address-taken variables in SSA form to construct a sparse value-flow graph that represents the def-use chains of a program.

2.1 LLVM-IR

We perform pointer analysis in the LLVM-IR of a program, as in [6, 20, 29, 30, 46, 61]. The domains and the LLVM instructions relevant to pointer analysis are given in Table 1. The set of all variables $\mathcal{V}$ are separated into two subsets, $\mathcal{O}$ that contains all possible abstract objects, i.e., *address-taken variables* of a pointer and $\mathcal{P}$ that contains all *top-level variables*.

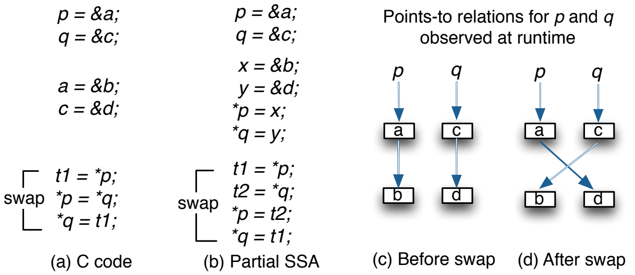
In LLVM-IR, top-level variables in $\mathcal{P} = \mathcal{S} \cup \mathcal{G}$, including stack virtual registers (symbols starting with ”%”) and global variables (symbols starting with ”@”) are explicit, i.e., directly accessed. Address-taken variables in $\mathcal{O}$ are implicit, i.e., accessed indirectly at LLVM's load or store instructions via top-level variables.

Only a subset of the complete LLVM instruction set that is relevant to pointer analysis are modeled. As in Table 1, every function f of a program contains nine types of instructions (statements), including seven types of instructions used in the function body of f, and one FUNENTRY instruction $f(r_1, \ldots, r_n)$ with the declarations of the parameters of f, and one FUNEXIT instruction $ret_f \ p$ as the unique return of f. Note that the LLVM pass UnifyFunctionExitNodes is executed before pointer analysis in order to ensure that every function has only one FUNEXIT instruction.

Let us go through the seven types of instructions used inside a function. For an ADDROF instruction $p{=}\&o$, known as an *allocation site*, o is one of the following objects: (1) a stack object, o_ℓ, where ℓ is its allocation site (via an LLVM alloca instruction), (2) a global object, i.e., a global object o_ℓ, where ℓ is its allocation site or a program function o_f, where f is its name, and (3) a dynamically created heap object o_ℓ^h, where ℓ is its heap allocation site (e.g., via a malloc() call). For flow-sensitive pointer analysis, the initializations for global objects take place at the entry of main().

Our handling of field-sensitivity is ANSI-compliant [23]. To model the field accesses of a struct object, FIELD represents a getelementptr instruction with its field offset fld as a constant value. SUPA uses a field-index-based approach to field-sensitivity similar to [6, 20, 34]. The fields of a struct object are distinguished by their unique indices.

Unlike [34] which treats the first field as the whole struct object. Our modeling brings additional precision by distinguishing the first field (index 0) from the whole struct object. For every struct allocation $p = \&o$, we create a field-insensitive object o to represent the entire struct object.



```
  p = &a;
  q = &c;


  a = &b;
  c = &d;


      t1 = *p;
swap  *p = *q;
      *q = t1;
```

(a) C code

```
      p = &a;
      q = &c;

      x = &b;
      y = &d;
      *p = x;
      *q = y;

      t1 = *p;
swap  t2 = *q;
      *p = t2;
      *q = t1;
```

(b) Partial SSA

(c) Before swap (d) After swap

Fig. 2: A swap example and its partial SSA form.

A field object denoted by o_{fld} is derived from o when analyzing FIELD $q{=}\&p{\rightarrow}fld$, where fld can also be the first field. Thus, different fields (including index 0) are modeled using distinct (sub) objects. Two pointer dereferences are aliased if one refers to o and another one refers to one of its fields e.g., o_{fld} since it is the sub component of o. However, dereferences refer to different fields (e.g., o_{fld0} and o_{fld1}) of o are distinguished and not aliased.

For a pointer arithmetic. e.g., $q = p + i$, if p points to a struct object, we conservatively assume that q can point any field of this struct object (i.e., the field-insensitive object). This is based on the assumption that i is not across the boundary of the object. A pointer arithmetic used for accessing an aggregate object out of the boundary may cause unsoundness in our analysis. Similar to previous pointer analysis for C, our handling of pointer to integer casting is also unsound. Arrays are treated monolithically, i.e., accessing any element of an array is treated as accessing the entire array object.

COPY denotes either a casting instruction or a value assignment from a PHI instruction in LLVM. PHI is a standard SSA instruction introduced at a confluence point in the CFG to select the value of a variable from different control-flow branches. An LLVM PHI, e.g., $p = PHI(q, r)$ is translated into two COPY instructions $p{=}q$ and $p{=}r$.

LOAD (STORE) is a memory accessing instruction that reads (write) a value from (into) an address-taken object.

CALL, $p = q(r_1, \ldots, r_n)$, denotes a call instruction, where q can be either a global variable (for a direct call) or a stack virtual register (for an indirect call).

LLVM-IR is known as a partial SSA form since only top-level variables are explicit put in SSA form by using a standard SSA construction algorithm [12] (with PHI instructions inserted at confluence points). However, address-taken variables are accessed indirectly at loads and stores via top-level variables and thus not in SSA form.

Figure 2 illustrates LLVM's partial SSA form by using a simple swap example. Figure 2(a) gives a swap program in C and Figure 2(b) shows its corresponding partial SSA form. Figures 2(c) and (d) depict some (runtime) points-to relations before and after the swap operation. In this example, we have $p, q, x, y, t1, t2 \in \mathcal{P}$ and $a, b, c, d \in \mathcal{O}$. Note that $x, y, t1$ and $t2$ are new temporary registers introduced in order to put the program given in Figure 2(a) into the partial SSA form given in Figure 2(b). In particular, $*p = *q$ is decomposed into $t2 = *q$ and $*p = t2$, where $t2$ is a top-level pointer.

2.2 Sparse Value-Flow Graph

Given a program in partial SSA form, a *sparse value-flow graph* (SVFG) $G = (N, E)$ is a multi-edged directed graph that captures its def-use chains conservatively. N is the set of nodes representing all statements and E is the set of edges representing all potential def-use chains. In particular, an edge $\ell_1 \xrightarrow{v} \ell_2$, where $v \in \mathcal{V}$, from statement ℓ_1 to statement ℓ_2 signifies a potential def-use chain for v with its def at ℓ_1 and use at ℓ_2. This representation is sparse since the intermediate program points between ℓ_1 and ℓ_2 are omitted.

As top-level variables are in SSA form, their uses have unique definitions (with ϕ functions inserted at confluence points as is standard). A def-use chain $\ell_1 \xrightarrow{t} \ell_2$, where $t \in \mathcal{P}$, represents a *direct value-flow* of t. Such def-use chains can be found easily without requiring pointer analysis.

As address-taken variables are not (yet) in SSA form, their indirect uses at loads may be defined indirectly at multiple stores. We build their def-use chains in several steps by following [20, 48], with an illustrating example given in Section 3. First, the points-to information in the program is computed by a pre-analysis. Second, a load $p = *q$ is annotated with a function $\mu(a)$ for each variable $a \in \mathcal{O}$ that may be pointed to by q to represent a potential use of a at the load. Similarly, a store $*p = q$ is annotated with a function $a = \chi(a)$ for each variable $a \in \mathcal{O}$ that may be pointed to by p to represent a potential def and use of a at the store. If a can be strongly updated, then a receives whatever q points to and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a weak update to a. Third, we convert all the address-taken variables into SSA form, with each $\mu(a)$ treated as a use of a and each $a = \chi(a)$ as both a def and use of a. Finally, we obtain the indirect def-use chains for an address-taken variable $a \in \mathcal{O}$ as follows. For a use of a identified as a_n (with its version identified by n) at a load or store ℓ, its unique definition in SSA form is a_n at a store ℓ'. Then, an indirect def-use chain $\ell' \xrightarrow{a} \ell$ is added to represent potentially the *indirect value-flow* of a from ℓ' to ℓ. Note that the ϕ functions introduced for address-taken variables will now be ignored as the value a that appears in $\ell' \xrightarrow{a} \ell$ is not versioned. The interprocedural value-flows across procedures are computed based the mod-ref analysis on top of the memory SSA form, as described in Section 4.1. The code for SVFG construction is publicly available at https://github.com/SVF-tools/SVF.

Our demand-driven pointer analysis, SUPA, operates on the SVFG of a program. It computes points-to queries on-demand by performing strong updates, whenever possible, to refine away imprecise value-flows in the SVFG.

3 A MOTIVATING EXAMPLE

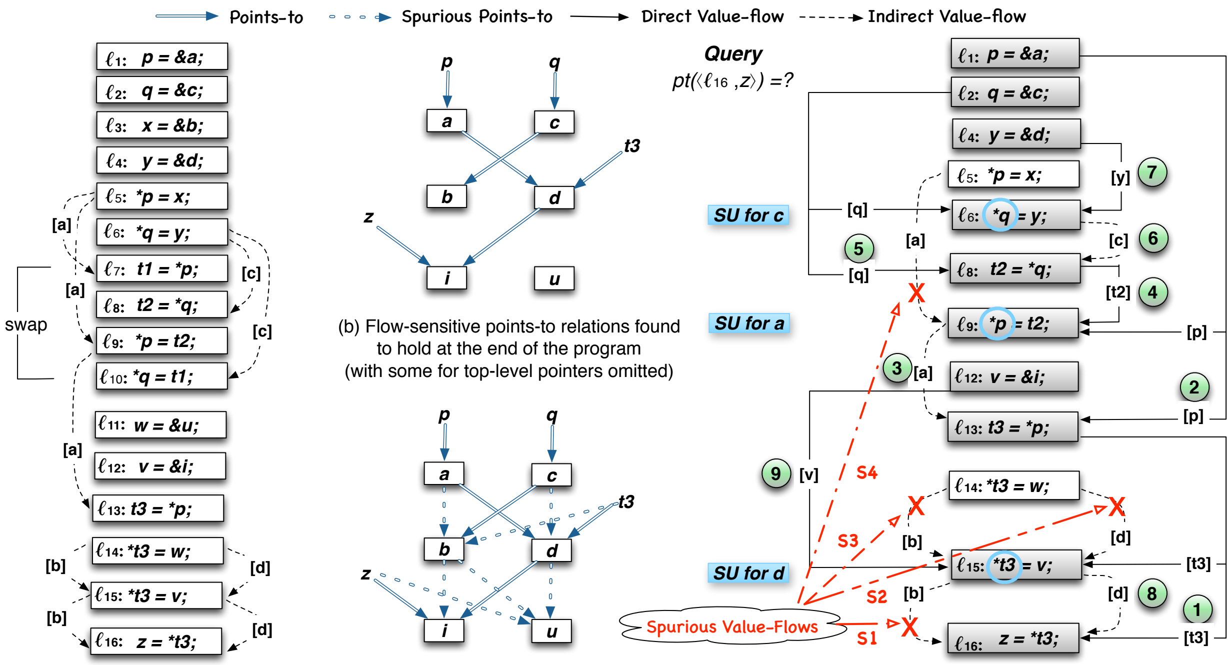
Our example program, shown in Figure 3(a), is simple (even with 16 lines). The program consists of a straight-line sequence of code, with $\ell_1 - \ell_{10}$ taken directly from Figure 2(b) and the six new statements $\ell_{11} - \ell_{16}$ added in order to highlight some key properties of SUPA. We assume that u at ℓ_{11} is uninitialized but i at ℓ_{12} is initialized. The SVFG embedded in Figure 3(a) will be referred to shortly below. We describe how SUPA can be used to prove that z at ℓ_{16} points only to the initialized object i, by computing flow-sensitively on-demand the points-to query $pt(\langle \ell_{16}, z \rangle)$, i.e., the points-to set of z at the program point after ℓ_{16}, which is defined in (1) in Section 4.

Figure 3(b) depicts the points-to relations for the six address-taken variables and some top-level ones found at the *end* of the code sequence by a whole-program flow-sensitive analysis (with strong updates) like SFS [20]. Due to flow-sensitivity, multiple solutions for a pointer are maintained. In this example, these are the true relations observed at the end of program execution. Note that SFS gives rise to Figure 2(c) by analyzing $\ell_1 - \ell_6$, Figure 2(d) by analyzing also $\ell_7 - \ell_{10}$, and finally, Figure 3(b) by analyzing $\ell_{11} - \ell_{16}$ further. As z points to i but not u, no warning is issued for z, implying that z is regarded as being properly initialized.

Figure 3(c) shows how the points-to relations in Figure 3(b) are over-approximated flow-insensitively by applying Andersen's analysis [3]. In this case, a single solution is computed conservatively for the entire program. Due to the lack of strong updates in analyzing the two stores performed by swap, the points-to relations in Figures 2(c) and 2(d) are merged, causing $*a$ and $*c$ to become spurious aliases. Thus, a points to b and c points to d are spurious. When $\ell_{11} - \ell_{16}$ are analyzed, the remaining five out of seven spurious points-to relations (shown in dashed arrows in Figure 3(c)) are introduced. Since z points to i (correctly) and u (spuriously), a false alarm for z will be issued. Failing to consider flow-sensitivity, Andersen's analysis is not precise for this uninitialization pointer detection client.

Let us now explain how SUPA works to tackle the imprecision. SUPA first performs a pre-analysis to the example program to build the SVFG given in Figure 3(a), as discussed in Section 2. For its top-level variables, their direct value-flows, i.e., def-use chains are explicit and thus omitted to avoid cluttering. For example, q has three def-use chains $\ell_2 \xrightarrow{q} \ell_6$, $\ell_2 \xrightarrow{q} \ell_8$ and $\ell_2 \xrightarrow{q} \ell_{10}$. For its address-taken variables, there are nine indirect value-flows, i.e., def-use chains depicted in Figure 3(a). Let us see how the two def-use chains for b are created. As $t3$ points to b, ℓ_{14}, ℓ_{15} and ℓ_{16} will be annotated with $b = \chi(b)$, $b = \chi(b)$ and $\mu(b)$, respectively. By putting b in SSA form, these three functions become $b2 = \chi(b1)$, $b3 = \chi(b2)$ and $\mu(b3)$. Hence, we have $\ell_{14} \xrightarrow{b} \ell_{15}$ and $\ell_{15} \xrightarrow{b} \ell_{16}$, indicating b at ℓ_{16} has two potential definitions, with the one at ℓ_{15} overwriting the one at ℓ_{14}. The def-use chains for d and a are built similarly.

Figure 3(d) shows how SUPA computes $pt(\langle \ell_{16}, z \rangle)$ on-demand, starting from ℓ_{16}, by performing a backward reachability analysis on the SVFG, with the visiting order of def-use chains marked as ① – ⑨. Formally, this is done as illustrated in Figure 7. The def-use chains for only the relevant top-level variables are shown, with strong updates



(b) Flow-sensitive points-to relations found to hold at the end of the program (with some for top-level pointers omitted)

(a) A program and its SVFG (with only indirect value-flows shown)

(c) Flow-insensitive points-to relations (with some for top-level pointers omitted)

(d) The **SUPA** analysis for resolving $pt(\langle \ell_{16}, z\rangle) = \{i\}$ by traversing from $\langle \ell_{16}, z\rangle$ backwards against the value-flows

Kind of spurious value-flows	Insight for refinement	Example
(1) store - - [o] - ✗ → load	Two dereferences are not flow-aliases with **o**	S1 ℓ_{16} - - [b] - → ℓ_{15}
(2) store - - [o] - ✗ → store	Strong updates for **o**	S2 ℓ_{15} - - [d] - → ℓ_{14} S4 ℓ_9 - [a] - → ℓ_5
	Two dereferences are not flow-aliases with **o**	S3 ℓ_{15} - [b] - → ℓ_{14}

(e) Three scenarios for refining two kinds of spurious value-flows and their corresponding examples in (d)

Fig. 3: A motivating example for illustrating SUPA (SU stands for "Strong Update").

highlighted at line ℓ_6, ℓ_9 and ℓ_{15}. By filtering out the four spurious value-flows S1 to S4 (marked by ×), SUPA finds that only i at ℓ_{12} is backward reachable from z at ℓ_{16}. Thus, $pt(\langle \ell_{16}, z\rangle) = \{i\}$. So z is proved to be initialized.

Figure 3(e) summarizes the insights of two kinds of spurious value-flows, the three scenarios under which the spurious value-flows should be refined and their corresponding examples in Figure 3(d). (1) Store-load case. A value-flow of object o connecting from a store to a load is spurious if the two dereferences at the load and the store are not flow-sensitive aliases with o. (2) Store-store case. It has two scenarios, one is similar to the store-load case and another one is identified due to the strong updates at the second store (destination node) that kills its old values defined at the first store. Note that strong updates can also help get more precise flow-sensitive aliasing results, e.g., the strong updates for a at line ℓ_9 cause that $t3$ points to d only, resulting in more precise aliases for ℓ_{15} and ℓ_{16}.

SUPA differs from prior work in the following three major aspects:

- ***On-Demand Strong Updates***
  A whole-program flow-sensitive analysis like SFS [20] can answer $pt(\langle \ell_{16}, z\rangle)$ precisely but must accomplish this task by analyzing all the 16 statements, resulting in a total of six strong updates performed at the six stores, with some strong updates performed unnecessarily for this query. Unfortunately, existing whole-program FSCS or even just FS algorithms do not scale well for large C programs [1].
  In contrast, SUPA computes $pt(\langle \ell_{16}, z\rangle)$ precisely by performing only three strong updates at ℓ_6, ℓ_9 and ℓ_{15}. The earlier a strong update is performed by SUPA during its reachability analysis, the fewer the number of statements traversed. After ① – ⑧ have been performed, SUPA finds that $t3$ points to d only. With a strong update performed at $\ell_{15} : *t3 = v$ (⑨), SUPA concludes that $pt(\langle \ell_{16}, z\rangle) = \{i\}$.
- ***Value-Flow Refinement***
  Demand-driven pointer analyses [39, 41, 57, 63, 66] are flow-insensitive and thus suffer from the same imprecision as their flow-insensitive whole-program counterparts. In the absence of strong updates, many spurious aliases (such as $*a$ and $*c$) result, causing z to point to both i and u. As a result, a false alarm for z is issued, as discussed earlier.
  However, SUPA performs strong updates flow-sensitively by filtering out the four spurious pre-computed value-flows marked by ×. As $t3$ points to d only, $\ell_{15} \xrightarrow{b} \ell_{16}$ is spurious and not traversed. In addition, a strong update is enabled at $\ell_{15} : *t3 = v$, rendering $\ell_{14} \xrightarrow{b} \ell_{15}$ and $\ell_{14} \xrightarrow{d} \ell_{15}$ spurious. Finally, $\ell_5 \xrightarrow{a} \ell_9$ is refined away due to another strong update performed at ℓ_9. Thus, SUPA has avoided many spurious aliases (e.g., $*a$ and $*c$) introduced flow-insensitively by pre-analysis, resulting in $pt(\langle \ell_{16}, z\rangle) = \{i\}$ precisely. Thus, no warning for z is issued.
- ***Query-based Precision Control***
  To balance efficiency and precision, SUPA operates in

a hybrid multi-stage analysis strategy. When asked to answer the query $pt(\langle \ell_{16}, z \rangle)$ under a budget, say, a maximum sequence of three steps traversed, SUPA will stop its traversal from ℓ_9 to ℓ_8 (at ④) in Figure 3(d) and fall back to the pre-computed results by returning $pt(\langle \ell_{16}, z \rangle) = \{u, i\}$. In this case, a false positive for z will end up being reported.

4 VALUE-FLOW-BASED DEMAND-DRIVEN POINTER ANALYSIS

We first detail the construction of an interprocedural Sparse Value-Flow Graph (SVFG), the core data structure for our demand-driven analysis. Next, we describe the inference rules of SUPA in a field- and flow-sensitive setting (Section 4.2). We then discuss our context-sensitive extension (Section 4.3).

4.1 Interprocedural Sparse Value-Flow Graph

SVFG serves as a key program representation for our demand-driven analysis. It is built upon the interprocedural memory SSA form, which captures def-use information of both top-level and address-taken variables. The def-use chains (value-flows) of address-taken variables obtained on memory SSA are over-approximated using fast but imprecise Andersen's analysis. The imprecise points-to information and value-flows computed this way will be refined by our demand-driven pointer analysis.

4.1.1 Interprocedural Memory SSA

Starting with LLVM's partial SSA form, we first perform a pre-analysis by using flow- and context-insensitive Andersen's algorithm [3], implemented in SVF [44]. We then put address-taken variables in memory SSA form, by using the SSA construction algorithm [12].

Given a variable v, $AnderPts(v)$ represents its points-to set computed by Andersen's algorithm. The following are two steps for building memory SSA [48] with illustrative intraprocedural and interprocedural examples in Figures 4 and 5 respectively by revisiting the example in Figure 2.

Step 1: Computing Modification and Reference Side-Effects As shown in Figure 4(a), every load, e.g., $t1 = *q$ is annotated with a $\boldsymbol{\mu}(a)$ operator for each object a pointed by q, i.e., $a \in AnderPts(q)$ to represent a potential use of a at the load. Similarly, every store, e.g., $*p = x$ is annotated with a $a = \boldsymbol{\chi}(a)$ operator for each object $a \in AnderPts(p)$ to represent a potential def and use of a at the store. If a can be *strongly updated*, then a receives whatever x points to and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a *weak update* to a.

We compute the side-effects of a function call by applying a lightweight interprocedural mod-ref analysis [48, §4.2.1]. For a given callsite ℓ, it is annotated with $\boldsymbol{\mu}(a)$ ($a = \boldsymbol{\chi}(a)$) if a may be read (modified) inside the callees of ℓ (discovered by Andersen's pointer analysis). In addition, appropriate $\boldsymbol{\chi}$ and $\boldsymbol{\mu}$ operators are also added for the FUNENTRY and FUNEXIT instructions of these callees in order to mimic passing parameters and returning results for address-taken variables.

Figure 5(a) gives an example modified from Figure 4(a) by moving the four swap instructions into a function, swap. For read side-effects, $\boldsymbol{\mu}(a)$ and $\boldsymbol{\mu}(c)$ are added before callsite ℓ_7 to represent the potential uses of a and c in swap. Correspondingly, swap's FUNENTRY instruction ℓ_8 is annotated with $a = \boldsymbol{\chi}(a)$ and $c = \boldsymbol{\chi}(c)$ to receive the values of a and c passed from ℓ_7. For modification side-effects, $a = \boldsymbol{\chi}(a)$ and $c = \boldsymbol{\chi}(c)$ are added after ℓ_7 to receive the potentially modified values of a and c returned from swap's FUNEXIT instruction ℓ_{13}, which are annotated with $\boldsymbol{\mu}(a)$ and $\boldsymbol{\mu}(c)$.

Step 2: Memory SSA Renaming All the address-taken variables are converted into SSA form as suggested in [11]. Every $\boldsymbol{\mu}(a)$ is treated as a use of a. Every $a = \boldsymbol{\chi}(a)$ is treated as both a def and use of a, as a may admit only a weak update. Then the SSA form for address-taken variables is obtained by applying a standard SSA construction algorithm [12]. For the program annotated with μ's and χ's in Figure 4(a), Figure 4(b) gives its memory SSA form. Similarly, Figure 5(b) gives the memory SSA form for Figure 5(a).

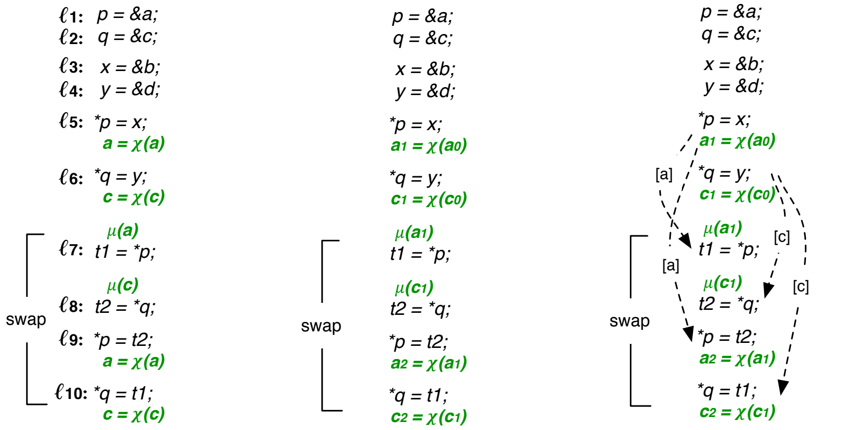
4.1.2 Sparse Value-Flow Graph

Once both top-level and address-taken variables are in SSA form, their def-use chains are immediately available, as shown in Table 2(a). We discussed top-level variables earlier. For the two address-taken variables a and c in Figure 2, Figure 4(c) depicts their def-use chains, i.e., sparse value-flows for the memory SSA form in Figure 4(b). Similarly, Figure 5(c) gives their sparse value-flows for the memory SSA form in Figure 5(b).

Given a program, a *sparse value-flow graph* (SVFG), $G_{\text{vfg}} = (N, E)$, is a multi-edged directed graph that captures its def-use chains for both top-level and address-taken variables. N is the set of nodes representing all instructions and E is the set of edges representing all potential def-use chains. In particular, an edge $\ell_1 \xrightarrow{v} \ell_2$, where $v \in \mathcal{V}$, from statement ℓ_1 to statement ℓ_2 signifies a potential def-use chain for v with its def at ℓ_1 and use at ℓ_2. We refer to $\ell_1 \xrightarrow{v} \ell_2$ a *direct value-flow* if $v \in \mathcal{P}$ and an *indirect value-flow* if $v \in \mathcal{O}$. This representation is *sparse* since the intermediate program points between ℓ_1 and ℓ_2 are omitted, thereby enabling the underlying points-to information to be gradually refined by applying a sparse demand-driven pointer analysis.

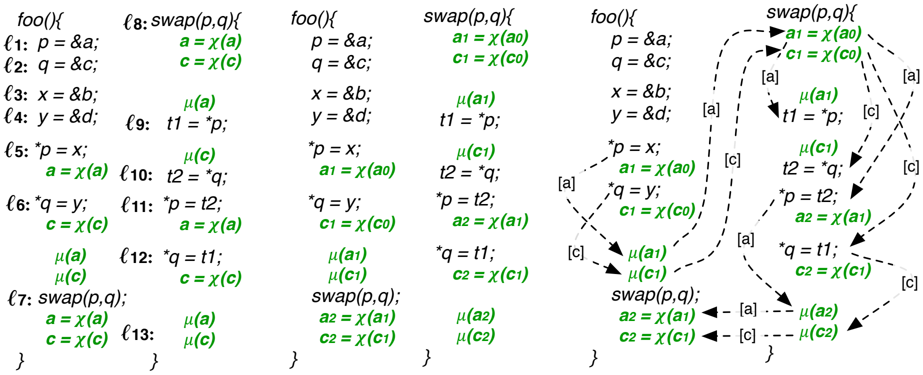
Table 2(b) gives the rules for connecting value-flows between two instructions based on the defs and uses computed in Table 2(a). For intraprocedural value-flows, `[INTRA-TOP]` and `[INTRA-ADDR]` handle top-level and address-taken variables, respectively. In SSA form, every use of a variable only has a unique definition. For a use of a identified as a_i (with its i-th version) at ℓ' annotated with $\boldsymbol{\mu}(a_i)$, its unique definition in SSA form is a_i at an ℓ annotated with $a_i = \boldsymbol{\chi}(a_{i-1})$. Then, $\ell \xrightarrow{a} \ell'$ is generated to represent potentially the value-flow of a from ℓ to ℓ'. Thus, the PHI functions introduced for address-taken variables will be ignored, as the value a in $\ell \xrightarrow{a} \ell'$ is not versioned.

Let us consider interprocedural value-flows. The def-use information in Table 2(a) is only intraprocedural. Ac-



(a) Step 1: adding μs and χs (b) Step 2: renaming (c) Sparse value-flows of a and c

Fig. 4: Memory SSA form and intraprocedural sparse value-flows for Figure 2, obtained with Andersen's analysis: $AnderPts(p) = \{a\}$ and $AnderPts(q) = \{c\}$.



(a) Step 1: adding μs and χs (b) Step 2: renaming (c) Sparse value-flows of a and c

Fig. 5: Memory SSA form and interprocedural sparse value-flows for an example modified from Figure 2 with its four swap instructions moved into a separate function, called swap. ℓ_8 and ℓ_{13} correspond to the FunEntry and FunExit of swap.

TABLE 2: Def-use information and the constructed intra- and inter-procedural value-flows of both top-level and address-taken variables. Def_v (Use_v) denotes the set of definition (use) instructions for a variable $v \in \mathcal{V}$.

(a) Def-use information

Instruction ℓ		Def-Use Information of variables on memory SSA
$p = \&o$		$\{\ell\} = \mathrm{Def}_p$
$p = q$		$\{\ell\} = \mathrm{Def}_p \quad \ell \in \mathrm{Use}_q$
$p = \phi(q, r)$		$\{\ell\} = \mathrm{Def}_p \quad \ell \in \mathrm{Use}_q \quad \ell \in \mathrm{Use}_r$
$p = \&q \rightarrow fld$		$\{\ell\} = \mathrm{Def}_p \quad \ell \in \mathrm{Use}_q$
$p = *q$	$\mu(a_i)$	$\{\ell\} = \mathrm{Def}_p \quad \ell \in \mathrm{Use}_q \quad \ell \in \mathrm{Use}_{a_i}$
$*p = q$	$a_{i+1} = \chi(a_i)$	$\ell \in \mathrm{Use}_p \quad \ell \in \mathrm{Use}_q \quad \ell \in \mathrm{Def}_{a_{i+1}} \quad \ell \in \mathrm{Use}_{a_i}$
$p = q(r_1, \ldots, r_n)$		$\{\ell\} = \mathrm{Def}_p \quad \ell \in \mathrm{Use}_q \quad \forall\, i \in 1, \ldots, n : \ell \in \mathrm{Use}_{r_i}$
$\mu(a_i) \quad a_{j+1} = \chi(a_j)$		$\ell \in \mathrm{Use}_{a_i} \quad \ell \in \mathrm{Def}_{a_{j+1}} \quad \ell \in \mathrm{Use}_{a_j}$
$f(r_1, \ldots, r_n)$	$a_{i+1} = \chi(a_i)$	$\forall\, i \in 1, \ldots, n : \ell \in \mathrm{Def}_{r_i} \quad \ell \in \mathrm{Def}_{a_{i+1}} \quad \ell \in \mathrm{Use}_{a_i}$
$ret_f\ p$	$\mu(a_i)$	$\ell \in \mathrm{Use}_p \quad \ell \in \mathrm{Use}_{a_i}$

(b) Value-flow information

Value-Flow Type	Def-Use Info of a memory SSA variable	Value-flows
[INTRA-TOP]	$\ell \in \mathrm{Def}_p \quad \ell' \in \mathrm{Use}_p$	$\ell \xrightarrow{p} \ell'$
[INTRA-ADDR]	$\ell \in \mathrm{Def}_a \quad \ell' \in \mathrm{Use}_a$	$\ell \xrightarrow{a} \ell'$
[INTER-CALL-TOP]	$\ell : p = q(r_1, \ldots, r_n) \quad \ell' : f(r'_1, \ldots, r'_n)$ $o_f \in AnderPts(q)$	$\forall i \in 1, \ldots, n : \ell \xrightarrow{r_i} \ell'$
[INTER-RET-TOP]	$\ell : p = q(\ldots) \quad \ell' : ret_f\ p'$ $a_f \in AnderPts(q)$	$\ell' \xrightarrow{p} \ell$
[INTER-CALL-ADDR]	$\ell : p = q(\ldots)\ \mu(a_i) \quad \ell' : f(\ldots)\ a_{j+1} = \chi(a_j)$ $a_f \in AnderPts(q)$	$\ell \xrightarrow{a} \ell'$
[INTER-RET-ADDR]	$\ell : \_ = q(\ldots)\ a_{j+1} = \chi(a_j) \quad \ell' : ret_f\ \_\ \ \mu(a_i)$ $a_f \in AnderPts(q)$	$\ell' \xrightarrow{a} \ell$

cording to Table 2(b), interprocedural value-flows are constructed to represent parameter passing for top-level variables ([INTER-CALL-TOP] and [INTER-RET-TOP]), and the μ/χ operators annotated at FunEntry, FunExit and Call for address-taken variables ([INTER-CALL-ADDR] and [INTER-RET-ADDR]).

[INTER-CALL-TOP] connects the value-flow from an actual argument r_i at a call instruction ℓ to its corresponding formal parameter r'_i at the FunEntry ℓ' of every callee f invoked at the call. Conversely, [INTER-RET-TOP] models the value-flow from the FunExit instruction of f to every callsite where f is invoked. Just like for top-level variables, [INTER-CALL-ADDR] and [INTER-RET-ADDR] build the value-flows of address-taken variables across the functions according to the annotated μ's and χ's. Note that the versions i and j of an SSA variable a in different functions may be different. For example, Figure 5(c) illustrates the four inter-procedural value-flows $\ell_7 \xrightarrow{a} \ell_8$, $\ell_7 \xrightarrow{c} \ell_8$, $\ell_{13} \xrightarrow{a} \ell_7$ and $\ell_{13} \xrightarrow{c} \ell_7$ obtained by applying the two rules to Figure 5(b).

The SVFG obtained this way may contain spurious def-use chains, such as $\ell_5 \xrightarrow{a} \ell_9$ in Figure 4, as Andersen's flow- and context-insensitive pointer analysis is fast but imprecise. However, this representation allows imprecise points-to information to be refined by performing sparse whole-program flow-sensitive pointer analysis as in prior work [20, 32, 49, 61]. In this paper, we introduce a demand-driven flow- and context-sensitive pointer analysis with strong updates that can answer points-to queries efficiently and precisely on-demand, by removing spurious def-use chains in the SVFG iteratively.

4.2 SUPA: Flow-Sensitivity

We present a formalization of a single-stage SUPA consisting of only a flow-sensitive (FS) analysis. Given a program, SUPA will operate on its SVFG representation G_{vfg} constructed by applying Andersen's analysis [3] as a pre-analysis, as discussed in Section 4.1.2 and illustrated in Section 3.

Let $\mathbb{V} = \mathcal{L} \times \mathcal{V}$ be the set of labeled variables lv, where $\mathcal{L}$ is the set of statement labels and $\mathcal{V} = \mathcal{P} \cup \mathcal{O}$ as defined in Table 1. SUPA conducts a backward reachability analysis flow-sensitively on G_{vfg} by computing a reachability relation, $\hookleftarrow \subseteq \mathbb{V} \times \mathbb{V}$. In our formalism, $\langle\ell, v\rangle \hookleftarrow \langle\ell', v'\rangle$ signifies a value-flow from a def of v' at ℓ' to a use of v at ℓ through one or multiple value-flow paths in G_{vfg}. For an object o created at an ADDROF statement, i.e., an allocation site at ℓ', identified as $\langle\ell', o\rangle$, we must distinguish it from $\langle\ell, o\rangle$ accessed elsewhere at ℓ in our inference rules. Our abbreviation for $\langle\ell', o\rangle$ is $\widehat{o}$.

Given a points-to query $\langle\ell, v\rangle$, SUPA computes $pt(\langle\ell, v\rangle)$, i.e., the points-to set of $\langle\ell, v\rangle$ by finding all reachable target objects $\widehat{o}$, defined as follows:

$$pt(\langle\ell, v\rangle) = \{o \mid \langle\ell, v\rangle \hookleftarrow \widehat{o}\} \quad (1)$$

Despite flow-sensitivity, our formalization in Figure 6 makes no explicit references to program points. As SUPA operates on the def-use chains in G_{vfg}, each variable $\langle\ell, v\rangle$ mentioned in a rule appears at the point just after ℓ, where v is defined.

Let us examine our rules in detail. By [ADDR], an object $\widehat{o}$ created at an allocation site ℓ is backward reachable from p at ℓ (or precisely at the point after ℓ). The pre-computed direct value-flows across the top-level variables in G_{vfg} are always precise ([COPY] and [PHI]). In partial SSA form, [PHI] exists only for top-level variables (Section 4.1.2).

However, the indirect value-flows across the address-taken variables in G_{vfg} can be imprecise; they need to be refined on the fly to remove the spurious aliases thus introduced. When handling a load $p = *q$ in [LOAD], we can traverse backwards from p at ℓ to the def of o at ℓ' only if $*q$ is *actually* refers to object o at ℓ, which requires the reachability relation $\langle\ell'', q\rangle \hookleftarrow \widehat{o}$ to be computed recursively. A store $*p = q$ is handled similarly ([STORE]): q defined at ℓ' can be reached backwards by o at ℓ only if o is aliased with $*p$ at ℓ.

If $*q$ in a load $\cdots = *q$ is aliased with $*p$ in a store $*p = \cdots$ executed earlier, then p and q must be both backward reachable from $\widehat{o}$. Otherwise, any alias relation established between $*p$ and $*q$ in G_{vfg} by pre-analysis must be spurious and will thus be filtered out by value-flow refinement.

[SU/WU] models strong and weak updates at a store $\ell : *p = \_$. Defining its kill set *kill*(ℓ, p) involves three cases. In Case (1), p points to one *singleton object* o' in *singletons*, which contains all objects in $\mathcal{O}$ except the local variables in recursion, arrays (treated monolithically) or heap objects [29]. In Section 4.3, we discuss how to apply strong updates to heap objects context-sensitively. A strong update is then possible to o. By killing its old contents at ℓ', no further backward traversal along the def-use chain $\ell' \xrightarrow{o} \ell$ is needed. Thus, $\langle\ell, o\rangle \hookleftarrow \langle\ell', o\rangle$ is falsified. In Case (2), the points-to set of p is empty. Again, further traversal to $\langle\ell', o\rangle$ must be prevented to avoid dereferencing a null pointer as is standard [19, 20, 29]. In Case (3), a weak update is performed to o so that its old contents at ℓ' are preserved. Thus, $\langle\ell, o\rangle \hookleftarrow \langle\ell', o\rangle$ is established, which implies that the backward traversal along $\ell' \xrightarrow{o} \ell$ must continue.

[FIELD] handles field-sensitivity. For a field access (e.g., $p = \&q \rightarrow fld$), pointer p points to the field object o_{fld} of object o pointed to by q.

[CALL] and [RET] handle the reachability traversal interprocedurally by computing the call graph for the program on the fly instead of relying on the imprecisely precomputed call graph built by the pre-analysis as in [20]. In the SVFG, the interprocedural value-flows sinking into a callee function f may come from a spurious indirect callsite ℓ. To avoid this, both rules ensure that the function pointer q at ℓ actually points to f ([CALL] and [RET]). Essentially, given a points-to query z at an indirect callsite $\ell : z = (*fp)()$, instead of analyzing all the callees found by the pre-analysis, SUPA recursively computes the points-to set of fp to discover new callees at the callsite and then continues refining $pt(\langle\ell, z\rangle)$ using the new callees.

Finally, $\hookleftarrow$ is transitive, stated by [COMPO].

4.2.1 Monotonicity

We discuss the monotonicity of our strong update analysis since monotonicity guarantees the termination and correctness of our static analysis. Particularly, we focus on the store statement since all other kinds of statements do not overwritten old values of objects. The $kill(l, p)$ set here is to implement the strong and weak updates while maintaining monotonicity of the analysis. For a store $*p = q$, $\mathcal{O} \backslash kill(l, p)$ depends on the points-to set of the top-level pointer p.

- (1) p points to an empty set. In this case, neither strong nor weak updates are performed.
- (2) p points to only o which is a singleton object (i.e., a unique object at runtime). In this case, o's value is strongly updated to q and its old information is killed.
- (3) p points to o which is not a singleton object and/or p points to other objects. In this case, o should keep its old points-to value.

The analysis can only transit from (1) $->$ (2), (1)$->$(3), (2)$->$(3), but never from (3)$->$(1) or (3)$->$(2) or (2)$->$(1). Intuitively, a transition can always start from a strong update of o to its weak update, but not the opposite way.

Handling null pointer is a special case to maintain monotonicity as also mentioned in [29]. If the points-to set of p is empty at a store, the $\mathcal{O} \backslash kill(l, p)$ should always be empty (i.e, $kll(l, p) = \mathcal{O}$), which disallows the analysis to perform strong or weak updates at this store. This is because either we will revisit the store instruction after p is updated or the program crashes when executing this statement due to dereferencing a null pointer. This special handling guarantees the monotonicity, i.e., preventing the analysis from looping forever (e.g., (1)$->$(2)$->$(1)) without converging to a fixed point.

For example, when p points to null, if we allows a weak update for an object (e.g., a) in a non-empty set $A \backslash kill(l, p)$

$$\texttt{[ADDR]}\;\dfrac{\ell : p = \&o}{\langle \ell, p\rangle \hookleftarrow \widehat{o}} \qquad \texttt{[COPY]}\;\dfrac{\ell : p = q \quad \ell' \xrightarrow{q} \ell}{\langle \ell, p\rangle \hookleftarrow \langle \ell', q\rangle} \qquad \texttt{[PHI]}\;\dfrac{\ell : p = \phi(q, r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{\langle \ell, p\rangle \hookleftarrow \langle \ell', q\rangle \quad \langle \ell, p\rangle \hookleftarrow \langle \ell'', r\rangle}$$

$$\texttt{[FIELD]}\;\dfrac{\ell : p = \&q \rightarrow fld \quad \ell' \xrightarrow{q} \ell \quad \langle \ell', q\rangle \hookleftarrow \widehat{o}}{\langle \ell, p\rangle \hookleftarrow \widehat{o_{fld}}} \qquad \texttt{[LOAD]}\;\dfrac{\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle \ell'', q\rangle \hookleftarrow \widehat{o} \quad \ell' \xrightarrow{o} \ell}{\langle \ell, p\rangle \hookleftarrow \langle \ell', o\rangle}$$

$$\texttt{[STORE]}\;\dfrac{\ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle \ell'', p\rangle \hookleftarrow \widehat{o} \quad \ell' \xrightarrow{q} \ell}{\langle \ell, o\rangle \hookleftarrow \langle \ell', q\rangle} \qquad \texttt{[SU/WU]}\;\dfrac{\ell : *p = \_ \qquad \ell' \xrightarrow{o} \ell \qquad o \in \mathcal{O} \backslash \mathit{kill}(\ell, p)}{\langle \ell, o\rangle \hookleftarrow \langle \ell', o\rangle}$$

$$\texttt{[CALL]}\;\dfrac{\begin{array}{c} \ell : \_ = q(\ldots, r, \ldots) \quad \boldsymbol{\mu}(o_j) \qquad \ell' : f(\ldots, r', \ldots) \quad o_{i+1} = \boldsymbol{\chi}(o_i) \\ \ell'' \xrightarrow{q} \ell \quad \langle \ell'', q\rangle \hookleftarrow \widehat{o_f} \quad \ell \xrightarrow{r} \ell' \quad \ell \xrightarrow{o} \ell' \end{array}}{\langle \ell', r'\rangle \hookleftarrow \langle \ell, r\rangle \quad \langle \ell', o\rangle \hookleftarrow \langle \ell, o\rangle}$$

$$\texttt{[RET]}\;\dfrac{\begin{array}{c} \ell : p = q(\ldots) \quad o_{j+1} = \boldsymbol{\chi}(o_j) \qquad \ell' : ret_f \; p' \quad \boldsymbol{\mu}(o_i) \\ \ell'' \xrightarrow{q} \ell \quad \langle \ell'', q\rangle \hookleftarrow \widehat{o_f} \quad \ell' \xrightarrow{p'} \ell \quad \ell' \xrightarrow{o} \ell \end{array}}{\langle \ell, p\rangle \hookleftarrow \langle p', \ell'\rangle \quad \langle \ell, o\rangle \hookleftarrow \langle \ell', o\rangle} \qquad \texttt{[COMPO]}\;\dfrac{lv \hookleftarrow lv' \qquad lv' \hookleftarrow lv''}{lv \hookleftarrow lv''}$$

$$\mathit{kill}(\ell, p) = \begin{cases} \{o'\} & \text{if } \mathit{pt}(\langle \ell, p\rangle) = \{o'\} \wedge o' \in \mathit{singletons} \\ \mathcal{O} & \text{else if } \mathit{pt}(\langle \ell, p\rangle) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Fig. 6: Single-stage flow-sensitive SUPA analysis with demand-driven strong updates.

$$\dfrac{\dfrac{\ell_{13} : t3 = *p \;\; \ell_1 \xrightarrow{p} \ell_{13} \;\; \dfrac{\ell_1 : p = \&a}{\langle \ell_1, p\rangle \hookleftarrow \widehat{a}}\,\textbf{[ADDR]} \;\; \ell_9 \xrightarrow{a} \ell_{13}}{\langle \ell_{13}, t3\rangle \hookleftarrow \langle \ell_9, a\rangle}\,\textbf{[LOAD]} \quad \dfrac{\ell_9 : *p = t2 \;\; \ell_1 \xrightarrow{p} \ell_9 \;\; \boxed{\langle \ell_1, p\rangle \hookleftarrow \widehat{a}} \;\; \ell_8 \xrightarrow{t2} \ell_9}{\langle \ell_9, a\rangle \hookleftarrow \langle \ell_8, t2\rangle}\,\textbf{[STORE]}}{\langle \ell_{13}, t3\rangle \hookleftarrow \langle \ell_8, t2\rangle}\,\textbf{[COMPO]}$$

(a) Deriving $pt(\langle \ell_{13}, t3\rangle)$ (corresponding to ① – ④ in Figure 3(d))

$$\dfrac{\dfrac{\dfrac{\langle \ell_{13}, t3\rangle \hookleftarrow \langle \ell_8, t2\rangle \;\; \dfrac{\ell_8 : t2 = *q \;\; \ell_2 \xrightarrow{q} \ell_8 \;\; \dfrac{\ell_2 : q = \&c}{\langle \ell_2, q\rangle \hookleftarrow \widehat{c}}\,\textbf{[ADDR]} \;\; \ell_6 \xrightarrow{c} \ell_8}{\langle \ell_8, t2\rangle \hookleftarrow \langle \ell_6, c\rangle}\,\textbf{[LOAD]}}{\langle \ell_{13}, t3\rangle \hookleftarrow \langle \ell_6, c\rangle}\,\textbf{[COMPO]} \quad \dfrac{\ell_6 : *q = y \;\; \ell_2 \xrightarrow{q} \ell_6 \;\; \boxed{\langle \ell_2, q\rangle \hookleftarrow \widehat{c}} \;\; \ell_4 \xrightarrow{y} \ell_6}{\langle \ell_6, c\rangle \hookleftarrow \langle \ell_4, y\rangle}\,\textbf{[STORE]}}{\langle \ell_{13}, t3\rangle \hookleftarrow \langle \ell_4, y\rangle}\,\textbf{[COMPO]} \quad \dfrac{\ell_4 : y = \&d}{\langle \ell_4, y\rangle \hookleftarrow \widehat{d}}\,\textbf{[ADDR]}}{\langle \ell_{13}, t3\rangle \hookleftarrow \widehat{d}}\,\textbf{[COMPO]}$$

(b) Deriving $pt(\langle \ell_{13}, t3\rangle)$ (corresponding to ⑤ – ⑦ in Figure 3(d))

$$\dfrac{\dfrac{\dfrac{\ell_{16} : z = *t3 \;\; \ell_{13} \xrightarrow{t3} \ell_{16} \;\; \langle \ell_{13}, t3\rangle \hookleftarrow \widehat{d} \;\; \ell_{15} \xrightarrow{d} \ell_{16}}{\langle \ell_{16}, z\rangle \hookleftarrow \langle \ell_{15}, d\rangle}\,\textbf{[LOAD]} \quad \dfrac{\ell_{15} : *t3 = v \;\; \ell_{13} \xrightarrow{t3} \ell_{15} \;\; \boxed{\langle \ell_{13}, t3\rangle \hookleftarrow \widehat{d}} \;\; \ell_{12} \xrightarrow{v} \ell_{15}}{\langle \ell_{15}, d\rangle \hookleftarrow \langle \ell_{12}, v\rangle}\,\textbf{[STORE]}}{\langle \ell_{16}, z\rangle \hookleftarrow \langle \ell_{12}, v\rangle}\,\textbf{[COMPO]} \quad \dfrac{\ell_{12} : v = \&i}{\langle \ell_{12}, v\rangle \hookleftarrow \widehat{i}}\,\textbf{[ADDR]}}{\langle \ell_{16}, z\rangle \hookleftarrow \widehat{i}}\,\textbf{[COMPO]}$$

(c) Deriving $pt(\langle \ell_{16}, z\rangle)$ (corresponding to ⑧ – ⑨ in Figure 3(d))

Fig. 7: Reachability derivations for $pt(\langle \ell_{16}, z\rangle)$ shown in Figure 3(d) (with reuse of cached points-to results inside each box).

([WU/SU] in Figure 5), this weak update may cause p points to a, then a strong update will be performed at this store to overwrite the previous values of a, preventing p from pointing to a in the first place. This transition from a weak update to a strong one is invalid and may generate infinite derivations without reaching a fixed point.

4.2.2 Revisiting our motivating example

Let us try all our rules, by first revisiting our motivating example where strong updates are performed (Example 1) and then examining weak updates (Example 2).

Example 1. Figure 7 shows how we apply the rules of SUPA to answer $pt(\langle \ell_{16}, z\rangle)$ illustrated in Figure 3(d). [SU/WU] (implicit in these derivations) is applied to ℓ_6, ℓ_9 and ℓ_{15} to cause a strong update at each store. At ℓ_6, $pt(\langle \ell_6, q\rangle) = \{c\}$, the old contents in c are killed. At ℓ_9, $\ell_5 \xrightarrow{a} \ell_9$ becomes spurious since $\langle \ell_9, a\rangle \hookleftarrow \langle \ell_5, a\rangle$ is falsified. $\ell_{14} \xrightarrow{b} \ell_{15}$ is spurious and filtered out since $t3$ only points to d, causing $\ell_{15} \xrightarrow{b} \ell_{16}$ falsified in the first place and there is no need for backward traversal against value-flows of object b from ℓ_{16} to ℓ_{15}. Therefore, any incoming value-flow of b reaching ℓ_{15} (e.g., $\ell_{14} \xrightarrow{b} \ell_{15}$) will never be traversed and become infeasible. Finally,

$\ell_{15} \xrightarrow{b} \ell_{16}$ is ignored since $t3$ points to d only (`[LOAD]`). □

SUPA improves performance by caching points-to results to reduce redundant traversal, with reuse happening in the marked boxes in Figure 7. For example, in Figure 7(c), $pt(\langle \ell_{13}, t3 \rangle) = \{\widehat{d}\}$ computed in `[LOAD]` is reused in `[STORE]`.

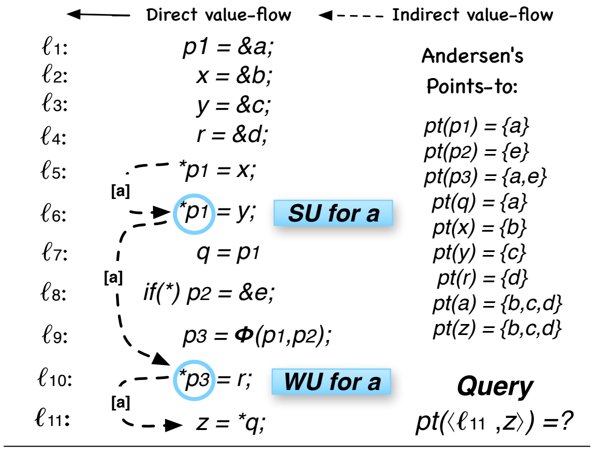


Fig. 8: Resolving $pt(\langle \ell_{11}, z \rangle) = \{c, d\}$ with a weak update.

Example 2. Let us consider a weak update example in Figure 8 by computing $pt(\langle \ell_{11}, z \rangle)$ on-demand. At the confluence point ℓ_9, $p3$ receives the points-to information from both $p1$ and $p2$ in its two branches: $\langle \ell_9, p_3 \rangle \hookleftarrow \widehat{a}$ and $\langle \ell_9, p_3 \rangle \hookleftarrow \widehat{e}$. Thus, a weak update is performed to the two locations a and e at ℓ_{10}. Let us focus on $\widehat{a}$ only. By applying `[STORE]`, $\langle \ell_{10}, a \rangle \hookleftarrow \langle \ell_4, r \rangle \hookleftarrow \widehat{d}$. By applying `[SU/WU]`, $\langle \ell_{10}, a \rangle \hookleftarrow \langle \ell_6, a \rangle \hookleftarrow \langle \ell_3, y \rangle \hookleftarrow \widehat{c}$. Thus, $pt(\langle \ell_{10}, a \rangle) = \{c, d\}$, which excludes b due to a strong update performed at ℓ_6. As $pt(\langle \ell_7, q \rangle) = \{a\}$, we obtain $pt(\langle \ell_{11}, z \rangle) = \{c, d\}$. □

Unlike [29], which falls back to the flow-insensitive points-to information for all weakly updated objects, SUPA handles them as precisely as (whole-program) flow-sensitive analysis subject to a sufficient budget. In Figure 8, due to a weak update performed to a at ℓ_{10}, $pt(\langle \ell_{10}, a \rangle) = \{c, d\}$ is obtained, forcing their approach to adopt $pt(\langle \ell_{10}, a \rangle) = \{b, c, d\}$ thereafter, causing $pt(\langle \ell_{11}, z \rangle) = \{b, c, d\}$. By maintaining flow-sensitivity with a strong update applied to ℓ_6 to kill b, SUPA obtains $pt(\langle \ell_{11}, z \rangle) = \{c, d\}$ precisely.

4.2.3 *Handling Value-Flow Cycles*

To compute soundly and precisely the points-to information in a value-flow cycle in the SVFG, SUPA retraverses it whenever new points-to information is found until a fix point is reached.

Example 3. Figure 9 shows a value-flow cycle formed by $\ell_5 \xrightarrow{x} \ell_6$ and $\ell_6 \xrightarrow{z} \ell_5$. To compute $pt(\langle \ell_6, z \rangle)$, we must compute $pt(\langle \ell_5, x \rangle)$, which requires the aliases of $*z$ at the load $\ell_5 : x = *z$ to be found by using $pt(\langle \ell_6, z \rangle)$. SUPA computes $pt(\langle \ell_6, z \rangle)$ by analyzing this value-flow cycle in two iterations. In the first iteration, a pointed-to target $\widehat{b}$ is found since $\langle \ell_6, z \rangle \hookleftarrow \langle \ell_4, y \rangle \hookleftarrow \widehat{b}$. Due to $\langle \ell_2, q \rangle \hookleftarrow \widehat{b}$, $*z$ and $*q$ are found to be aliases. In the second iteration, another target $\widehat{a}$ is found since $\langle \ell_6, z \rangle \hookleftarrow \langle \ell_5, x \rangle \hookleftarrow \langle \ell_3, b \rangle \hookleftarrow \langle \ell_1, p \rangle \hookleftarrow \widehat{a}$. Thus, $pt(\langle \ell_6, z \rangle) = \{a, b\}$ is obtained. □

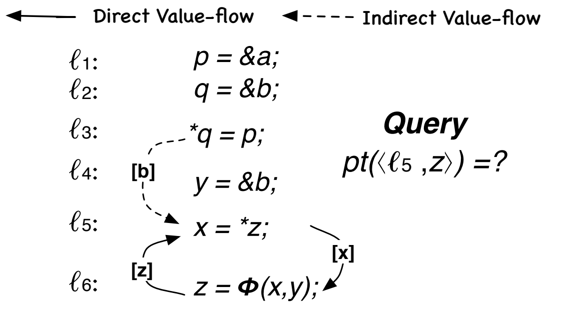


Fig. 9: Resolving $pt(\langle \ell_5, z \rangle) = \{a, b\}$ in a value-flow cycle.

4.2.4 *Field-Sensitivity*

Field-insensitive pointer analysis does not distinguish different fields of a struct object, and consequently, gives up opportunities for performing strong updates to a struct object, as a struct object may actually represent its distinct fields. In contrast, SUPA is truly field-sensitive, by avoiding the two limitations altogether.

Example 4. Figure 10 illustrates the effects of field-sensitivity on computing the points-to information for r at ℓ_{11}. Without field-sensitivity, as illustrated in Figure 10(a), the two statements at ℓ_4 and ℓ_5 are analyzed as if they were $\ell_4 : p = \&x$ and $\ell_5 : q = \&x$. As a result, no strong update is possible at ℓ_6 and ℓ_7, since x, which represents possibly multiple fields, is not a singleton. Thus, $pt(\langle \ell_{11}, r \rangle) = \{b, c\}$.

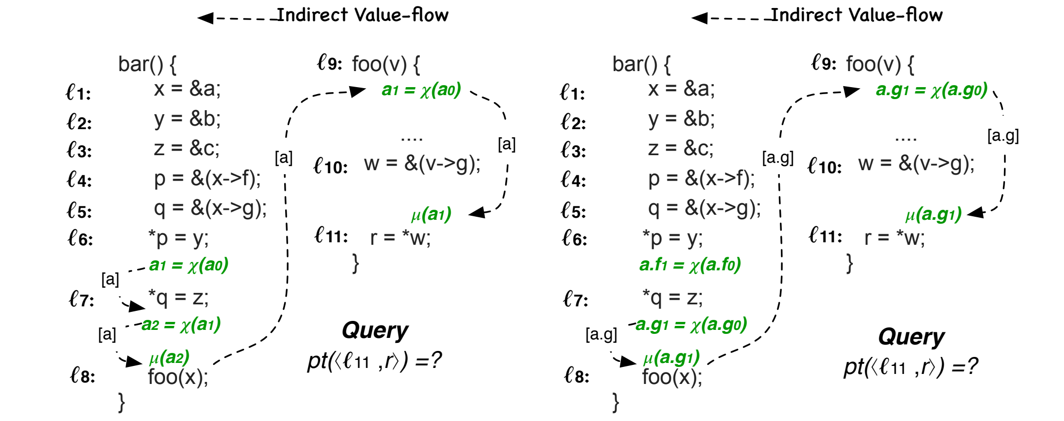
SUPA is field-sensitive. Assuming that f and g are the first and second field of the struct object a, our field-index-based modeling (Section 2) creates a as a field-insensitive object pointed by x. Two sub objects $a.f$ and $a.g$ are created for fields f and g when deriving $p = \&(x{-}{>}\ f)$ and $p = \&(x{-}{>}\ g)$. Therefore, pointers p and q point to two distinct objects $a.f$ and $a.g$ respectively. Note that if replacing $*w$ with $*v$ in foo, then a and its two fields $a.f$ and $a.g$ are all aliased with $*v$.

To answer the points-to query for r at $\ell_{11} : r = *w$, we first compute the points-to of w, which is defined at ℓ_{10}, where $\langle \ell_{10}, v \rangle \hookleftarrow \langle \ell_9, v \rangle \hookleftarrow \langle \ell_8, x \rangle \hookleftarrow \langle \ell_1, x \rangle \hookleftarrow \widehat{a}$. By applying `[FIELD]` for $\ell_{10} : w = \&(v- > g)$, we obtain $\langle \ell_{10}, w \rangle \hookleftarrow \widehat{a.g}$. Thus, we get $\langle \ell_{11}, r \rangle \hookleftarrow \langle \ell_9, a.g \rangle$ after applying `[LOAD]` at ℓ_{11}. By traversing the three indirect def-use chains for $a.g$, $\ell_7 \xrightarrow{b.g} \ell_8$, $\ell_8 \xrightarrow{a.g} \ell_9$ and $\ell_9 \xrightarrow{a.g} \ell_{11}$, backwards from ℓ_{11}, we obtain $\langle \ell_{11}, r \rangle \hookleftarrow \langle \ell_9, a.g \rangle \hookleftarrow \langle \ell_8, a.g \rangle \hookleftarrow \langle \ell_7, a.g \rangle \hookleftarrow \langle \ell_3, z \rangle \hookleftarrow \widehat{c}$. □

4.2.5 *Properties*

Theorem 1 (Soundness). SUPA is sound in analyzing a program as long as its pre-analysis (for computing the SVFG of the program) is sound.

Proof Sketch: When building the SVFG for a program, the def-use chains for its top-level variables are identified explicitly in its partial SSA form. If the pre-analysis (for computing the sparse value-flow graph of the program)



(a) Field-insensitive value-flows (*pt(<ℓ11 ,r>) = {b,c}*) (b) Field-sensitive value-flows (*pt(<ℓ11 ,r>) = {c}*)

Fig. 10: Resolving $pt(\langle\ell_{11}, r\rangle) = \{c\}$ with field-sensitivity.

is sound, then the def-use chains built for all the address-taken variables are over-approximate. According to its inference rules in Figure 4, SUPA performs essentially a flow-sensitive analysis on-demand, by restricting the propagation of points-to information along the precomputed def-use chains, and falls back to the sound points-to information computed by the pre-analysis when running out of its given budgets. Thus, SUPA is sound if the pre-analysis is sound. □

Theorem 2 (Precision). Given a points-to query $\langle\ell, v\rangle$, $pt(\langle\ell, v\rangle)$ computed by SUPA is the same as that computed by (whole-program) FS if SUPA can successfully resolve the points-to query within a given budget.

Proof Sketch: Let $pt_{\text{SUPA}}(\langle\ell, v\rangle)$ and $pt_{\text{FS}}(\langle\ell, v\rangle)$ be the points-to sets computed by SUPA and FS, respectively. By Theorem 1, $pt_{\text{SUPA}}(\langle\ell, v\rangle) \supseteq pt_{\text{FS}}(\langle\ell, v\rangle)$, since SUPA is a demand-driven version of FS and thus cannot be more precise. To show that $pt_{\text{SUPA}}(\langle\ell, v\rangle) \subseteq pt_{\text{FS}}(\langle\ell, v\rangle)$, we note that SUPA operates on the SVFG of the program to improve its efficiency, by also filtering out value-flows imprecisely pre-computed by the pre-analysis. For the top-level variables, their direct value-flows are precise. So SUPA proceeds exactly the same as FS (`[ADDR]`, `[COPY]`, `[PHI]`, `[FIELD]`, `[CALL]`, `[RET]` and `[COMPO]`). For the address-taken variables, SUPA establishes the same indirect value-flows flow-sensitively as FS does but in a demand-driven manner, by refining away imprecisely pre-computed value-flows (`[LOAD]`, `[STORE]`, `[SU/WU]`, `[CALL]`, `[RET]` and `[COMPO]`). If SUPA can complete its query within the given budget, then $pt_{\text{SUPA}}(\langle\ell, v\rangle) \subseteq pt_{\text{FS}}(\langle\ell, v\rangle)$. Thus, $pt_{\text{SUPA}}(\langle\ell, v\rangle) = pt_{\text{FS}}(\langle\ell, v\rangle)$. □

4.3 SUPA: Flow- and Context-Sensitivity

We extend our flow-sensitive formalization by considering also context-sensitivity to enable more strong updates (especially now for heap objects). We solve a *balanced-parentheses* problem by matching calls and returns to filter out unrealizable inter-procedural paths [31, 38, 39, 41, 57]. A context stack c is encoded as a sequence of callsites, $[\kappa_1 \ldots \kappa_m]$, where κ_i is a call instruction ℓ. $c \oplus \kappa$ denotes an operation for pushing a callsite κ into c. $c \ominus \kappa$ pops κ from c if c contains κ as its top value or is empty since a realizable path may start and end in different functions.

With context-sensitivity, a statement is parameterized additionally by a context c, e.g., $c, \ell : p = \&o$, to represent its instance when its containing function is analyzed under c. A labeled variable lv has the form $\langle c, \ell, v\rangle$, representing variable v accessed at statement ℓ under context c. An object $\widehat{o}$ that is created at an ADDROF statement under context c is also context-sensitive, identified as $(c, \widehat{o})$.

Given a points-to query $\langle c, \ell, v\rangle$, SUPA computes its points-to set both flow- and context-sensitively by applying the rules given in Figure 11:

$$pt(\langle c, \ell, v\rangle) = \{(c', o) \mid \langle c, \ell, v\rangle \hookleftarrow (c', \widehat{o})\} \tag{2}$$

where the reachability relation $\hookleftarrow$ is now also context-sensitive.

Passing parameters to and returning results from a callee invoked at a callsite are handled by `[C-CALL]` and `[C-RET]`. `[C-CALL]` deals with the direct and indirect value-flows backwards from the entry instruction of a callee function to each of its callsites based on the call graph computed on the fly similarly as in `[CALL]` in Figure 6, except that `[C-CALL]` is context-sensitive. Likewise, `[C-RET]` deals with the direct and indirect value-flows backwards from a callsite to the return instruction of every callee function.

With context-sensitivity, SUPA will filter out more spurious value-flows generated by Andersen's analysis, thereby producing more precise points-to information to enable more strong updates (`[C-SU/WU]`). At a store $c, \ell : *p = \_$, its kill set is context-sensitive. A strong update is applied if p points to a *context-sensitive singleton* $(c', o') \in$ *cxtSingletons*, where o' is a (non-heap) singleton defined in Section 4.2 or a heap object with c' being a *concrete* context, i.e., one not involved in recursion or loops.

Example 5. Let us use an example given in Figure 12 to illustrate the effects of context-sensitive strong updates on computing the points-to information for z at ℓ_5. This example is adapted from a real application, `milc-v6`, given in Figure 18(c). Without context-sensitivity, SUPA will only perform a weak update at $\ell_8 : *x = y$, since x points to both a and b passed into `foo()` from the two callsites at ℓ_3 and ℓ_4. As a result, z at ℓ_5 is found to point to not only what y points to, i.e., c but also what

$$\texttt{[C-ADDR]}\ \frac{c,\ell : p = \&o}{\langle c,\ell,p\rangle \hookleftarrow (c,\widehat{o})} \qquad \texttt{[C-COPY]}\ \frac{c,\ell : p = q \quad \ell' \xrightarrow{q} \ell}{\langle c,\ell,p\rangle \hookleftarrow \langle c,\ell',q\rangle} \qquad \texttt{[C-PHI]}\ \frac{c,\ell : p = \phi(q,r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{\langle c,\ell,p\rangle \hookleftarrow \langle c,\ell',q\rangle \quad \langle c,\ell,p\rangle \hookleftarrow \langle c,\ell'',r\rangle}$$

$$\texttt{[C-FIELD]}\ \frac{c,\ell : p = \&q \rightarrow fld \quad \ell' \xrightarrow{q} \ell \quad \langle c,\ell',q\rangle \hookleftarrow (c',\widehat{o})}{\langle c,\ell,p\rangle \hookleftarrow (c',\widehat{o.fld})} \qquad \texttt{[C-LOAD]}\ \frac{c,\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle c,\ell'',q\rangle \hookleftarrow (c',\widehat{o}) \quad \ell' \xrightarrow{o} \ell}{\langle c,\ell,p\rangle \hookleftarrow \langle c',\ell',o\rangle}$$

$$\texttt{[C-STORE]}\ \frac{c,\ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle c,\ell'',p\rangle \hookleftarrow (c',\widehat{o}) \quad \ell' \xrightarrow{q} \ell}{\langle c',\ell,o\rangle \hookleftarrow \langle c,\ell',q\rangle} \qquad \texttt{[C-SU/WU]}\ \frac{c,\ell : *p = \_ \quad \ell' \xrightarrow{o} \ell \quad (c',o) \in \mathcal{O} \backslash \mathit{kill}(c,\ell,p)}{\langle c',\ell,o\rangle \hookleftarrow \langle c',\ell',o\rangle}$$

$$\texttt{[C-CALL]}\ \frac{\begin{array}{lll} c,\ell : \_ = q(\dots,r,\dots) & \boldsymbol{\mu}(o_j) & \ell'' \xrightarrow{q} \ell \ \langle c,\ell'',q\rangle \hookleftarrow (\_,\widehat{o_f}) \ \ \ell \xrightarrow{r} \ell' \ \ \ell \xrightarrow{o} \ell' \\ c',\ell' : f(\dots,r',\dots) & o_{i+1} = \boldsymbol{\chi}(o_i) & c = c' \ominus \ell \end{array}}{\langle c',\ell',r'\rangle \hookleftarrow \langle c,\ell,r\rangle \quad \langle c',\ell',o\rangle \hookleftarrow \langle c,\ell,o\rangle}$$

$$\texttt{[C-RET]}\ \frac{\begin{array}{lll} c,\ell : p = q(\dots) & o_{j+1} = \boldsymbol{\chi}(o_j) & \ell'' \xrightarrow{q} \ell \ \langle c,\ell'',q\rangle \hookleftarrow (\_,\widehat{o_f}) \ \ \ell' \xrightarrow{p'} \ell \ \ \ell' \xrightarrow{o} \ell \\ c',\ell' : ret_f\ p' & \boldsymbol{\mu}(o_i) & c' = c \oplus \ell \end{array}}{\langle c,\ell,p\rangle \hookleftarrow \langle c',p',\ell'\rangle \quad \langle c,\ell,o\rangle \hookleftarrow \langle c',\ell',o\rangle}$$

$$\texttt{[C-COMPO]}\ \frac{lv \hookleftarrow lv' \quad lv' \hookleftarrow lv''}{lv \hookleftarrow lv''} \qquad \mathit{kill}(c,\ell,p) = \begin{cases} \{(c',o')\} & \text{if } \mathit{pt}(\langle c,\ell,p\rangle) = \{(c',o')\} \wedge (c',o') \in \mathit{cxtSingletons} \\ \mathcal{O} & \text{else if } \mathit{pt}(\langle c,\ell,p\rangle) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Fig. 11: Flow- and context-sensitive SUPA analysis with demand-driven strong updates.

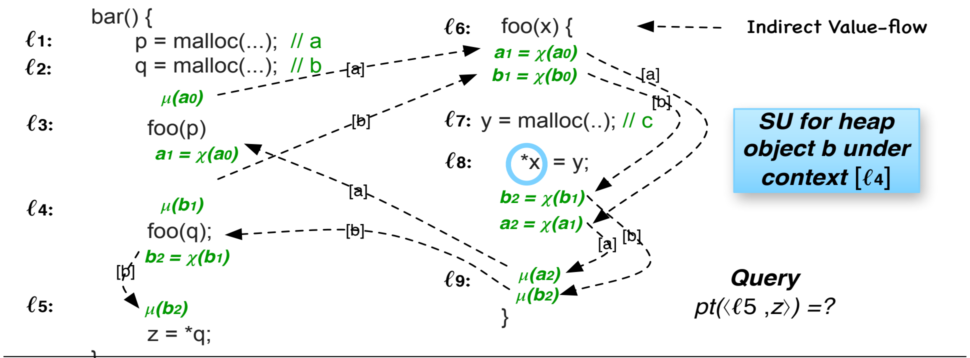


Fig. 12: Resolving $pt([\,],\ell_5,z) = \{[\ell_4],c\}$ with context-sensitive strong updates.

b points to previously (not shown to avoid cluttering). With context-sensitivity, SUPA finds that $\langle[\,],\ell_5,z\rangle \hookleftarrow \langle[\,],\ell_5,b\rangle \hookleftarrow \langle[\,],\ell_4,b\rangle \hookleftarrow \langle[\ell_4],\ell_9,b\rangle \hookleftarrow \langle[\ell_4],\ell_8,b\rangle \hookleftarrow \langle[\ell_4],\ell_7,y\rangle \hookleftarrow ([\ell_4],\widehat{c})$. Since $\langle[\ell_4],\ell_8,x\rangle$ points to a context-sensitive singleton (ℓ_4,b) at ℓ_8, a strong update is performed to b at ℓ_8, causing the old contents in b to be killed. □

4.3.1 Handling Recursion

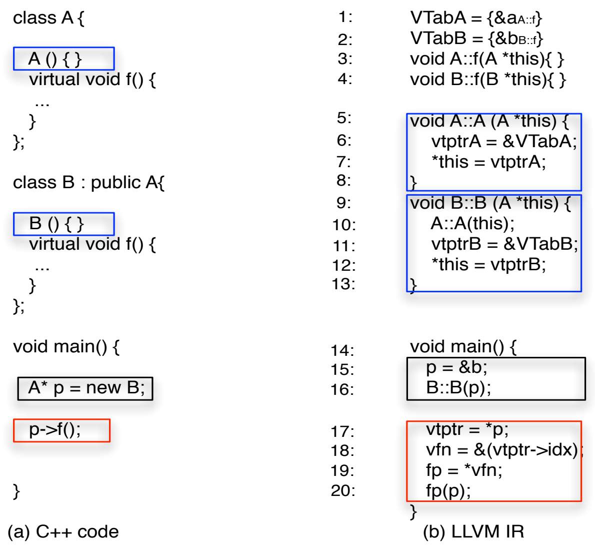
Context-sensitivity with recursions is undecidable [37], which must approximate to guarantee termination. One approach is to use k-limiting (i.e., the k-most-recent calling context leading to a callsite), which usually approximates more than unnecessary to achieve decidability for analyzing recursions [41].

Our handling of recursions uses SCC-based approach following the previous demand-driven pointer analysis (§Section 4.3, [41]). The SCC-based approach [28, 41, 55] applies k-limiting solution for call edges between SCCs, but conservatively treats the calls and returns inside an SCC context-insensitively by merging the context information inside an SCC (parameter passings and returns are treated as COPY instructions). This handling is only for SUPA's context-sensitive analysis and does not affect the precision of SUPA's flow-sensitive and context-insensitive analysis.

The SCC-based approach guarantees monotonicity during context-sensitive on-the-fly call graph construction, because (1) once a call graph edge is discovered during our graph traversal for answering a points-to query, the edge is added to the call graph but never removed when answering any subsequent query, (2) once an SCC is found during call graph construction, the functions inside the same SCC are collapsed into one, downgrading a previous context-sensitive call path inside an SCC (e.g., $cs_1, cs_2, ..., cs_n$) to context-insensitive one with every callsite cs_i being replaced with an special empty value ϵ which supports conservative unbalanced parentheses matching (ϵ matches any callsites at calls/returns) [41], thus merging multiple context-sensitive call paths into a context-insensitive one for over-approximations. The resulting points-to relations can only become less precise but never more precise if multiple functions are merged into an SCC during call graph construction.

Compared to the complete k-limiting approach (i.e., k-context-sensitivity for both recursive and non-recursive functions), our SCC-based approach is a less precise solution. However, as also mentioned in [41], the complete k-limiting approach is not favourable for demand-driven analysis since it requires tight analysis budgets for answering client queries. Repeatedly analyzing a recursion with a large k not only affects the analysis performance but may also cause an early termination inside the recursion after exhausting the analysis budget, resulting in context-insensitive analysis results across SCCs. Adjusting the analysis to a smaller k may accelerate the analysis for analyzing recursive functions, but unfortunately introduces imprecision for analyzing non-recursive functions.

Finding a good balance between efficiency and precision for handling recursions is an interesting future topic for demand-driven pointer analysis.



```
class A {

  A () { }
  virtual void f() {
   ...
  }
};

class B : public A{

  B () { }
  virtual void f() {
   ...
  }
};

void main() {

  A* p = new B;

  p->f();

}
```

```
1:   VTabA = {&aA::f}
2:   VTabB = {&bB::f}
3:   void A::f(A *this){ }
4:   void B::f(B *this){ }

5:   void A::A (A *this) {
6:     vtptrA = &VTabA;
7:     *this = vtptrA;
8:   }
9:   void B::B (A *this) {
10:    A::A(this);
11:    vtptrB = &VTabB;
12:    *this = vtptrB;
13:  }

14:  void main() {
15:    p = &b;
16:    B::B(p);

17:    vtptr = *p;
18:    vfn = &(vtptr->idx);
19:    fp = *vfn;
20:    fp(p);
     }
```

(a) C++ code (b) LLVM IR

Fig. 13: C++ code and its corresponding LLVM IR.

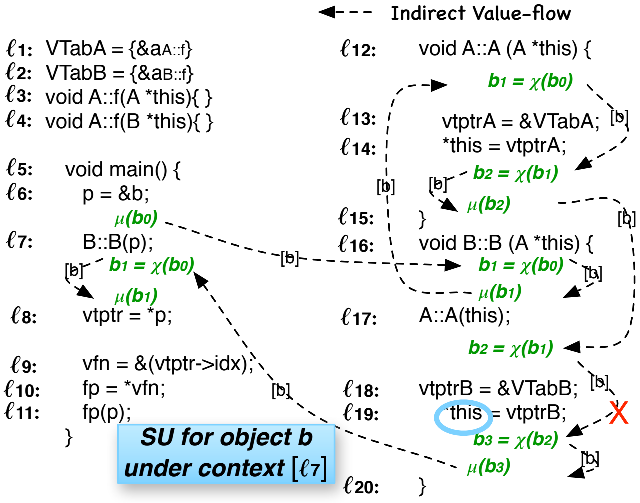


Fig. 14: SUPA precisely resolves that the virtual table pointer $vtptr$ at ℓ_8 points to only B's vtable $VTabB$ in Figure 13.

4.4 SUPA: Analyzing C++ Programs

Compared to C programs, the complicated features (e.g., inheritance, class object creations and virtual calls) in low-level object-oriented C++ programs pose big challenges to precise pointer analysis. Figure 13(a) gives a commonly used code fragment, including the constructors of two classes `A` and `B` (highlighted in the blue boxes), an object creation statement (black box) and a virtual call (red box). Figure 13(b) shows the corresponding LLVM IR.

In C++, base classes (e.g., A) and derived classes (e.g., B) can have virtual functions with the same function signature (i.e., name, argument types and qualifiers) for polymorphism (e.g., $f()$). For virtual functions $A :: f$ and $B :: f$ of class A and B, LLVM generates two virtual tables, i.e., (global constant arrays) $VTabA$ and $VTabB$ to store the function object $a_{A::f}$ (lines 1-2). The virtual table pointer (vtptr) which points to the vtable is stored into an object of class A when the object is created and initialized via a constructor of A (line 6 and 11). SUPA treats the global constant array $VTabA$ as a struct object with each element being a field of the struct for our field-sensitive analysis.

The object creation statement $A * p = newB$ is translated into two instructions, an ADDRESSOF that allocates object b (line 15), and a function call CALL that passes the pointer p into B's constructor (line 16), in which A's constructor is first called (line 10) to initialize b by storing the vtable pointer $vptrA$ into it (lines 6-7), then B's constructor updates b by storing $vptrB$ into it (lines 11-12).

The virtual call $p \rightarrow f()$ invokes the actual runtime function through dynamic dispatch, which is translated into the four LLVM instructions as also in Figure 13(b): (1) a LOAD at line 30 (obtain vtptr by dereferencing the pointer to the object, (2) an FIELD at line 31 (access the right function pointer in the vtable), (3) a LOAD at line 32 (read the virtual function pointer), and (4) an indirect function call CALL at line 33.

Figure 14 demonstrates our strong updates for resolving C++ virtual calls by using the same example from Figure 13 (b). To focus on the key idea of SUPA in analyzing C++, we simplify the example by issuing the query of vtable pointer $vtptr$ at ℓ_8 instead of pointer fp at line ℓ_{10} to show that SUPA can precisely resolves $vtptr$ pointing to only vtable $VTabB$ but not $VTabA$, so that the virtual call $p \rightarrow f()$ invokes $B :: f()$ only.

The pre-computed value-flows of object b are highlighted in dotted arrows based on the memory SSA form in Figure 14. Given query $vtptr$, we start our demand-driven analysis from ℓ_8 with its backtraces $\langle[\ ], \ell_8, vtptr\rangle \hookleftarrow \langle[\ ], \ell_7, b\rangle \hookleftarrow \langle[\ell_7], \ell_{20}, b\rangle \hookleftarrow \langle[\ell_7], \ell_{19}, b\rangle \hookleftarrow \langle[\ell_7], \ell_{19}, vtptrB\rangle \hookleftarrow \langle[\ell_7], \ell_{18}, \widehat{VTabB}\rangle$. We obtain that $vtptr$ points to $VTabB$ only. The strong update occurs at ℓ_{19} since $*this$ refers to a single memory location according to $\langle[\ell_7], \ell_{19}, this\rangle \hookleftarrow \langle[\ell_7], \ell_{16}, this\rangle \hookleftarrow \langle[\ ], \ell_7, p\rangle \hookleftarrow \langle[\ ], \ell_6, \widehat{b}\rangle$. Therefore, value-flow $\ell_{17} \xrightarrow{b} \ell_{19}$ is spurious, so that SUPA stops backtracking for object b from ℓ_{19} to ℓ_{17} due to strong updates.

5 IMPLEMENTATION

We have implemented SUPA in LLVM (4.0.0) and it is available at https://github.com/SVF-tools/SUPA. The source files of a program are compiled under "-O0" (to facilitate detection of undefined values [65]) into bit-code by `clang` and then merged using the `LLVM Gold Plugin` at link time to produce a whole program bc file. The compiler option `mem2reg` is applied to promote memory into registers. Otherwise, SUPA will perform more strong updates on memory locations that would otherwise be promoted to registers, favoring SUPA undesirably.

All the analyses evaluated are field-sensitive. Positive weight cycles that arise from processing fields of struct objects are collapsed [34]. Arrays are considered monolithic so that the elements in an array are not distinguished. Distinct allocation sites (i.e., ADDROF statements) are modeled by distinct abstract objects.

We build the SVFG for a program based on our open-source software, SVF [44]. The def-use chains are pre-computed by Andersen's algorithm flow and context-insensitively. In order to compute soundly and precisely the points-to information in a value-flow cycle, Supa retraverses the cycle whenever new points-to information is discovered until a fix point is reached.

To compare Supa with whole-program analysis, we have implemented a sparse flow-sensitive (SFS) analysis described in [20] also in LLVM, as SFS is a recent solution yielding exactly the flow-sensitive precision with good scalability. However, there are some differences. In [20], SFS was implemented in LLVM (2.5.0), by using imprecisely pre-computed call graphs and representing points-to sets with binary decision diagrams (BDDs). In this paper, just like Supa, SFS is implemented in LLVM (4.0.0), by building a program's call graph on the fly (Section 4.2) and representing points-to sets with sparse bit vectors.

We have not implemented a whole-program FSCS pointer analysis in LLVM. There is no open-source implementation either in LLVM. According to [1], existing FSCS algorithms for C "do not scale even for an order of magnitude smaller size programs than those analyzed" by Andersen's algorithm. As shown here, SFS can already spend hours on analyzing some programs under 500 KLOC.

6 Evaluation

Our evaluation addresses the following research questions:

RQ1 How to help analysis developers to effectively validate the correctness of various implementations (e.g., flow-insensitive, flow-sensitive and context-sensitive implementations of Supa) for C and C++ programs?

RQ2 How does Supa compare to the existing whole-program analysis SFS for analyzing large-scale programs in terms of precision and scalability when using a practical client?

RQ3 What is the precision improvement of a flow- and context-sensitive version Supa-FSCS of Supa over its flow-sensitive and context-insensitive one Supa-FS?

RQ4 How effective is Supa in analyzing large C++ programs, especially for resolving low-level virtual tables?

RQ5 What are the real code scenarios (snippets) that demonstrate the strong updates found by Supa?

6.1 RQ1: C/C++ Micro-Benchmarks for Pointer Analysis

We have designed PtaBen, an open and comprehensive micro-benchmark suite with around 400 test cases (including hand-written ones and the ones extracted from real programs) to evaluate the precision and correctness of pointer analysis (e.g., Supa) for both C and C++ programs. PtaBen is available at https://github.com/SVF-tools/PTABen.

6.1.1 Design of PTABen

Our benchmarks are designed to separate analysis validation as an independent concern from pointer analysis implementations based on any compiler platforms (e.g., LLVM/GCC). Each test case is developed to cover a program feature to validate the soundness and precision of a pointer analysis implementation. The cases are designed to test both basic analysis (e.g., Andersen's flow-insensitive and field-sensitive analysis) and sophisticated analysis (e.g., flow-sensitive and/or context-sensitive analyses).

For each micro-benchmark, we manually insert one or more stub functions (MAYALIAS or NOALIAS with each having two pointers as its parameters). These sub functions are treated as correct alias results (test oracle) to match against the results from a pointer analysis implementation. For example, MAYALIAS(p,&a) inserted at a particular program point in a micro-benchmark represents that `p` may point to object `a` at that program point, while NOALIAS(p,&a) means `p` does not point to `a`.

The following code snippet shows a simple case `struct-simple.c` in PtaBen to validate field-sensitivity. It directly enforces the correct pointer/alias result, i.e., the pointer `s2.a` points to only `x` but not `y` using the two stub functions MAYALIAS or NOALIAS.

```
struct s{
        int *a;
        int b;
};
int main(){
        struct s s1, s2; int x, y;
        s1.a = &x;
        s2.a = s1.a;
        MAYALIAS(s2.a, &x);
        NOALIAS(s2.a, &y);
}
```

PtaBen is simple to use. Given a test case annotated with the correct alias results, a pointer analysis implementation (e.g., Supa) can query the points-to results of the two parameters at the callsite of a stub function by comparing against the correct alias result to validate the precision and soundness of a pointer analysis implementation.

The stub functions can also be used to test function pointers. For example, the following code from `funptr-simple.c` validates whether a function pointer `fptr` points to function `f` (i.e., the indirect call transfer from the callsite `fptr(p)` to `f`) by using the stub function MAYALIAS to test the alias relation between the parameter `p` and the global object `&g`.

```
int g;
void (*fptr)(int*p);
void f(int* p) {   MAYALIAS(&g,p);   }
int main() {
          fptr=f;
          ...
          int *p = &g;
          fptr(p);
}
```

6.1.2 Testing C and C++ program features

Table 3 lists the basic test cases that cover important analysis aspects to validate pointer analysis (Andersen's field-sensitive implementation) for both C and C++ programs.

There are four categories of the basic C test cases designed to validate both intra- and inter-procedure analyses.

- `Stack & Heap` is to validate the analysis results of pointers point to local and heap objects, including variables defined and used in control-flow branches, pointer arithmetics for arrays and structs, test cases that form constraint cycles, dynamically allocated linked lists and integer to pointer castings.
- `Global` is to validate the analysis results of global variables, including global initializations, global arrays and structures, global variables modified and/or used

TABLE 3: Basic test cases in PTABEN to validate pointer analysis for C and C++.

Category	Test Case Name	Description
Basic Test Cases to Validate Pointer Analysis for C Stack & Heap	branch-intra.c	Testing aliases at a joint point from two branches on CFG
	branch-call.c	Testing aliases in a callee function which called from two callsites
	constraint-cycle-copy.c	Testing aliases in the presence of copy constraint cycles
	constraint-cycle-field.c	Testing aliases in the presence of positive weight cycles
	array-constIdx.c	Testing aliases between two array elements with constant indexes
	array-varIdx.c	Testing aliases between two array elements with variable indexes
	field-ptr-arith-constIdx.c	Testing aliases between fields of a struct using pointer arithmetics with constants
	field-ptr-arith-varIdx.c	Testing aliases between fields of a struct using pointer arithmetics with variables
	heap-indirect.c	Testing aliases between two dereferences referring to heap objects
	heap-linkedlist.c	Testing aliases of two elements in the dynamically allocated linked list
	heap-wrapper.c	Testing aliases of objects returned from wrapper functions of a heap allocation site
	int2pointer.c	Testing aliases of two pointers with one casted from an integer
Global	global-initializer.c	Testing global variables with their initializers
	global-array.c	Testing global array objects
	global-call-struct.c	Testing global struct objects
	global-call-noparam.c	Testing aliases of two global variables through calls which have no parameters
	global-call-twoparms.c	Testing fields of a global struct object through calls which have two parameters
	global-const-struct.c	Testing global constant struct objects
	global-funptr.c	Testing function pointers residing in a global struct object
	global-nested-calls.c	Testing aliases of two global variables through nested calls
FunPtr	funptr-simple.c	Testing function pointers on one branch of the CFG
	funptr-global.c	Testing function pointers which are global variables
	funptr-nested-call.c	Testing function pointers which are passed through nested calls
	funptr-struct.c	Testing function pointers stored in local struct objects
Struct	struct-array.c	Testing aliases of elements in an array field of a struct
	struct-assignment-direct.c	Testing aliases between two field access (st.fld) given a struct assignment
	struct-assignment-indirect.c	Testing aliases between two dereferences (st→fld) given a struct assignment
	struct-assignment-nested.c	Testing struct assignments in the presence of nested structs
	struct-field-multi-deref.c	Testing a pointer point to the inner field of a struct via multiple field dereferences
	struct-incompab-typecast.c	Testing type casting between pointers of incompatible struct types
	struct-incompab-typecast-n.c	Testing type casting between pointers of incompatible (nested) struct types
	struct-instance-return.c	Testing aliases when returning a struct instance from a callee function
	struct-nested-1-layer.c	Testing fields of a struct object nested in another struct
	struct-nested-2-layers.c	Testing fields of a struct object nested via two nested struct
	struct-nested-array.c	Testing array fields of a nested struct object
	struct-simple.c	Testing field aliases of a simple struct object
	struct-onefld.c	Testing a struct object with only one field
	struct-twoflds.c	Testing a struct object with only two fields
Basic Test Cases to Validate Pointer Analysis for C++	constructor.cpp	Testing a virtual call residing in a constructor of a child class
	destructor.cpp	Testing a virtual call residing in a destructor of a parent class
	dynamic_cast.cpp	Testing aliases of "dynamic_cast" (down casting) in C++
	global-obj-in-array.cpp	Testing aliases inside a virtual function in the presence of a global array of structs
	member-variable.cpp	Testing aliases between a global object and a member variable of a class object
	func-ptr-in-class.cpp	Testing virtual calls in the presence of global pointers
	single-inheritance.cpp	Testing virtual calls in the presence of the single-inheritance pattern
	multi-inheritance.cpp	Testing virtual calls in the presence of the multi-inheritance pattern
	diamond-inheritance.cpp	Testing virtual calls in the presence of the diamond inheritance pattern
	vdiamond-multi-inher.cpp	Testing virtual calls in the presence of virtual diamond inheritance pattern
	vector-field-sensitivity	Testing the points-to result of a field of an element stored in a vector
	forward_list.cpp	Testing the virtual call of a object stored in a C++ forward_list
	list.cpp	Testing the virtual call of a object stored in a C++ list
	map.cpp	Testing the virtual call of a object stored in a C++ map
	queue.cpp	Testing the virtual call of a object stored in a C++ queue
	set.cpp	Testing the virtual call of a object stored in a C++ set
	deque.cpp	Testing the virtual call of a object stored in a C++ deque
	vector.cpp	Testing the virtual call of a object stored in a C++ vector
	array.cpp	Testing aliases between two objects in a C++ array (e.g., array⟨const A *, 2⟩)
	unordered_map.cpp	Testing the virtual call of a object stored in a C++ unordered_map
	unordered_set.cpp	Testing the virtual call of a object stored in a C++ unordered_set
	stack.cpp	Testing the virtual call of a object stored in a C++ stack

inside a function or passed as parameters through nested calls.

- FunPtr is to test function pointers, including function pointers which are global variables or fields of a struct, and an indirect call whose function pointer is passed via nested calls before reaching the indirect callsite.
- Struct gives the test cases for analyzing struct objects, which are particularly useful for validating field-sensitive analysis implementations. For example, testing aliases between elements in an array field of a struct, testing aliases between two struct fields after a struct assignment or receiving a struct instance from a function return, and testing aliases in the presence of incompatible (struct) type castings, nested structs, structs of arrays and arrays of structs.

We have also provided C++ test cases to validate pointer analysis for various C++ language features, including constructors, destructors, virtual calls, single and multiple inheritances, virtual diamond inheritances and C++ STL library functions. Due to space limitation, only representative C++ cases are listed in Table 3. All C++ tests are available in the basic_cpp_tests folder in PTABEN's Github project.

For example, the following code snippet is from test case `single-inheritance.cpp`, which validates the virtual call `pb->f(ptr)` whose callee function is `B::f(int *i)`, where `B` is a subclass of class `A`.

```
int global_obj; int* global_ptr=&global_obj;
class A {
    virtual void f(int *i) {}
};
class B: public A {
    virtual void f(int *i) {
        MAYALIAS(global_ptr, i);
    }
};
int main(int argc, char **argv){
  int *ptr = &global_obj;
  A *pb = new B;
  pb->f(ptr);
}
```

Our tests are also able to validate the field-sensitivity in the presence of C++ containers. The following code snippet is from `vector-field-sensititivity.cpp`, which first pushes an object `c_` of class `C` into a vector `g`, and then retrieves the object from the vector (via C++ reference) to validate the aliases between the two fields of this object.

```
class C {
        int f1;
        int f2;
};
vector<C> g;
int main(int argc, char *argv[]) {
    C c_;
    g.push_back(c_);
    C &c = g[0];
    NOALIAS(&c.f1, &c.f2);
}
```

6.1.3 *Validating flow-sensitive and context-sensitive analyses*

PTABEN provides micro-benchmarks to support validating flow-sensitive analysis (e.g., SUPA-FS) and flow- and context-sensitive analysis (e.g., SUPA-FSCS). These two sets of test cases are placed in the folders `fs_tests` and `cs_tests` respectively in the PTABEN project.

In addition to the basic test cases in Table 3, `fs_tests` aims to validate flow-sensitive analysis with strong and weak updates for stack and global variables, including strong updates for global objects across multiple functions, strong updates in the presence of control-flow branches and function pointers. Strong and weak updates for struct and array objects. The following code snippet shows a global variable `q` which first points `y`, and then it is strongly updated to point to `x` in `bar`.

```
int x, y; int *p = &x; int *q = &y;
int **pp = &p; int**qq = &q;
void foo() {
        NOALIAS(*pp, *qq);
}
void bar() {
        q = &x;
}
int main() {
        foo();
        bar();
        MUSTALIAS(*pp, *qq);
}
```

`cs_tests` aims to validate flow- and context-sensitive analysis including one callee function is called via multiple callsites inside the same or different caller functions, strong updates for heap variables (Figure 12), and strong updates in the presence of function recursions. The full test suite for context-sensitive analysis is available in folder `cs_tests` in PTABEN. The following code snippet gives the correct points-to relations at different program points when program statements are involved in function recursions.

```
void f() {
  if (cond) {
         x = &y;
        MUSTALIAS(x,&y);
         f();
         x = &z;
        MUSTALIAS(x,&z);
        NOALIAS(x,&y);
         f();
  }
}
```

6.2 RQ2: Effectiveness and Precision of Our Demand-Driven Analysis

The objective is to show that SUPA is effective in answering client queries for real-world large-scale programs, in environments with small time and memory budgets.

We choose uninitialized pointer detection as a major client, named Uninit, which requires strong update analysis to be effective. We will show that SUPA can answer Uninit's on-demand queries efficiently while achieving nearly the same precision as SFS. We provide evidence to demonstrate a good correlation between the number of strong updates performed on-demand and the degree of precision achieved during the analysis.

6.2.1 *Methodology*

As a common type of bugs in C programs, uninitialized pointers are dangerous, as dereferencing them can cause system crashes and security vulnerabilities. For Uninit, flow-sensitivity is crucial. Otherwise, strong updates are impossible, making Uninit checks futile.

For C, global and static variables are default initialized, but local variables are not. In order to mimic the default uninitialization at a stack or heap allocation site $\ell : p = \&a$ for an uninitialized pointer a, we add a special store $*p = u$ immediately after ℓ, where u points to an *unknown abstract object* (UAO), u_a. Given a load $x = *y$, we can issue a points-to query for x to detect its potential uninitialization. If x points to a u_a (for some a), then x may be uninitialized. By performing strong updates more often, a flow-sensitive analysis can find more UAO's that do not reach any pointer and thus prove more pointers to be initialized. Note that SFS can yield false positives since, for example, path correlations are not modeled.

We do not introduce UAO's for the local variables involved in recursion and array objects since they cannot be strongly updated. We also ignore all the default-initialized stack or heap objects (e.g., those created by `calloc()`).

We generate meaningful points-to queries, one query for the top-level variable x at each load $x = *y$. However, we ignore this query if x is found not to point to any UAO by pre-analysis. This happens only when x points to either default-initialized objects or unmodeled local variables in recursion cycles or arrays. The number of queries issued in each program is listed in the last column in Table 4.

TABLE 4: Program characteristics.

Program	KLOC	Stmts	Pointers	Allocation Sites	Queries
spell-1.1	0.8	1011	1274	42	17
bc-1.06	14.4	17018	15212	654	689
milc-v6	15	11713	29584	865	3
less-451	27.1	6766	22835	1135	100
sed-4.2	38.6	25835	34226	395	1191
hmmer-2.3	36	27924	74689	1472	2043
make-4.1	40.4	14926	36707	1563	1133
gzip-1.6	64.4	22028	25646	1180	551
a2ps-4.14	64.6	49172	116129	3625	5065
bison-3.0.4	113.3	36815	90049	1976	4408
grep-2.21	118.4	10199	33931	1108	562
tar-1.28	132	30504	85727	3350	909
wget-1.16	140.0	51556	63199	726	1142
bash-4.3	155.9	59442	191413	6359	5103
gnugo-3.4	197.2	369741	286986	27511	1970
sendmail-8.15	259.9	86653	256074	7549	2715
vim-7.4	413.1	147550	466493	8960	6753
emacs-24.4	431.9	189097	754746	12037	4438
Total	2263.0	1157950	2584920	80507	38792

TABLE 5: Pre-processing times taken by pre-analysis shared by SUPA and SFS and analysis times of SFS (in seconds).

Program	Pre-Analysis Times Shared by SUPA and SFS			Analysis Time of SFS
	Andersen	SVFG	Total	
spell	0.01	0.01	0.01	0.01
bc	0.35	0.21	0.56	0.98
milc	0.42	0.1	0.52	0.16
less	0.42	0.37	0.79	1.94
sed	1.38	0.34	1.73	5.46
hmmer	1.57	0.46	2.03	1.07
make	1.74	1.17	2.91	13.94
gzip	0.27	0.10	0.37	0.20
a2ps	7.34	1.31	8.65	60.61
bison	8.18	3.66	11.84	44.16
grep	1.44	0.17	1.61	2.39
tar	2.73	1.71	4.44	12.27
wget	1.86	0.90	2.76	3.47
bash	53.48	44.07	97.55	2590.69
gnugo	5.68	2.75	8.44	9.86
sendmail	24.05	23.43	47.48	348.63
vim	445.88	85.69	531.57	13823
emacs	135.93	146.94	282.87	8047.55

6.2.2 Experimental Setup

We use a machine with a 3.7GHz Intel Xeon 8-core CPU and 64 GB memory. As shown in Table 4, we have selected a total of 18 open-source programs from a variety of domains: `spell-1.1` (a spelling checker), `bc-1.06` (a numeric processing language), `milc-v6` (quantum chromodynamics), `less-451` (a terminal pager), `sed-4.2` (a stream editor), `milc-v6` (quantum chromodynamics), `hmmer-2.3` (sequence similarity searching), `make-4.1` (a build automation tool), `a2ps-4.14` (a postScript filter), `bison-3.04` (a parser), `grep-2.2.1` (string searching), `tar-1.28` (tar archiving), `wget-1.16` (a file downloading tool), `bash-4.3` (a unix shell and command language), `gnugo-3.4` (a Go game), `sendmail-8.15.1` (an email server and client), `vim74` (a text editor), and `emacs-24.4` (a text editor).

For each program, Table 4 lists its number of lines of code, statements which are LLVM instructions relevant to our pointer analysis, pointers, allocation sites (or `AddrOf` statements), and queries issued (as discussed in Section 6.2.1).

We evaluate SUPA with two configurations, SUPA-FS and SUPA-FSCS. SUPA-FS is a one-stage FS analysis by considering flow-sensitivity only. SUPA-FSCS is a two-stage analysis consisting of FSCS and FS. The FSCS is first applied for answering the query. It will downgrade to a less precise FS one if the FSCS stage is out of analysis budget.

6.2.3 Evaluating SUPA-FS

When assessing SUPA-FS, we consider two different criteria: efficiency (its analysis time and memory usage per query) and precision (its competitiveness against SFS). For each query, its analysis budget, denoted B, represents the maximum number of def-use chains that can be traversed. We consider a wide range of budgets with B falling into $[10, 200000]$.

SUPA-FS is highly effectively. With $B = 10000$, SUPA-FS is nearly as precise as SFS, by consuming about 0.18 seconds and 65KB of memory per query, on average.

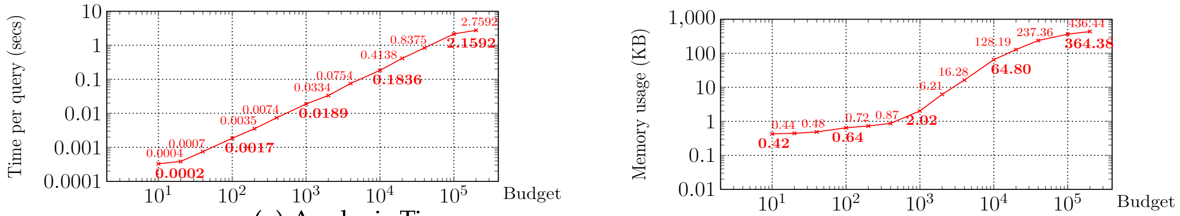
Efficiency: Figure 15(a) shows the average analysis time per query for all the programs under a given budget, with about 0.18 seconds when $B = 10000$ and about 2.76 seconds when $B = 200000$. Both axes are logarithmic. The longest-running queries can take an order of magnitude as long as the average cases. However, most queries (around 80% across the programs) take much less than the average cases. Take emacs for example. SFS takes over two hours (8047.55 seconds) to finish. In contrast, SUPA-FS spends less than ten minutes (502.10 seconds) when $B = 2000$, with an average per-query time (memory usage) of 0.18 seconds (0.12KB), and produces the same answers for all the queries as SFS (shown in Figure 16 and explained below).

For SUPA, its pre-analysis is lightweight, as shown in Table 5, with `vim` taking the longest at 531.57 seconds. The same pre-analysis is also shared by SFS in order to enable its own sparse whole-program analysis. The additional time taken by SFS for analyzing each program entirely is given in the last column.

Figure 15(b) shows the average memory usage per query under different budgets. Following the common practice, we measure the real-time memory usage by reading the virtual memory information (`VmSize`) from the linux kernel file (`/proc/self/status`). The memory monitor starts after the pre-analysis to measure the memory usage for answering queries only. The average amount of memory consumed per query is small, with about 65KB when $B = 10000$ and about 436KB when $B = 200000$. Even under the largest budget $B = 200000$ evaluated, SUPA-FS never uses more than 3MB for any single query processed.

Precision: Given a query $pt(\langle \ell, p \rangle)$, p is initialized if no UAO is pointed by p and potentially uninitialized otherwise. We measure the precision of SUPA-FS in terms of the percentage of queried variables proved to be initialized by comparing with SFS, which yields the best precision achievable as a whole-program flow-sensitive analysis.

Figure 16 reports our results. As B increases, the precision of SUPA-FS generally improves. With $B = 10000$, SUPA-FS can answer correctly 97.4% of all the queries from the 18 programs. These results indicate that our analysis is highly accurate, even under tight budgets. For the 18 programs except `a2ps`, `bison` and `bash`, SUPA-FS produces the same



(a) Analysis Time

(b) Memory Usage

Fig. 15: Average analysis time and memory usage per query consumed by SUPA-FS under different analysis budgets (with both axes being logarithmic).

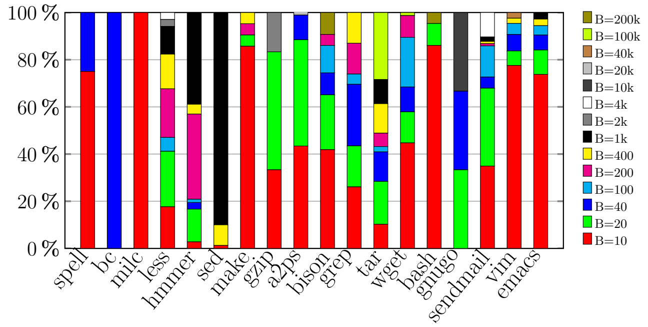


Fig. 16: Percentage of queried variables proved to be initialized by SUPA-FS over SFS under different budgets.

answers for all the queries when $B = 100000$ as SFS. When $B = 200000$ for these three programs, SUPA becomes as precise as SFS, by taking an average of 0.02 seconds (88.5KB) for `a2ps`, 0.25 seconds (194.7KB) for `bison`, and 3.18 seconds (1139.3KB) for `bash`, per query.

The results between whole-program flow-sensitive analysis SFS and SUPA always return identical results as long as the budget is large enough. This because both SFS and SUPA are implemented based on top of the same SVFG (sparse value-flow graph) and the call graphs are constructed the on-the-fly when discovering new callees. If there is an out of budget query, the results are correct since SUPA fails back to a less precise but conservative analysis (e.g., Andersen's analysis).

Understanding On-Demand Strong Updates: Let us examine the benefits achieved by SUPA-FS in answering client queries by applying on-demand strong updates. For each program, Figure 17 shows a good correlation between the number of strong updates performed (#SU on the left y-axis) in a blue curve and the number of UAO's reaching some uninitialized pointers (#UAO on the right y-axis) in a red curve under varying budgets (on the logarithmic x-axis). The number of such UAO's reported by SFS is shown as the lower bound for SUPA-FS in a dashed line.

In most programs, SUPA-FS performs increasingly more strong updates to block increasingly more UAO's to reach the queried variables as the analysis budget B increases, because SUPA-FS falls back increasingly less frequently from FS to the pre-computed points-to information. When B increases, SUPA-FS can filter out more spurious value-flows in the SVFG to obtain more precise points-to information, thereby enabling more strong updates to kill the UAO's.

When $B = 200000$, SUPA-FS gives the same answers as SFS in all the 18 programs except `bison` and `vim`, which causes SUPA-FS to report 16 and 35 more UAOs, respectively.

For some programs such as `spell`, `bc`, `milc`, `hmmer` and `grep`, most of their strong updates happen under small budgets (e.g., $B = 1000$). In `hmmer`, for example, 192 strong updates are performed when $B = 10000$. Of the 5126 queries issued, SUPA-FS runs out-of-budget for only three queries, which are all fully resolved when $B = 200000$, but with no further strong updates being observed.

For programs like `bison`, `bash`, `gnugo` and `emacs`, quite a few strong updates take place when $B > 1000$. There are two main reasons. First, these programs have many indirect call edges (with 8709 in `bison`, 1286 in `bash`, 23150 in `gnugo` and 4708 in `emacs`), making their on-the-fly call graph construction costly (Section 4.1.2). Second, there are many value-flow cycles (with over 50% def-use chains occurring in cycles in bison), making their constraint resolution costly (to reach a fixed point). Therefore, relatively large budgets are needed to enable more strong updates to be performed.

Interestingly, in programs such as `a2ps`, `gnugo` and `vim`, fewer strong updates are observed when larger budgets are used. In `vim`, the number of strong updates performed is 1492 when $B = 2000$ but drops to 1204 when $B = 4000$. This is due to the forward reuse described in Section 4.2 and [45, §4.3]. When answering a query $pt(\langle \ell, v \rangle)$ under two budgets B_1 and B_2, where $B_1 < B_2$, SUPA-FS has reached $\langle \ell', v' \rangle$ and needs to compute $pt(\langle \ell', v' \rangle)$ in each case. SUPA-FS may fall back to the flow-insensitive points-to set of v' under B_1 but not B_2, resulting in more strong updates performed under B_1 in the part of the program that is not explored under B_2.

6.3 RQ3: Evaluating Flow- and Context-Sensitive SUPA

For C programs, flow-sensitivity is regarded as being important for achieving useful high precision. However, context-sensitivity can be important for some C programs, in terms of both obtaining more precise points-to information and enabling more strong updates. Unfortunately, whole-program analysis does not scale well to large programs when both are considered (Section 5).

In this section, we demonstrate that SUPA can exploit both flow- and context-sensitivity effectively *on-demand* in a hybrid three-stage analysis framework, providing improved precision needed by some programs.

SUPA-FSCS answers a query on-demand by applying its three analyses successively, starting from flow- and context-sensitive analysis (FSCS). If the query is not answered after budget has been exhausted at the FSCS stage, SUPA-FS reissues the query for flow-sensitive but context-insensitive

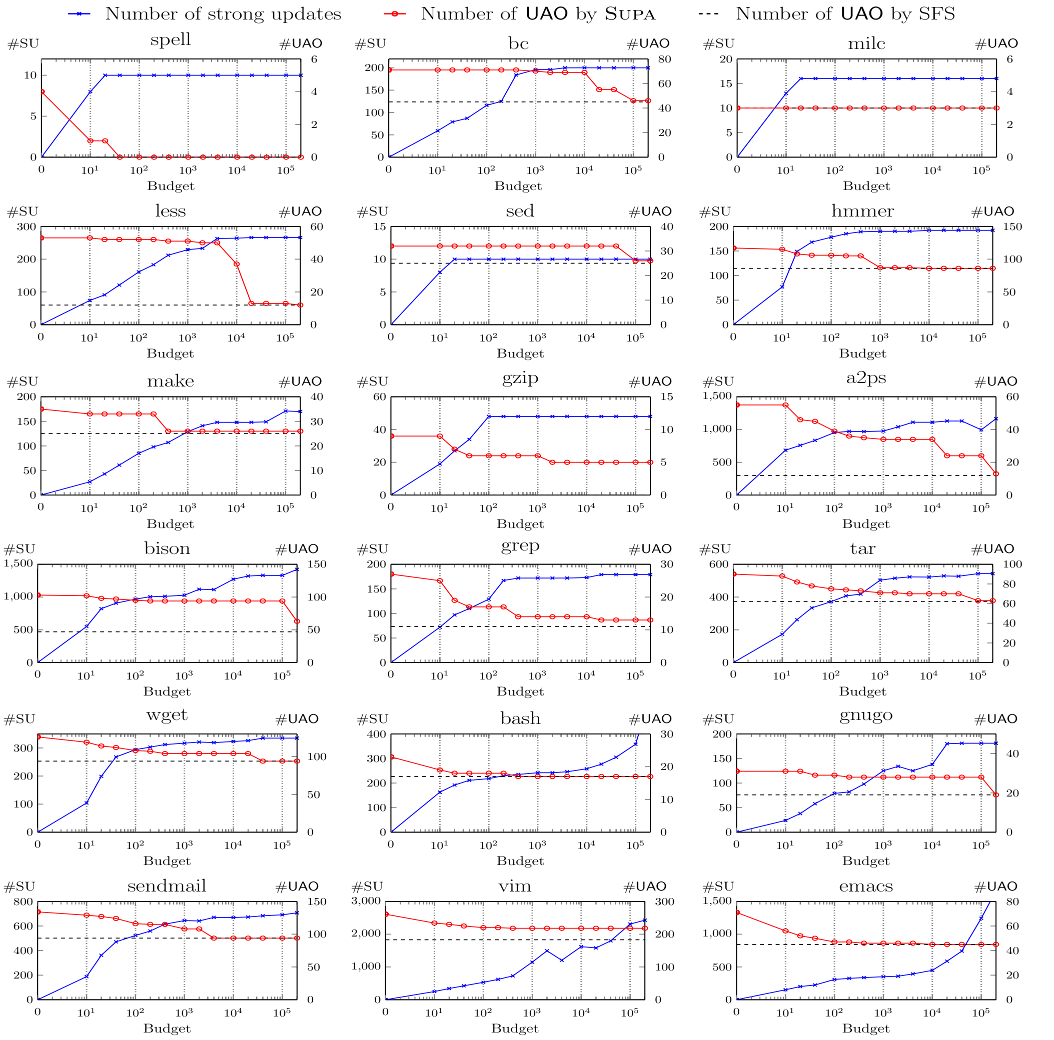


Fig. 17: Correlating the number of strong updates with the number of UAO's under SUPA-FS with different analysis budgets.

analysis (FSCI), and eventually falls back to the results precomputed by Andersen's analysis (FICI) if the budget is exhausted at FSCI stage.

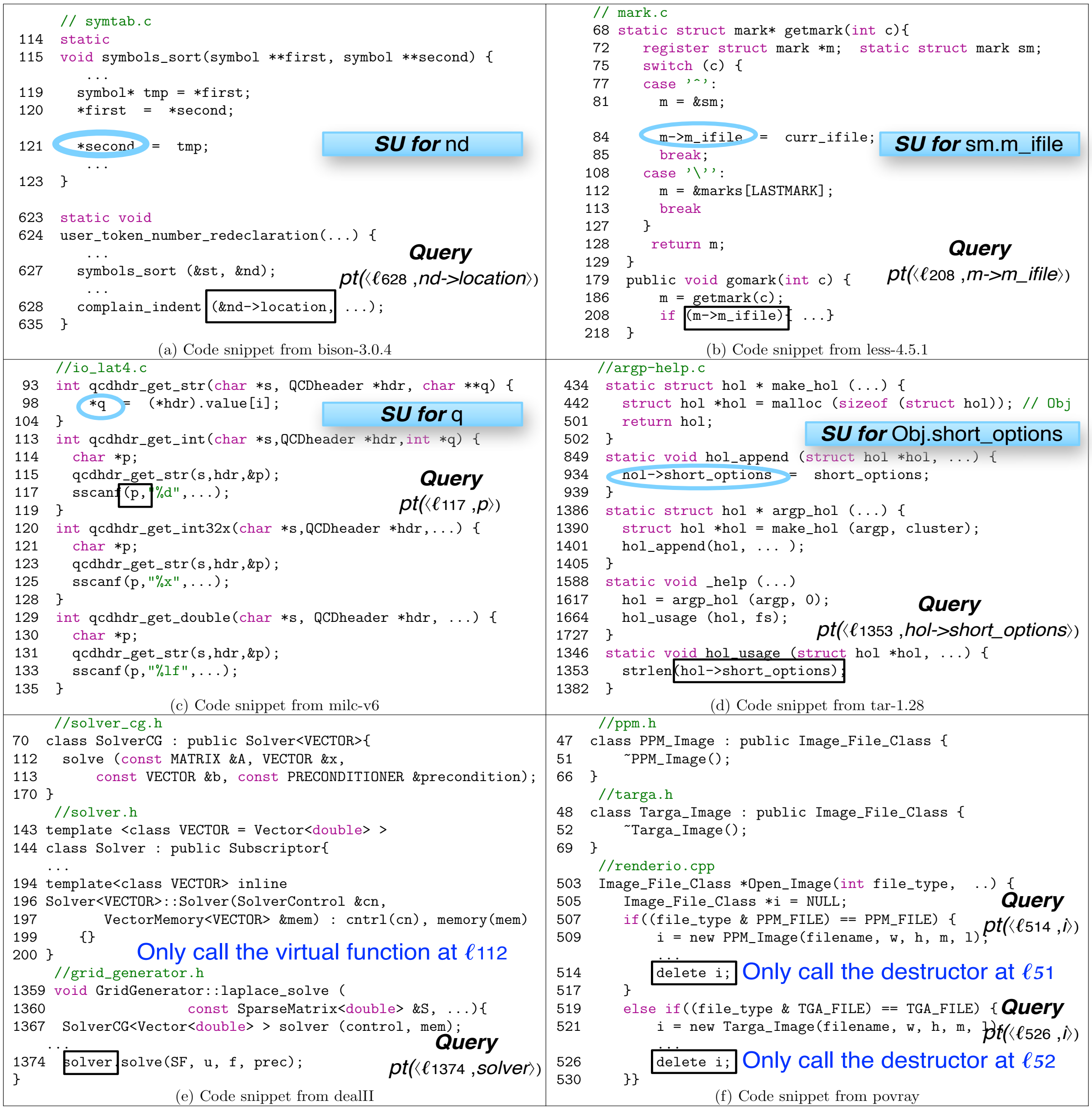
Table 6 compares SUPA-FSCS (with a budget of 20000 divided evenly in its FSCS and FS stages) with SUPA-FS (with a budget of 10000 in its single FS stage). The maximal depth of a context stack allowed is 3. By allocating the budgets this way, we can investigate some additional precision benefits achieved by considering both flow- and context-sensitivity.

In general, SUPA-FSCS has longer query response times than SUPA-FS due to the larger budgets used in our setting and the times taken in handling context-sensitivity. In `milc`, `hmmer`, `a2ps`, `bison`, `tar` , `gnugo` and `sendmail`, SUPA-FSCS reports fewer UAO's than SUPA-FS, for two reasons. First, SUPA-FSCS can perform strong updates context-sensitively for stack and global objects, resulting in 0 UAO's reported by SUPA-FSCS for milc. Second, SUPA-FSCS can perform strong updates to context-sensitive singleton heap objects defined in Section 4.3, by eliminating 8 UAO's in `bison`, 1 in `tar` and 1 in `sendmail`, which have been reported by SUPA-FS.

6.4 RQ4: Evaluating SUPA in Resolving C++ Virtual Calls

Precisely resolving virtual calls is a key challenge in analyzing low-level object-oriented language, such as C++. Precise virtual call resolution benefits for a wide range of other program analyses where a call graph is needed. Section 4.4 describes how SUPA analyzes virtual tables at callsites in the form of low-level LLVM IR. We compare SUPA-FS and SUPA-FSCS to demonstrate that our context-sensitive demand-driven analysis can precisely remove spurious virtual table targets which are produced by pointer analysis without strong updates for heap objects.

We evaluate SUPA using all SPEC2000/2006 C++ programs except `444.namd` and `473.astar`. Because `444.namd` has no virtual calls and `473.astar` has one virtual call with one target resolved by Andersen's analysis. As shown in Table 7, all the six CPU2000/2006 benchmarks evaluated are `eon`, `dealII`, `omentpp`, `povray`, `soplex`, and `xalan`. It also gives the analysis time per query by SUPA, and the total number of virtual callsite (`#Virtual callsite`). We ignore some trivial callsites that SUPA-FS has the same results as Andersen's analysis to fairly compare with SUPA-FSCS. `#More precise callsite` denotes the number of callsites where SUPA-FSCS get fewer virtual table targets than SUPA-FS. On average, SUPA-FSCS reduces 7.35% spurious targets with only 0.4 secs per query.



```
    // symtab.c
114 static
115 void symbols_sort(symbol **first, symbol **second) {
      ...
119   symbol* tmp = *first;
120   *first  =  *second;

121   *second  =  tmp;
      ...
123 }

623 static void
624 user_token_number_redeclaration(...) {
      ...
627   symbols_sort (&st, &nd);
      ...
628   complain_indent (&nd->location, ...);
635 }
```

(a) Code snippet from bison-3.0.4

```
    // mark.c
68  static struct mark* getmark(int c){
72     register struct mark *m;  static struct mark sm;
75     switch (c) {
77     case '^':
81       m = &sm;

84       m->m_ifile  =  curr_ifile;
85       break;
108    case '\'':
112      m = &marks[LASTMARK];
113      break
127    }
128     return m;
129  }
179  public void gomark(int c) {
186      m = getmark(c);
208      if (m->m_ifile){ ...}
218  }
```

(b) Code snippet from less-4.5.1

```
    //io_lat4.c
 93 int qcdhdr_get_str(char *s, QCDheader *hdr, char **q) {
 98     *q  =  (*hdr).value[i];
104 }
113 int qcdhdr_get_int(char *s,QCDheader *hdr,int *q) {
114   char *p;
115   qcdhdr_get_str(s,hdr,&p);
117   sscanf(p,"%d",...);
119 }
120 int qcdhdr_get_int32x(char *s,QCDheader *hdr,...) {
121   char *p;
123   qcdhdr_get_str(s,hdr,&p);
125   sscanf(p,"%x",...);
128 }
129 int qcdhdr_get_double(char *s, QCDheader *hdr, ...) {
130   char *p;
131   qcdhdr_get_str(s,hdr,&p);
133   sscanf(p,"%lf",...);
135 }
```

(c) Code snippet from milc-v6

```
     //argp-help.c
434  static struct hol * make_hol (...) {
442    struct hol *hol = malloc (sizeof (struct hol)); // Obj
501    return hol;
502  }
849  static void hol_append (struct hol *hol, ...) {
934    hol->short_options  =  short_options;
939  }
1386 static struct hol * argp_hol (...) {
1390   struct hol *hol = make_hol (argp, cluster);
1401   hol_append(hol, ... );
1405 }
1588 static void _help (...)
1617   hol = argp_hol (argp, 0);
1664   hol_usage (hol, fs);
1727 }
1346 static void hol_usage (struct hol *hol, ...) {
1353   strlen(hol->short_options);
1382 }
```

(d) Code snippet from tar-1.28

```
     //solver_cg.h
70   class SolverCG : public Solver<VECTOR>{
112   solve (const MATRIX &A, VECTOR &x,
113       const VECTOR &b, const PRECONDITIONER &precondition);
170 }
     //solver.h
143 template <class VECTOR = Vector<double> >
144 class Solver : public Subscriptor{
    ...
194 template<class VECTOR> inline
196 Solver<VECTOR>::Solver(SolverControl &cn,
197        VectorMemory<VECTOR> &mem) : cntrl(cn), memory(mem)
199     {}
200 }
     //grid_generator.h
1359 void GridGenerator::laplace_solve (
1360                   const SparseMatrix<double> &S, ...){
1367  SolverCG<Vector<double> > solver (control, mem);
    ...
1374  solver.solve(SF, u, f, prec);
}
```

(e) Code snippet from dealII

```
    //ppm.h
47  class PPM_Image : public Image_File_Class {
       ~PPM_Image();
66  }
    //targa.h
48  class Targa_Image : public Image_File_Class {
52     ~Targa_Image();
69  }
    //renderio.cpp
503  Image_File_Class *Open_Image(int file_type,  ..) {
505    Image_File_Class *i = NULL;
507    if((file_type & PPM_FILE) == PPM_FILE) {
509        i = new PPM_Image(filename, w, h, m, l);
           ...
514        delete i;
517    }
519    else if((file_type & TGA_FILE) == TGA_FILE) {
521        i = new Targa_Image(filename, w, h, m, l);
           ...
526        delete i;
530    }}
```

Query
pt(⟨ℓ514 ,i⟩)
Only call the destructor at ℓ51
Query
pt(⟨ℓ526 ,i⟩)
Only call the destructor at ℓ52

(f) Code snippet from povray

Fig. 18: Selected code snippets.

6.5 RQ5: Case Studies

We examine some real code to see how client queries are answered precisely for the two major clients, i.e., `Unint` and C++ virtual table resolution. Figure 18 (a) - (d) show four different scenarios under `Unint` client, while Figure 18 (e) and (f) give two code snippets to demonstrate the benefit of SUPA in precise virtual call resolution.

Figure 18(a) There is a swap from `bison`. In line 121, `second` points to a singleton stack object `nd` passed from line 627. Therefore, a strong update is applied. When querying `nd->location` in line 628, SUPA knows that `nd` points to what `st` pointed to before.

Figure 18(b) In the code fragment from `less`, `m->m_ifile` is initialized in two different branches, one recognized due to a strong update performed at the store in line 84 and one due to the default initialization in line 112. According to SUPA, `m->m_ifile` in line 208 is initialized.

Figure 18(c) In the code fragment from `milc`, `q` in line 98 can point to several stack variables that are all named p in lines 115, 123 and 131. With context-sensitivity, SUPA finds that `q` points to one singleton under each context. Thus, a strong update is performed so that each stack variable becomes properly initialized when queried at each call to `sscanf()`.

Figure 18(d) In the code fragment from `tar`, `hol` in line 1390 points to a heap object o allocated in line 442. With o treated as a context-sensitive singleton (requiring a context stack of at least depth 1), a strong update can be performed in line 934 to initialize its field `short_options` properly.

Figure 18(e) This code fragment from `dealIII` is very similar to the example in Figure 13. `solver` created at line 1367 is an object of template class `SolverCG` (line 70), which is a subclass of `Solver` (line 196). SUPA's strong update analysis precisely identifies that the virtual call at line 1374 only calls the virtual function `solve` in `SolverCG` instead of the one in superclass `Solver`.

TABLE 6: Average analysis times consumed and UAO's reported by Supa-FSCS (with a budget of 10000 in each stage) and Supa-FS (with a budget of 10000 in total).

Program	Supa-FS		Supa-FSCS	
	Time (ms)	#UAO	Time (ms)	#UAO
spell	0.01	0	0.01	0
bc	18.35	69	287.23	69
milc	0.02	3	14.52	**0**
less	15.15	37	92.41	37
sed	355.60	32	4725.42	32
hmmer	11.41	86	135.05	**71**
make	124.40	26	229.44	26
gzip	0.64	5	4.28	5
a2ps	126.01	34	448.26	**32**
bison	465.54	94	529.20	**86**
grep	124.46	14	197.66	14
tar	26.31	70	83.10	**68**
wget	24.51	104	84.90	104
bash	188.69	17	327.16	17
gnugo	72.73	28	80.08	**27**
sendmail	200.32	94	250.19	**85**
vim	168.67	218	473.25	218
emacs	159.22	45	222.65	45

TABLE 7: Virtual table resolution at virtual callsites. Supa-FS v.s. Supa-FSCS (with a budget of 10000 in its FSCS stage and 10000 in its FSCI stage).

Program	Time per query (ms)	#Virtual callsite	#More precise callsite	Improved callsite (%)
eon	272	68	2	2.94%
dealII	199	404	34	8.41%
omnetpp	831	514	33	6.42%
povray	410	120	29	24.16%
soplex	318	421	2	0.48%
xalan	427	7388	128	1.73%
average	409	-	-	7.35%

Figure 18(f) This code from `povray` demonstrates Supa's analysis in precisely analyzing C++ destructors. Supa successfully identifies that the virtual call at 514 only calls the destructor ~`PPM_Image()` (line 51) due to strong updates in C++ object initialization (as explained in Section 4.4), i.e., `i` points to an object of class `PPM_Image` which is a subclass of `Image_File_Class` (line 509). Likewise, Supa identifies precisely the virtual call at line 526.

7 RELATED WORK

Demand-driven and whole-program approaches represent two important solutions to long-standing pointer analysis problems. While a whole-program pointer analysis aims to resolve all the pointers in the program, a demand-driven pointer analysis is designed to resolve only a (typically small) subset of the set of these pointers in a client-specific manner. This work is not concerned with developing an ultra-fast whole-program pointer analysis. Rather, our objective is to design a staged demand-driven strong update analysis framework that facilitates efficiency and precision tradeoffs flow- and context-sensitively according to the needs of a client (e.g., user-specified budgets). We limit our discussion to the work that is most relevant to Supa.

7.1 Flow-Sensitive Pointer Analysis

Strong updates require pointers to be analyzed flow-sensitively with respect to program execution order. Whole-program flow-sensitive pointer analysis has been studied extensively in the literature. [10] and [14] gave some formulations in an iterative data-flow framework [24]. [56] considered both flow- and context-sensitivity by representing procedure summaries with partial transfer functions, but restricted strong updates to top-level variables only. To eliminate unnecessary propagation of points-to information during the iterative data-flow analysis [19, 20, 33, 62], some form of sparsity has been exploited. The sparse value-flows, i.e., def-use chains in a program are captured by sparse evaluation graphs (SEG) [9, 36] as in [22] and various SSA representations such as HSSA [11], partial SSA [27] and SSI [2, 53]. The def-use chains for top-level pointers, once put in SSA, can be explicitly and precisely identified, giving rise to a so-called semi-sparse flow-sensitive analysis [19]. Later, the idea of staged analysis [17] has been leveraged to make pointer analysis full-sparse for both top-level and address-taken variables by using fast Andersen's analysis as precise analysis [20, 49, 61]. This paper is the first to exploit sparsity to improve the performance of a flow- and context-sensitive demand-driven analysis with strong updates being performed for C programs.

Recently, Balatsouras and Smaragdakis [6] propose a fine-grained field-sensitive modeling technique for performing Andersen's analysis by inferring lazily the types of heap objects in order to filter out redundant field derivations. This technique can be exploited to obtain a more precise pre-analysis to improve the precision and/or efficiency of sparse flow-sensitive analysis.

7.2 Demand-Driven Pointer Analysis

Demand-driven pointer analyses for C [21, 63, 66] and Java [31, 39, 41, 43, 57] are flow-insensitive, formulated in terms of CFL (Context-Free-Language) reachability [38]. [21] introduced the first on-demand Andersen-style pointer analysis for C. Later, [66] performed alias analysis for C in terms of CFL-reachability flow- and context-insensitively with indirect function calls handled conservatively. Sridharan et al. gave two CFL-reachability-based formulations for Java, initially without considering context-sensitivity [42] and later with context-sensitivity [41]. [39] and [57] investigated how to summarize points-to information discovered during the CFL-reachability analysis to improve performance for Java programs. [16] focused on answering demand queries for Java programs in a context-sensitive analysis framework (without performing strong updates). Boomerang [40] represents a very recent flow- and context-sensitive demand-driven pointer analysis for Java. However, its access-path-based analysis performs only strong updates partially at a store $a.f = \dots$, by updating $a.f$ strongly but the aliases of $a.f.*$ weakly. This paper presents Supa, which focuses on performing strong updates on-demand flow and context-sensitively for analyzing C and C++ programs with two practical clients.

7.3 Hybrid Pointer Analysis

The basic idea is to find a right balance between efficiency and precision. For C programs, the one-level approach [13] achieves a precision between Steensgaard's and Andersen's analyses by applying a unification process to address-taken variables only. In the case of Java programs, context-sensitivity can be made more effective by considering both call-site-sensitivity and object-sensitivity together than either alone [25]. In [18], how to adjust the analysis precision according to a client's needs is discussed. [64] focus on finding effective abstractions for whole-program analyses written in Datalog via abstraction refinement. Lhoták and Chung [29] trades precision for efficiency by performing strong updates only on flow-sensitive singleton objects but falls back to the flow-insensitive points-to information otherwise. In this paper, we propose to carry out our on-demand strong update analysis. Unlike [29], SUPA can achieve the same precision as whole-program flow-sensitive analysis, subject to a given budget.

8 Conclusion

We have introduced, SUPA, a demand-driven pointer analysis that enables computing precise points-to information for C and C++ programs flow- and context-sensitively with strong updates by refining away imprecisely pre-computed value-flows, subject to some analysis budgets. SUPA is suitable for environments with small time and memory budgets such as IDEs. We have evaluated SUPA by choosing uninitialized pointer detection and virtual call resolution as major clients on 18 C and 6 C++ programs. SUPA can achieve nearly the same precision as whole-program flow-sensitive analysis under small budgets and can outperform the precision of traditional flow-sensitive analysis by performing strong updates for field and heap objects.

References

[1] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE '11*, pages 746–755, 2011. URL http://doi.acm.org/10.1145/1985793.1985898.

[2] C. S. Ananian. *The static single information form*. PhD thesis, Master's Thesis, MIT, 1999.

[3] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI '14*, pages 259–269, 2014. doi: 10.1145/2666356.2594299. URL http://doi.acm.org/10.1145/2666356.2594299.

[5] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *International Conference on Software Engineering (ICSE '05)*, pages 332–341. ACM, 2005.

[6] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *SAS '16*, 2016.

[7] M. Barbar, Y. Sui, H. Zhang, S. Chen, and J. Xue. Live path control flow integrity. In *ICSE '18*, pages 195–196. ACM, 2018.

[8] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. In *PLDI '13*, pages 275–286, 2013.

[9] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91*, pages 55–66, 1991.

[10] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93*, pages 232–245, 1993.

[11] 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.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS'91*, 13(4):490, 1991.

[13] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI '00*, pages 35–46, 2000. ISBN 1-58113-199-2. doi: 10.1145/349299.349309. URL http://doi.acm.org/10.1145/349299.349309.

[14] R. Emami, M. Ghiya and J. Hendren. Context-sensitive interprocedural points-to analysis in presence of function pointers. In *PLDI '94*, pages 242–256, 1994.

[15] X. Fan, Y. Sui, X. Liao, and J. Xue. Boosting the precision of virtual call integrity protection with partial pointer analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 329–340, 2017. doi: 10.1145/3092703.3092729. URL http://doi.acm.org/10.1145/3092703.3092729.

[16] Y. Feng, X. Wang, I. Dillig, and C. Lin. EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. In *OOPSLA '15*, pages 520–534, 2015. doi: 10.1145/2814270.2814284. URL http://doi.acm.org/10.1145/2814270.2814284.

[17] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17 (2):9, 2008.

[18] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *SAS '03*, pages 1073–1073, 2003.

[19] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238, 2009.

[20] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298, 2011.

[21] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI '01*, pages 24–34, 2001. doi: 10.1145/381694.378802. URL http://doi.acm.org/10.1145/381694.378802.

[22] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98*, pages 57–81. 1998.

[23] ISO90. ISO/IEC. international standard ISO/IEC 9899, programming languages - C. 1990.

[24] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[25] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI '13*, pages 423–434, 2013.

[26] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[27] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86, 2004.

[28] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI'07*, pages 278–289, 2007.

[29] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16, 2011.

[30] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353, 2011.

[31] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC'13*, 2013.

[32] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT '13*, pages 19–28, 2013. ISBN 978-1-4799-1021-2. URL http://dl.acm.org/citation.cfm?id=2523721.2523728.

[33] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI '12*, pages 229–238, 2012.

[34] D. Pearce, P. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM TOPLAS*, 30(1):4–es, 2007.

[35] G. Ramalingam. The undecidability of aliasing. *ACM TOPLAS*, 16 (5):1467–1471, 1994.

[36] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1):119–147, 2002.

[37] T. Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

[38] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995. ISBN 0-89791-692-1. doi: 10.1145/199448.199462. URL http://doi.acm.org/10.1145/199448.199462.

[39] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274, 2012.

[40] J. Späth, L. N. Q. Do, K. Ali, and E. Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. ECOOP, 2016.

[41] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. *PLDI '06*, pages 387–400, 2006.

[42] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA '05*, pages 59–76, 2005. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094817. URL http://doi.acm.org/10.1145/1094811.1094817.

[43] Y. Su, D. Ye, J. Xue, and X. Liao. An efficient GPU implementation of inclusion-based pointer analysis. *IEEE Trans. Parallel Distrib.*

Syst., 27(2):353–366, 2016. doi: 10.1109/TPDS.2015.2397933. URL http://dx.doi.org/10.1109/TPDS.2015.2397933.

[44] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.

[45] Y. Sui and J. Xue. On-demand strong update analysis via value-flow refinement. In *FSE '16*, 2016.

[46] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264, 2012.

[47] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11, 2013.

[48] Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *TSE '14*, 40(2):107–122, 2014.

[49] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *CGO '16*, pages 160–170. ACM, 2016.

[50] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization. In *LCTES '16*, pages 41–51. ACM, 2016.

[51] Y. Sui, D. Ye, Y. Su, and J. Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[52] Y. Sui, X. Fan, H. Zhou, and J. Xue. Loop-oriented pointer analysis for automatic SIMD vectorization. In *TECS '18*, 2018.

[53] A. Tavares, B. Boissinot, F. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *CC '14*, pages 18–39. Springer, 2014.

[54] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*, pages 350–360. ACM, 2018.

[55] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*, pages 131–144, 2004.

[56] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. *PLDI '95*, pages 1–12, 1995.

[57] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In *ISSTA '11*, pages 155–165, 2011.

[58] H. Yan, Y. Sui, S. Chen, and J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *ACSAC '17*, pages 42–54. ACM, 2017.

[59] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE '18*, 2018.

[60] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In *CGO '14*, 2014.

[61] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. 2014.

[62] 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, 2010.

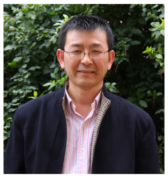
[63] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for C. In *OOPSLA '14*, pages 829–845, 2014. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660213. URL http://doi.acm.org/10.1145/2660193.2660213.

[64] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. In *PLDI '14*, pages 239–248, 2014.

[65] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL '12*, pages 427–440, 2012.

[66] X. Zheng and R. Rugina. Demand-driven alias analysis for C. In *POPL '08*, pages 197–208, 2008.

Yulei Sui is a Lecturer (Assistant Professor) and an ARC DECRA at Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). He obtained his Ph.D from University of New South Wales (UNSW), where he also holds an Adjunct Lecturer position. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a software engineer in Program Analysis Group for Memory Safe C project in Oracle Lab Australia. He was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM and a Best Paper Award winner at CGO, and has been awarded an Australian Discovery Early Career Researcher Award (DECRA) 2017-2019.



Jingling Xue received his BSc and MSc degrees in Computer Science and Engineering from Tsinghua University in 1984 and 1987, respectively, and his PhD degree in Computer Science and Engineering from Edinburgh University in 1992. He is a Scientia Professor in the School of Computer Science and Engineering, University of New South Wales, Australia, where he heads the Programming Languages and Compilers Group.

His main research interest has been programming languages and compilers for about 20 years. He is currently supervising a group of postdocs and PhD students on a number of topics including programming and compiler techniques for multicore processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities.

He is presently an Associate Editor of IEEE Transactions on Computers, Software: Practice and Engineering, International Journal of Parallel, Emergent and Distributed Systems, and Journal of Computer Science and Technology. He has served in various capacities on the Program Committees of many conferences in his field.