

Demand-Driven Pointer Analysis with Strong Updates via Value-Flow Refinement

Yulei Sui, School of Computer Science and Engineering, UNSW Australia

Jingling Xue, School of Computer Science and Engineering, UNSW Australia

We present a new demand-driven flow- and context-sensitive pointer analysis with strong updates for C programs, called SUPA, that enables computing points-to information via value-flow refinement, in environments with small time and memory budgets such as IDEs. 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. SUPA facilitates efficiency and precision tradeoffs by applying different pointer analyses in a hybrid multi-stage analysis framework.

We have implemented SUPA in LLVM (3.5.0) and evaluate it by choosing uninitialized pointer detection as a major client on 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 budget of at most 10000 value-flow edges per query). With context-sensitivity also considered, SUPA's two-stage analysis becomes more precise for some programs but also incurs more analysis times. SUPA is also amenable to parallelization. A parallel implementation of its single-stage flow-sensitive analysis achieves a speedup of up to 6.9x with an average of 3.05x a 8-core machine with respect to its sequential version.

CCS Concepts: •**Software and its engineering** → Software verification and validation; Software defect analysis; •**Theory of computation** → Program analysis;

Additional Key Words and Phrases: strong updates, value flow, pointer analysis, flow sensitivity

1. INTRODUCTION

Pointer analysis is one of the most fundamental static program analyses, on which virtually all others are built. The goal of pointer analysis is to compute an approximation of the set of abstract objects that a pointer can refer to. A pointer analysis is (1) *flow-sensitive* if it respects control flow and *flow-insensitive* otherwise and (2) *context-sensitive* if it distinguishes different calling contexts and *context-insensitive* otherwise.

Strong updates, where stores overwrite, i.e., kill the previous contents of their abstract destination objects with new values, is an important factor in the precision of pointer analysis [Hardekopf and Lin 2009; Lhoták and Chung 2011]. In the case of *weak updates*, these objects are assumed conservatively to also retain their old contents. Strong updates are possible only if flow-sensitivity is maintained. In addition, 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. By applying strong updates where needed, a pointer analysis can improve precision, thereby providing significant benefits to many clients, such as change impact analysis [Acharya and Robinson 2011], bug detection [Yan et al. 2016; Ye et al. 2014a], security analysis [Arzt et al. 2014], type state verification [Fink et al. 2008], compiler optimization [Sui et al. 2016b, 2013, 2014b], and symbolic execution [Blackshear et al. 2013].

In this paper, we introduce a demand-driven pointer analysis for C by investigating how to perform strong updates effectively in a flow- and context-sensitive framework. For C programs, flow-sensitivity is important in achieving the precision required by the afore-mentioned client applications due to strong updates performed. If context-sensitivity is also considered, some more strong updates are possible for some pro-

grams at the expense of more analysis times. For object-oriented languages like Java, context-sensitivity (without strong updates) is widely used in achieving useful precision [Lhoták and Hendren 2003; Li et al. 2014; Milanova et al. 2002, 2005; Smaragdakis et al. 2011; Sun et al. 2011; Xiao and Zhang 2011].

Ideally, strong updates at stores should be performed by analyzing 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 [Landi 1992; Ramalingam 1994].

Instead, traditional flow-sensitive pointer analysis (FS) for C [Hind and Pioli 1998; Kam and Ullman 1977] 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) [Hardekopf and Lin 2011; Li et al. 2011; Oh et al. 2012; Ye et al. 2014b; Yu et al. 2010] 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 [Lhoták and Chung 2011] for performing strong updates in C programs is introduced. It sacrifices the precision of FS 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. Virtually all of the prior 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 [Lhoták and Chung 2011] are less precise, due to both missing strong updates and propagating the points-to information flow-insensitively across the weakly-updated locations.

In practice, a client application of a pointer analysis may require only parts of the program to be analyzed. 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. In all these cases, performing strong updates blindly across the entire program is cost-ineffective in achieving precision.

For C programs, 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? Demand-driven analyses for C [Heintze and Tardieu 2001; Zhang et al. 2014a; Zheng and Rugina 2008] and Java [Lu et al. 2013; Shang et al. 2012; Sridharan and Bodík 2006; Su et al. 2016; Yan et al. 2011] are flow-insensitive and thus cannot perform strong updates to produce the precision needed by some clients. BOOMERANG [Späth et al. 2016] represents a recent flow- and context-sensitive demand-driven pointer analysis for Java. However, its access-path-based approach performs strong updates at a store $a.f = \dots$ only partially, by updating $a.f$ strongly and the aliases of $a.f.*$ weakly. Elsewhere, advances in whole-program flow-sensitive analysis for C have exploited some form of sparsity to improve performance [Hardekopf and Lin 2011; Li et al. 2011; Oh et al. 2012; Ye et al. 2014b; Yu et al. 2010]. However, how to replicate this success for demand-driven flow-sensitive analysis for C is unclear. Finally, it remains open as to whether sparse strong update analysis can be performed both flow- and context-sensitively on-demand to avoid under- or over-analyzing.

In this paper, we introduce SUPA, the first demand-driven pointer analysis with strong updates for C, designed to support flexible yet effective tradeoffs between effi-

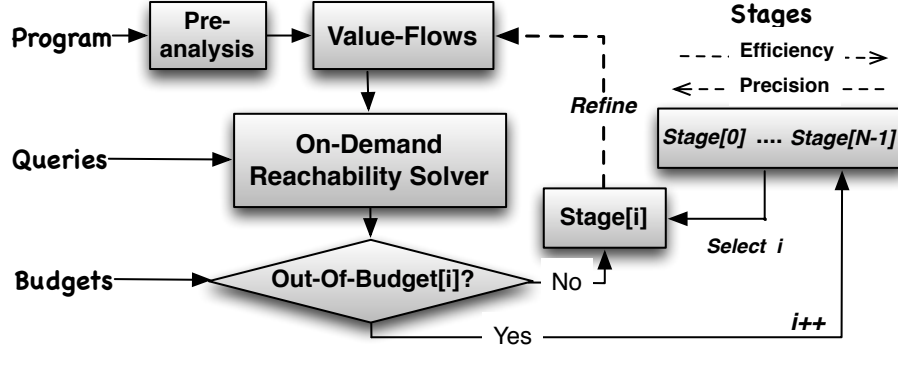


Fig. 1: Overview of SUPA

ciency and precision in answering client queries, in environments with small time and memory budgets such as IDEs. As shown in Figure 1, the novelty behind SUPA lies in performing **Strong Update Analysis** precisely by refining imprecisely pre-computed value-flows away in a hybrid multi-stage analysis framework. Given a points-to query, strong updates are performed by solving a graph-reachability problem on an interprocedural 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 [Andersen 1994] (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 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.

SUPA handles large C programs by staging analyses in increasing efficiency but decreasing precision in a hybrid manner. Currently, SUPA proceeds in two stages by applying FSCS and FS in that order with a configurable budget for each analysis. When failing to answer a query in a stage within its allotted budget, SUPA downgrades itself to a more scalable but less precise analysis in the next stage and will eventually fall back to the pre-computed flow-insensitive information. At each stage, SUPA will re-answer the query by reusing the points-to information found from processing the current and earlier queries. By increasing the budgets used in the earlier stages (e.g., FSCS), SUPA will obtain improved precision via improved value-flow refinement.

In summary, this paper makes the following contributions:

- We present the first demand-driven flow- and context-sensitive pointer analysis with strong updates for C that enables computing precise points-to information by refining away imprecisely precomputed value-flows, subject to analysis budgets.
- We introduce a hybrid multi-stage analysis framework that facilitates efficiency and precision tradeoffs by staging different analyses in answering client queries.
- We have produced an implementation of SUPA in LLVM (3.5.0) [SUPA 2016]. 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 also possible. SUPA’s two-stage analysis then becomes more precise for some programs at the expense of more analysis times.

- We present four case studies to demonstrate that SUPA is effective in checking whether variables are initialized or not in real-world applications.
- We show that SUPA is amenable to parallelization. To demonstrate this, we have developed a parallel implementation of SUPA’s single-stage flow-sensitive analysis based on Intel Threading Building Blocks (TBB), achieving a speedup of up to 6.9x with an average of 3.05x a 8-core machine over its sequential version.

The rest of this paper is organized as follows. Section 2 provides the background information. Section 3 presents a motivating example. Section 4 introduces our formalism for SUPA. Section 5 discusses and analyzes our experimental results. Section 6 contains four case studies. Section 7 describes a parallel implementation of SUPA. Section 8 describes the related work. Finally, Section 9 concludes the paper.

2. BACKGROUND

We describe how to represent a C program by an interprocedural 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 how to put top-level variables in SSA form. Section 2.3 describes how to put address-taken variables in SSA form. Section 2.4 constructs a sparse value-flow graph that represents the def-use chains for both top-level and address-taken variables in the program.

2.1. LLVM-IR

We perform pointer analysis in the LLVM-IR of a program, as in [Balatsouras and Smaragdakis 2016; Hardekopf and Lin 2011; Lhoták and Chung 2011; Li et al. 2011; Sui et al. 2012; Ye et al. 2014b]. The domains and the LLVM instructions relevant to pointer analysis are given in Table I. 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 I, 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 $f(r_1, \dots, r_n)$ 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 each object o (except for a function), we write o_{fld} to represent the sub-object that corresponds to its field fld . For flow-sensitive pointer analysis, the initializations for global objects take place at the entry of main().

Table I: Domains and LLVM instructions used by pointer analysis.

Analysis Domains			LLVM Instruction Set	
ℓ	$\in \mathcal{L}$	instruction labels	ADDR_OF	$p = \&o$
fld	$\in \mathcal{C}$	constants (field accesses)	COPY	$p = q$
s	$\in \mathcal{S}$	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	$p = q(r_1, \dots, r_n)$
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{O}$	program variables	FUNENTRY	$f(r_1, \dots, r_n)$
			FUNEXIT	$ret_f p$

COPY denotes a casting instruction (e.g., bitcast) 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. LOAD (STORE) is a memory accessing instruction that reads (write) a value from (into) an address-taken object.

Our handling of field-sensitivity is ANSI-compliant. Given a pointer to an aggregate (e.g., a struct or an array), pointer arithmetic used for accessing anything other than the aggregate itself has undefined behavior [ISO90 1990; Pearce et al. 2007] and thus not modeled. To model the field accesses of a struct object, FIELD represents a getelementptr instruction with its field offset fld as a constant value. A getelementptr instruction that operates on a non-constant field of a struct is modeled as COPY instructions, one for every field of the struct conservatively. Arrays are treated monolithically.

CALL, $p = q(r_1, \dots, r_n)$, 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).

2.2. SSA Form for Top-Level Variables

LLVM-IR is known as a partial SSA form since only top-level variables are in SSA form. In LLVM-IR, top-level variables are explicit, i.e., directly accessed and can thus be put in SSA form by using a standard SSA construction algorithm [Cytron et al. 1991] (with PHI instructions inserted at confluence points).

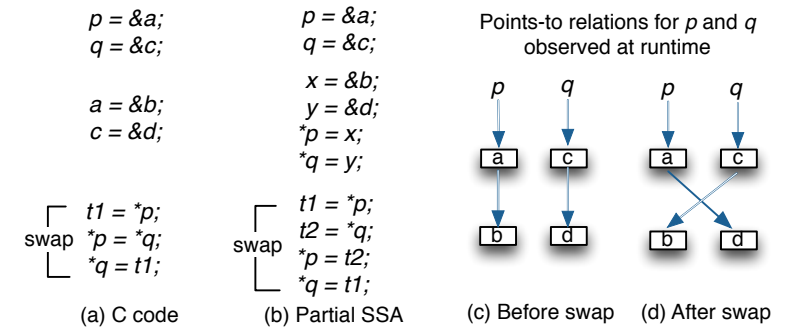


Fig. 2: A swap example and its partial SSA form.

Let us illustrate LLVM's partial SSA form by using an example in Figure 2. Figure 2(a) shows a swap program in C and Figure 2(b) gives 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.

In LLVM-IR, all top-level variables are in SSA form. In this example, all top-level variables are trivially in SSA form, as each has exactly one definition only. As a result, the def-use chains for top-level variables are readily available.

However, address-taken variables are accessed indirectly at loads and stores via top-level variables and thus not in SSA form. For example, the address-taken variable a is defined implicitly twice, once at $*p = x$ and once at $*p = t2$, and the address-taken variable c is also defined implicitly twice, once at $*q = y$ and once at $*q = t1$. As a result, the def-use chains for address-taken variables are not immediately available.

2.3. SSA Form for Address-Taken Variables

Starting with LLVM’s partial SSA form, we first perform a pre-analysis by using Andersen’s algorithm flow- and context-insensitively [Andersen 1994], implemented in SVF [Sui and Xue 2016]. We then put address-taken variables in *memory SSA form*, by using the SSA construction algorithm [Cytron et al. 1991]. Imprecise points-to information computed this way will be refined by our demand-driven pointer analysis.

Given a variable v , $AnderPts(v)$ represents its points-to set computed by Andersen’s algorithm. There are two steps [Sui et al. 2014a], illustrated in Figures 3(a) and (b) intraprocedurally and in Figures 4(a) and (b) interprocedurally.

Step 1: Computing Modification and Reference Side-Effects. As shown in Figure 3(a), every load, e.g., $t1 = *q$ is annotated with a $\mu(a)$ 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 $a = \chi(a)$ 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 [Sui et al. 2014a, §4.2.1]. For a given callsite ℓ , it is annotated with $\mu(a)$ ($a = \chi(a)$) if a may be read (modified) inside the callees of ℓ (discovered by Andersen’s pointer analysis). In addition, appropriate χ and μ 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 4(a) gives an example modified from Figure 3(a) by moving the four swap instructions into a function, `swap`. For read side-effects, $\mu(a)$ and $\mu(c)$ 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 $a = \chi(a)$ and $c = \chi(c)$ to receive the values of a and c passed from ℓ_7 . For modification side-effects, $a = \chi(a)$ and $c = \chi(c)$ 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 $\mu(a)$ and $\mu(c)$.

Step 2: Memory SSA Renaming. All the address-taken variables are converted into SSA form as suggested in [Chow et al. 1996]. Every $\mu(a)$ is treated as a use of a . Every $a = \chi(a)$ 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 [Cytron et al. 1991].

For the program annotated with μ ’s and χ ’s in Figure 3(a), Figure 3(b) gives its memory SSA form. Similarly, Figure 4(b) gives the memory SSA form for Figure 4(a).

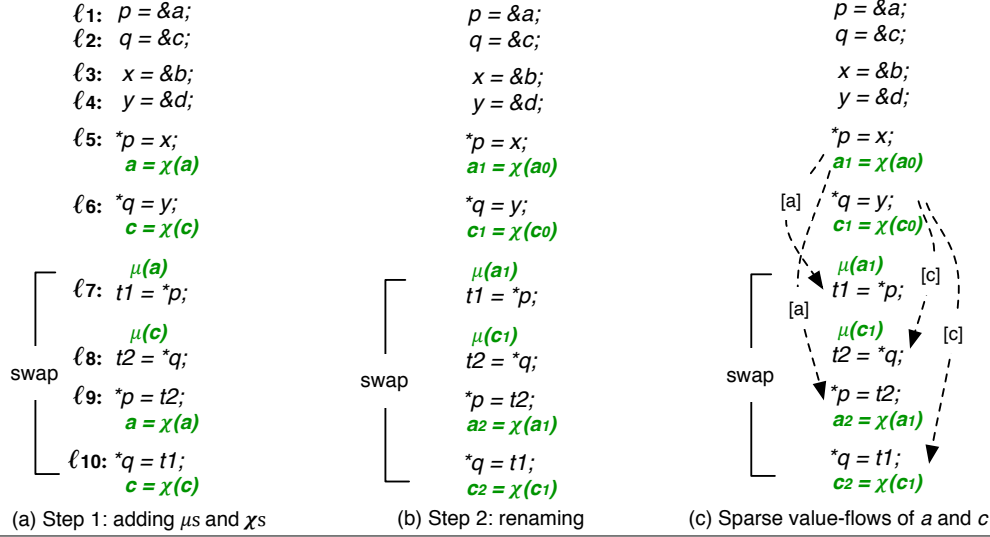


Fig. 3: Memory SSA form and sparse value-flows constructed intraprocedurally for Figure 2, obtained with Andersen’s analysis: $\text{AnderPts}(p) = \{a\}$ and $\text{AnderPts}(q) = \{c\}$.

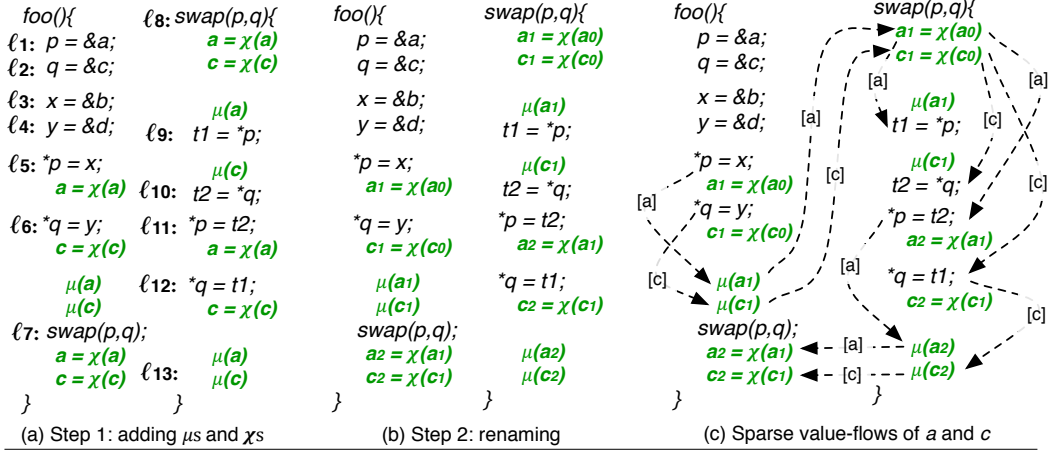


Fig. 4: Memory SSA form and sparse value-flows constructed interprocedurally for an example modified from Figure 2 with its four swap instructions moved into a separate function, called swap. ℓ_8 and ℓ_{13} correspond to the FUNENTRY and FUNEXIT of swap.

2.4. 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 II. We discussed top-level variables earlier. For the two address-taken variables a and c in Figure 2, Figure 3(c) depicts their def-use chains, i.e., sparse value-flows for the memory SSA form in Figure 3(b). Similarly, Figure 4(c) gives their sparse value-flows for the memory SSA form in Figure 4(b).

Given a program, a *sparse value-flow graph* (SVFG), $G_{\text{svfg}} = (N, E)$, is a multi-edged directed graph that captures its def-use chains for both top-level and address-taken

Table II: Def-use information of both top-level and address-taken variables. Def_v (Use_v) denotes the set of definition (use) instructions for a variable $v \in \mathcal{V}$.

Instruction	ℓ	Defs and Uses of Variables in Memory SSA Form
$p = \&o$		$\{\ell\} = Def_p$
$p = q$		$\{\ell\} = Def_p \quad \ell \in Use_q$
$p = \phi(q, r)$		$\{\ell\} = Def_p \quad \ell \in Use_q \quad \ell \in Use_r$
$p = \&q \rightarrow fld$		$\{\ell\} = Def_p \quad \ell \in Use_q$
$p = *q$	$\mu(a_i)$	$\{\ell\} = Def_p \quad \ell \in Use_q \quad \ell \in Use_{a_i}$
$*p = q$	$a_{i+1} = \chi(a_i)$	$\ell \in Use_p \quad \ell \in Use_q \quad \ell \in Def_{a_{i+1}} \quad \ell \in Use_{a_i}$
$p = q(r_1, \dots, r_n)$		$\{\ell\} = Def_p \quad \ell \in Use_q \quad \forall i \in 1, \dots, n : \ell \in Use_{r_i}$
$\mu(a_i) \quad a_{j+1} = \chi(a_j)$		$\ell \in Use_{a_i} \quad \ell \in Def_{a_{j+1}} \quad \ell \in Use_{a_j}$
$f(r_1, \dots, r_n)$	$a_{i+1} = \chi(a_i)$	$\forall i \in 1, \dots, n : \ell \in Def_{r_i} \quad \ell \in Def_{a_{i+1}} \quad \ell \in Use_{a_i}$
$ret_f p$	$\mu(a_i)$	$\ell \in Use_p \quad \ell \in Use_{a_i}$

$$\begin{array}{c}
\text{[INTRA-TOP]} \quad \frac{\ell \in Def_p \quad \ell' \in Use_p}{\ell \xrightarrow{p} \ell'} \qquad \text{[INTRA-ADDR]} \quad \frac{\ell \in Def_{a_i} \quad \ell' \in Use_{a_i}}{\ell \xrightarrow{a} \ell'} \\
\text{[INTER-CALL-TOP]} \quad \frac{\ell : p = q(r_1, \dots, r_n) \quad o_f \in \mathbf{AnderPts}(q) \quad \ell' : f(r'_1, \dots, r'_n)}{\forall i \in 1, \dots, n : \ell \xrightarrow{r_i} \ell'} \\
\text{[INTER-RET-TOP]} \quad \frac{\ell : p = q(\dots) \quad a_f \in \mathbf{AnderPts}(q) \quad \ell' : ret_f p'}{\ell' \xrightarrow{p} \ell} \\
\text{[INTER-CALL-ADDR]} \quad \frac{\ell : p = q(\dots) \quad \mu(a_i) \quad a_f \in \mathbf{AnderPts}(q) \quad \ell' : f(\dots) \quad a_{j+1} = \chi(a_j)}{\ell \xrightarrow{a} \ell'} \\
\text{[INTER-RET-ADDR]} \quad \frac{\ell : - = q(\dots) \quad a_{j+1} = \chi(a_j) \quad a_f \in \mathbf{AnderPts}(q) \quad \ell' : ret_f - \quad \mu(a_i)}{\ell' \xrightarrow{a} \ell}
\end{array}$$

Fig. 5: Value-flow construction in Memory SSA form.

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.

Figure 5 gives the rules for connecting value-flows between two instructions based on the defs and uses computed in Table II. 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 $\mu(a_i)$, its unique definition in SSA form is a_i at an ℓ annotated with $a_i = \chi(a_{i-1})$. 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 II is only intraprocedural. According to Figure 5, 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 4(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 4(b).

The SVFG obtained this way may contain spurious def-use chains, such as $\ell_5 \xrightarrow{a} \ell_9$ in Figure 3, 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 [Hardekopf and Lin 2011; Nagaraj and Govindarajan 2013; Sui et al. 2016a; Ye et al. 2014b]. 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.

3. A MOTIVATING EXAMPLE

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

Our example program, shown in Figure 6(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 to enable us 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 6(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 6(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 [Hardekopf and Lin 2011]. 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 6(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 6(c) shows how the points-to relations in Figure 6(b) are over-approximated flow-insensitively by applying Andersen's analysis [Andersen 1994]. 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. When $\ell_{11} - \ell_{16}$ are analyzed, the seven spurious points-to relations (shown in dashed arrows in Figure 6(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 uninitialized pointer detection client.

Let us now explain how SUPA, shown in Figure 1, works. SUPA will first perform a pre-analysis to the example program to build the SVFG given in Figure 6(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

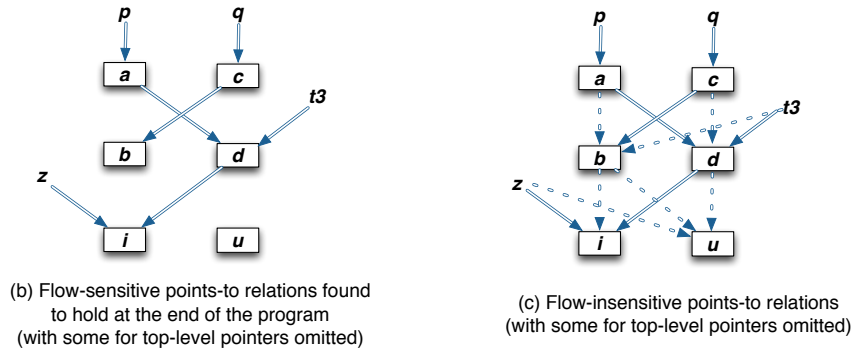
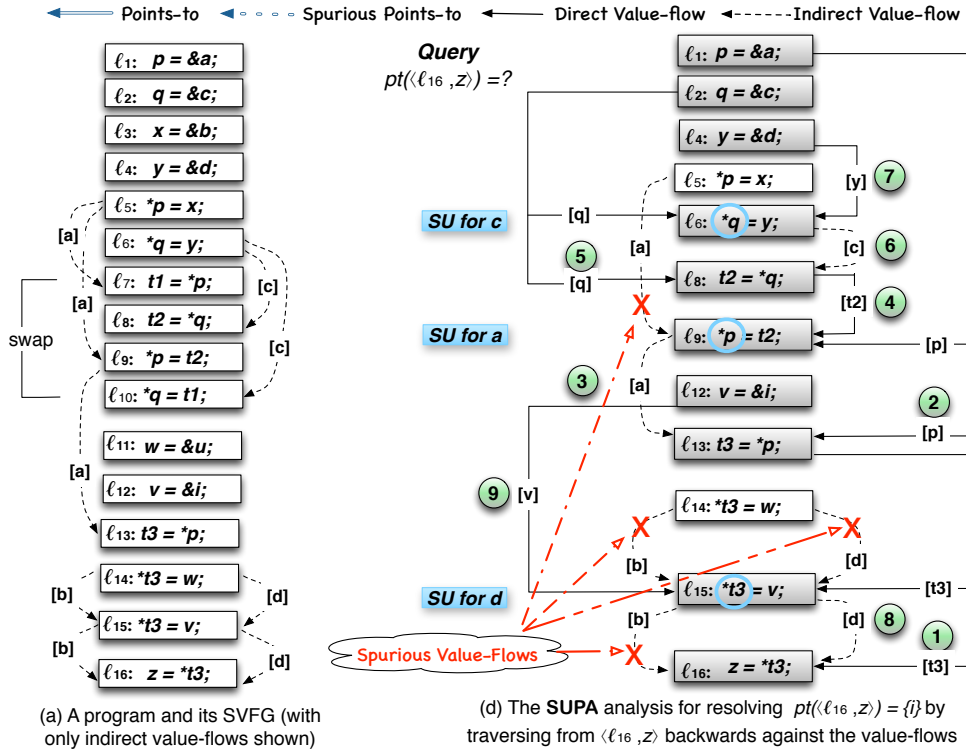


Fig. 6: A motivating example for illustrating SUPA (SU stands for “Strong Update”).

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 6(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.

Let us consider a single-stage analysis with $\mathbf{Stage}[N-1] = \mathbf{Stage}[0] = FS$ in Figure 1. Figure 6(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 8. The def-use chains for only the relevant top-level variables are shown. By filtering out the four spurious value-flows (marked by \times), 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 no warning for z will be issued.

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

— **On-Demand Strong Updates**

A whole-program flow-sensitive analysis like SFS [Hardekopf and Lin 2011] 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 [Acharya and Robinson 2011].

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 [Shang et al. 2012; Sridharan and Bodik 2006; Yan et al. 2011; Zhang et al. 2014a; Zheng and Rugina 2008] 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 \times . 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 framework. 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 6(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. DEMAND-DRIVEN STRONG UPDATES

We introduce our demand-driven pointer analysis with strong updates, as illustrated in Figure 1. We first describe our inference rules in a flow-sensitive setting (Section 4.1). We then discuss our context-sensitive extension (Section 4.2). Finally, we present our hybrid multi-stage analysis framework (Section 4.3). All our analyses are field-sensitive, thereby enabling more strong updates to be performed to struct objects.

$$\begin{array}{c}
\text{[ADDR]} \quad \frac{\ell : p = \&o}{\langle \ell, p \rangle \leftrightarrow \widehat{o}} \quad \text{[COPY]} \quad \frac{\ell : p = q \quad \ell' \xrightarrow{q} \ell}{\langle \ell, p \rangle \leftrightarrow \langle \ell', q \rangle} \\
\\
\text{[PHI]} \quad \frac{\ell : p = \phi(q, r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{\langle \ell, p \rangle \leftrightarrow \langle \ell', q \rangle \quad \langle \ell, p \rangle \leftrightarrow \langle \ell'', r \rangle} \\
\\
\text{[FIELD]} \quad \frac{\ell : p = \&q \rightarrow fld \quad \ell' \xrightarrow{q} \ell \quad \langle \ell', q \rangle \leftrightarrow \widehat{o}}{\langle \ell, p \rangle \leftrightarrow \widehat{ofld}} \\
\\
\text{[LOAD]} \quad \frac{\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle \ell'', q \rangle \leftrightarrow \widehat{o} \quad \ell' \xrightarrow{o} \ell}{\langle \ell, p \rangle \leftrightarrow \langle \ell', o \rangle} \\
\\
\text{[STORE]} \quad \frac{\ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle \ell'', p \rangle \leftrightarrow \widehat{o} \quad \ell' \xrightarrow{q} \ell}{\langle \ell, o \rangle \leftrightarrow \langle \ell', q \rangle} \\
\\
\text{[SU/WU]} \quad \frac{\ell : *p = - \quad \ell' \xrightarrow{o} \ell \quad o \in \mathcal{A} \setminus \text{kill}(\ell, p)}{\langle \ell, o \rangle \leftrightarrow \langle \ell', o \rangle} \\
\\
\text{[CALL]} \quad \frac{\ell : - = q(\dots, r, \dots) \quad \mu(o_j) \quad \ell' : f(\dots, r', \dots) \quad o_{i+1} = \chi(o_i)}{\frac{\ell'' \xrightarrow{q} \ell \quad \langle \ell'', q \rangle \leftrightarrow \widehat{o}_f \quad \ell \xrightarrow{r} \ell' \quad \ell \xrightarrow{o} \ell'}{\langle \ell', r' \rangle \leftrightarrow \langle \ell, r \rangle \quad \langle \ell', o \rangle \leftrightarrow \langle \ell, o \rangle}} \\
\\
\text{[RET]} \quad \frac{\ell : p = q(\dots) \quad o_{j+1} = \chi(o_j) \quad \ell' : \text{ret}_f p' \quad \mu(o_i)}{\frac{\ell'' \xrightarrow{q} \ell \quad \langle \ell'', q \rangle \leftrightarrow \widehat{o}_f \quad \ell' \xrightarrow{p'} \ell \quad \ell' \xrightarrow{o} \ell}{\langle \ell, p \rangle \leftrightarrow \langle p', \ell' \rangle \quad \langle \ell, o \rangle \leftrightarrow \langle \ell', o \rangle}} \\
\\
\text{[COMPO]} \quad \frac{lv \leftrightarrow lv' \quad lv' \leftrightarrow lv''}{lv \leftrightarrow lv''} \\
\\
\text{kill}(\ell, p) = \begin{cases} \{o'\} & \text{if } pt(\langle \ell, p \rangle) = \{o'\} \wedge o' \in \text{singletons} \\ \mathcal{A} & \text{else if } pt(\langle \ell, p \rangle) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

Fig. 7: Single-stage flow-sensitive SUPA analysis with demand-driven strong updates.

4.1. Formalism: 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 [Andersen 1994] as a pre-analysis, as discussed in Section 2.4 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 I. SUPA conducts a backward reachability analysis flow-sensitively on G_{vfg} by computing a reachability relation, $\leftrightarrow \subseteq \mathbb{V} \times \mathbb{V}$. In our formalism, $\langle \ell, v \rangle \leftrightarrow \langle \ell', v' \rangle$ 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 ADDR OF

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 \hat{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 \hat{o} , defined as follows:

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

Despite flow-sensitivity, our formalization in Figure 7 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 \hat{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 2.4).

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 o is *actually* used by, i.e., aliased with $*q$ at ℓ , which requires the reachability relation $\langle \ell'', q \rangle \leftrightarrow \hat{o}$ 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 $\dots = *q$ is aliased with $*p$ in a store $*p = \dots$ executed earlier, then p and q must be both backward reachable from \hat{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 $\ell : p = \dots$. Defining its kill set $kill(\ell, p)$ involves three cases. In Case (1), p points to one *singleton object* o' in *singletons*, which contains all objects in \mathcal{A} except the local variables in recursion, arrays (treated monolithically) or heap objects [Lhoták and Chung 2011]. In Section 4.2, 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, $\langle \ell, o \rangle \leftrightarrow \langle \ell', o \rangle$ 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 [Hardekopf and Lin 2009, 2011; Lhoták and Chung 2011]. In Case (3), a weak update is performed to o so that its old contents at ℓ' are preserved. Thus, $\langle \ell, o \rangle \leftrightarrow \langle \ell', o \rangle$ 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 pre-computed call graph built by the pre-analysis as in [Hardekopf and Lin 2011]. 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, \leftrightarrow is transitive, stated by [COMPO].

Let us try all our rules, by first revisiting our motivating example where strong updates are performed (Example 4.1) and then examining weak updates (Example 4.2).

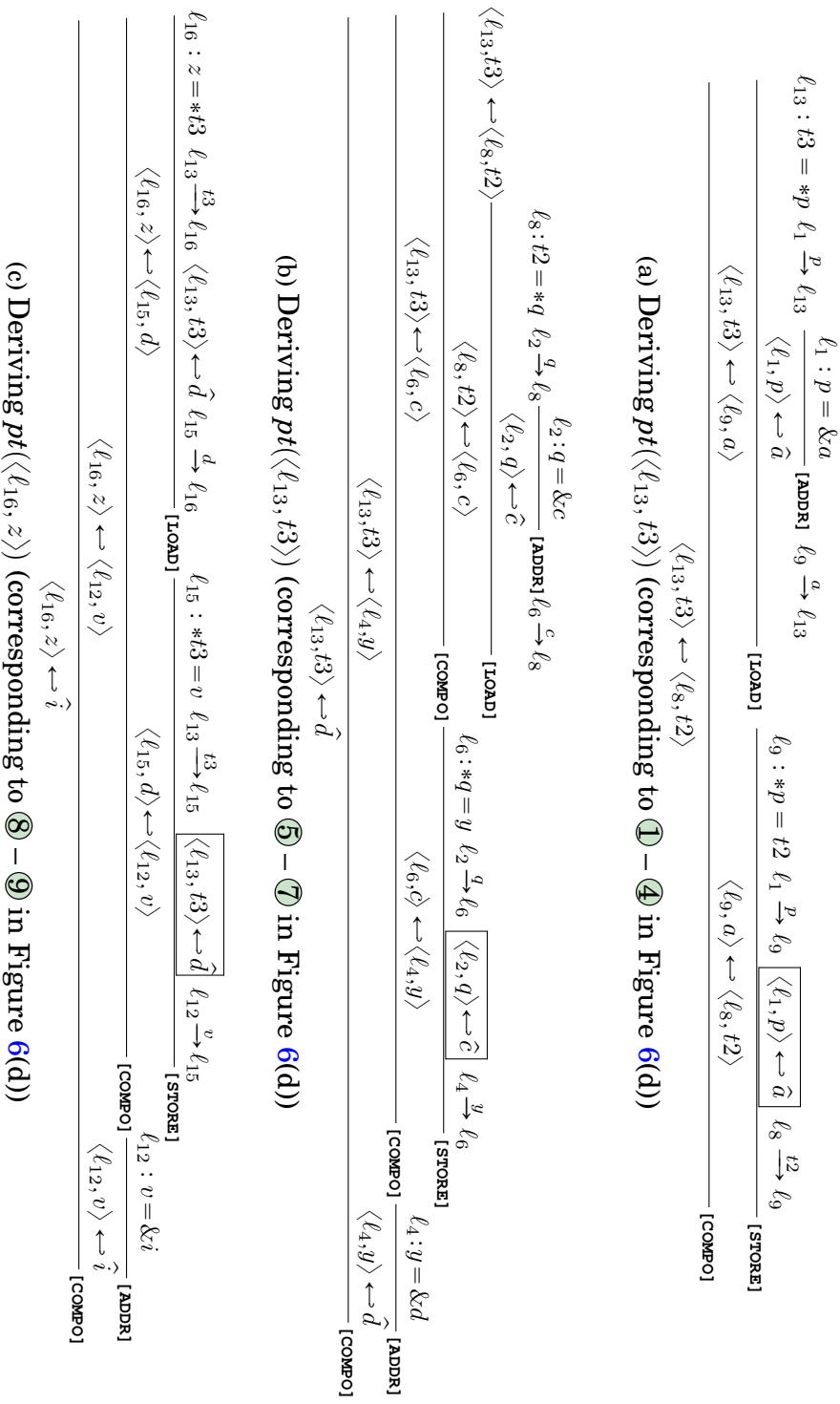


Fig. 8: Reachability derivations for $pt(\langle\langle\ell_{16}, z\rangle\rangle)$ shown in Figure 6(d) (with reuse of cached points-to results inside each box).

Example 4.1. Figure 8 shows how we apply the rules of SUPA to answer $pt(\langle \ell_{16}, z \rangle)$ illustrated in Figure 6(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 $\langle \ell_9, a \rangle \leftarrow \langle \ell_5, a \rangle$ is falsified. At ℓ_{15} , $\ell_{14} \xrightarrow{b} \ell_{15}$ and $\ell_{14} \xrightarrow{d} \ell_{15}$ are filtered out since $\langle \ell_{15}, b \rangle \leftarrow \langle \ell_{14}, b \rangle$ and $\langle \ell_{15}, d \rangle \leftarrow \langle \ell_{14}, d \rangle$ are falsified. Finally, $\ell_{15} \xrightarrow{b} \ell_{16}$ is ignored since $t3$ points to d only ([LOAD]). \square

SUPA improves performance by caching points-to results to reduce redundant traversal, with reuse happening in the marked boxes in Figure 8. For example, in Figure 8(c), $pt(\langle \ell_{13}, t3 \rangle) = \{\hat{d}\}$ computed in [LOAD] is reused in [STORE].

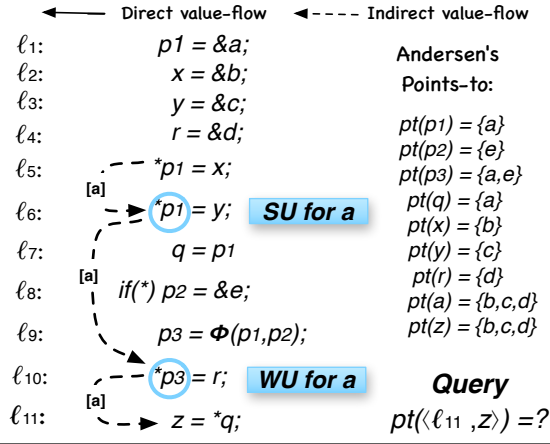


Fig. 9: Resolving $pt(\langle \ell_{11}, z \rangle) = \{c, d\}$ with a weak update.

Example 4.2. Let us consider a weak update example in Figure 9 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: $\langle \ell_9, p3 \rangle \leftarrow \hat{a}$ and $\langle \ell_9, p3 \rangle \leftarrow \hat{e}$. Thus, a weak update is performed to the two locations a and e at ℓ_{10} . Let us focus on \hat{a} only. By applying [STORE], $\langle \ell_{10}, a \rangle \leftarrow \langle \ell_4, r \rangle \leftarrow \hat{d}$. By applying [SU/WU], $\langle \ell_{10}, a \rangle \leftarrow \langle \ell_6, a \rangle \leftarrow \langle \ell_3, y \rangle \leftarrow \hat{c}$. Thus, $pt(\langle \ell_{11}, a \rangle) = \{c, d\}$, which excludes b due to a strong update performed at ℓ_6 . As $pt(\langle \ell_7, q \rangle) = \{a\}$, we obtain $pt(\langle \ell_{11}, z \rangle) = \{c, d\}$. \square

Unlike [Lhoták and Chung 2011], 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 9, 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.

4.1.1. 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 4.3. Figure 10 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

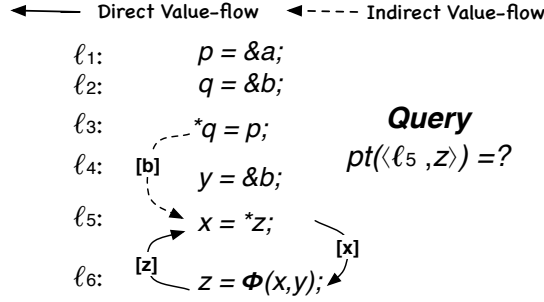


Fig. 10: Resolving $pt(\langle \ell_5, z \rangle) = \{a, b\}$ in a value-flow cycle.

this value-flow cycle in two iterations. In the first iteration, a pointed-to target \hat{b} is found since $\langle \ell_6, z \rangle \leftrightarrow \langle \ell_4, y \rangle \leftrightarrow \hat{b}$. Due to $\langle \ell_2, q \rangle \leftrightarrow \hat{b}$, $*z$ and $*q$ are found to be aliases. In the second iteration, another target \hat{a} is found since $\langle \ell_6, z \rangle \leftrightarrow \langle \ell_5, x \rangle \leftrightarrow \langle \ell_3, b \rangle \leftrightarrow \langle \ell_1, p \rangle \leftrightarrow \hat{a}$. Thus, $pt(\langle \ell_6, z \rangle) = \{a, b\}$ is obtained. \square

4.1.2. 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.

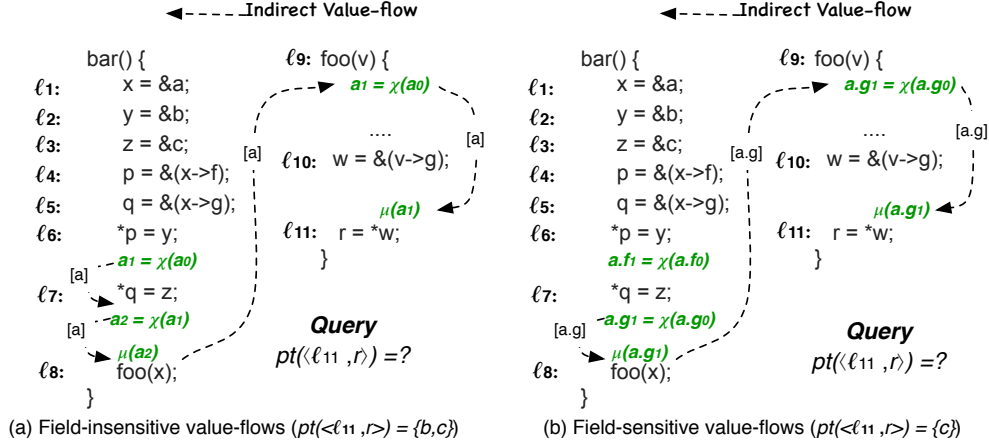


Fig. 11: Resolving $pt(\langle \ell_{11}, r \rangle) = \{c\}$ with field-sensitivity.

Example 4.4. Figure 11 illustrates the effects of field-sensitivity on computing the points-to information for r at ℓ_{11} . Without field-sensitivity, as illustrated in Figure 11(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. To answer the points-to query for r at ℓ_{11} , we compute first $\langle \ell_{11}, r \rangle \leftrightarrow \langle \ell_{10}, w \rangle$ and then $\langle \ell_{10}, w \rangle \leftrightarrow \langle \ell_9, v \rangle \leftrightarrow \langle \ell_8, x \rangle \leftrightarrow \langle \ell_1, x \rangle \leftrightarrow \hat{a}$. By applying

[FIELD] at ℓ_{10} and [LOAD] at ℓ_{11} , we obtain $\langle \ell_{11}, r \rangle \leftarrow \langle \ell_{11}, a.g \rangle$. 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 $pt(\langle \ell_{11}, r \rangle) \leftarrow \langle \ell_9, a.g \rangle \leftarrow \langle \ell_8, a.g \rangle \leftarrow \langle \ell_7, a.g \rangle \leftarrow \langle \ell_3, z \rangle \leftarrow \hat{c}$. \square

4.1.3. Properties

THEOREM 4.5 (SOUNDNESS). *SUPA is sound in analyzing a program as long as its pre-analysis (for computing the SVFG of the program) is sound.*

PROOF. *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) 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.* \square

THEOREM 4.6 (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. *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, $pt_{\text{SUPA}}(\langle \ell, v \rangle) = pt_{\text{FS}}(\langle \ell, v \rangle)$. \square*

4.2. Formalism: 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 [Lu et al. 2013; Reps et al. 1995; Shang et al. 2012; Sridharan and Bodík 2006; Yan et al. 2011]. A context stack c is encoded as a sequence of callsites, $[\kappa_1 \dots \kappa_m]$, where κ_i is a call instruction. $c \oplus \kappa$ 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 \hat{o} that is created at an ADDR_OF statement under context c is also context-sensitive, identified as (c, \hat{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 12:

$$pt(\langle c, \ell, v \rangle) = \{(c', o) \mid \langle c, \ell, v \rangle \leftarrow (c', \hat{o})\} \quad (2)$$

$$\begin{array}{c}
\text{[C-ADDR]} \frac{c, \ell : p = \&o}{\langle c, \ell, p \rangle \leftrightarrow (c, \widehat{o})} \quad \text{[C-COPY]} \frac{c, \ell : p = q \quad \ell' \xrightarrow{q} \ell}{\langle c, \ell, p \rangle \leftrightarrow \langle c, \ell', q \rangle} \\
\text{[C-PHI]} \frac{c, \ell : p = \phi(q, r) \quad \ell' \xrightarrow{q} \ell \quad \ell'' \xrightarrow{r} \ell}{\langle c, \ell, p \rangle \leftrightarrow \langle c, \ell', q \rangle \quad \langle c, \ell, p \rangle \leftrightarrow \langle c, \ell'', r \rangle} \\
\text{[C-FIELD]} \frac{c, \ell : p = \&q \rightarrow fld \quad \ell' \xrightarrow{q} \ell \quad \langle c, \ell', q \rangle \leftrightarrow (c', \widehat{o})}{\langle c, \ell, p \rangle \leftrightarrow (c', \widehat{o.fld})} \\
\text{[C-LOAD]} \frac{c, \ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle c, \ell'', q \rangle \leftrightarrow (c', \widehat{o}) \quad \ell' \xrightarrow{o} \ell}{\langle c, \ell, p \rangle \leftrightarrow \langle c', \ell', o \rangle} \\
\text{[C-STORE]} \frac{c, \ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle c, \ell'', p \rangle \leftrightarrow (c', \widehat{o}) \quad \ell' \xrightarrow{q} \ell}{\langle c', \ell, o \rangle \leftrightarrow \langle c, \ell', q \rangle} \\
\text{[C-SU/WU]} \frac{c, \ell : *p = - \quad \ell' \xrightarrow{o} \ell \quad (c', o) \in \mathcal{A} \setminus \text{kill}(c, \ell, p)}{\langle c', \ell, o \rangle \leftrightarrow \langle c', \ell', o \rangle} \\
\text{[C-COMPO]} \frac{lv \leftrightarrow lv' \quad lv' \leftrightarrow lv''}{lv \leftrightarrow lv''} \\
\text{[C-CALL]} \frac{c, \ell : - = q(\dots, r, \dots) \quad \mu(o_j) \quad \ell'' \xrightarrow{q} \ell \quad \langle c, \ell'', q \rangle \leftrightarrow (-, \widehat{o}_f) \quad \ell \xrightarrow{r} \ell' \quad \ell' \xrightarrow{o} \ell' \quad c', \ell' : f(\dots, r', \dots) \quad o_{i+1} = \chi(o_i) \quad c = c' \ominus \ell}{\langle c', \ell', r' \rangle \leftrightarrow \langle c, \ell, r \rangle \quad \langle c', \ell', o \rangle \leftrightarrow \langle c, \ell, o \rangle} \\
\text{[C-RET]} \frac{c, \ell : p = q(\dots) \quad o_{j+1} = \chi(o_j) \quad \ell'' \xrightarrow{q} \ell \quad \langle c, \ell'', q \rangle \leftrightarrow (-, \widehat{o}_f) \quad \ell' \xrightarrow{p'} \ell \quad \ell' \xrightarrow{o} \ell \quad c', \ell' : \text{ret}_f p' \quad \mu(o_i) \quad c' = c \oplus \ell}{\langle c, \ell, p \rangle \leftrightarrow \langle c', p', \ell' \rangle \quad \langle c, \ell, o \rangle \leftrightarrow \langle c', \ell', o \rangle} \\
\text{kill}(c, \ell, p) = \begin{cases} \{(c', o')\} & \text{if } pt(\langle c, \ell, p \rangle) = \{(c', o')\} \wedge (c', o') \in \text{cxtSingletons} \\ \mathcal{A} & \text{else if } pt(\langle c, \ell, p \rangle) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

Fig. 12: Single-stage flow- and context-sensitive SUPA analysis with demand-driven strong updates.

where the reachability relation \leftrightarrow 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 [CALL] in Figure 7, 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* $(c', o') \in$

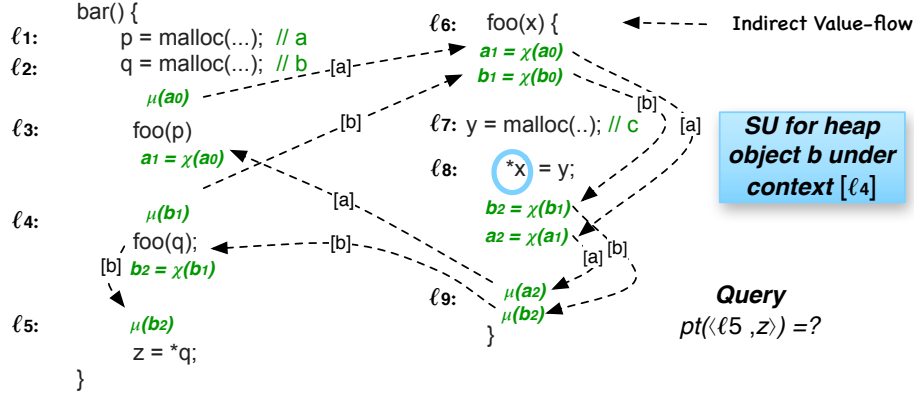


Fig. 13: Resolving $pt(\langle \ell_5, z \rangle) = \{[\ell_4], c\}$ with context-sensitive strong updates.

$extSingletons$, where o' is a (non-heap) singleton defined in Section 4.1 or a heap object with c' being a *concrete* context, i.e., one not involved in recursion or loops.

Example 4.7. Let us use an example given in Figure 13 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 17(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 b points to previously (not shown to avoid cluttering). With context-sensitivity, SUPA finds that $\langle [\ell_5], z \rangle \leftarrow \langle [\ell_5], b \rangle \leftarrow \langle [\ell_4], b \rangle \leftarrow \langle [\ell_4], \ell_9, b \rangle \leftarrow \langle [\ell_4], \ell_8, b \rangle \leftarrow \langle [\ell_4], \ell_7, y \rangle \leftarrow ([\ell_4], \hat{c})$. 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. \square

Given a program, the SCCs (strongly connected components) in its call graph are constructed on the fly. SUPA handles the SCCs in the program context-sensitively but the function calls inside a SCC context-insensitively as in [Sridharan and Bodík 2006].

4.3. SUPA: Hybrid Multi-Stage Analysis

To facilitate efficiency and precision tradeoffs in answering on-demand queries, SUPA, as illustrated in Figure 1, organizes its analyses in multiple stages sorted in increasing efficiency but decreasing precision. Let there be M queries issued successively. Let the N stages of SUPA, **Stage**[0], \dots , **Stage**[$N-1$], be configured with budgets $\eta_0, \dots, \eta_{N-1}$, respectively. In our current implementation, each budget is specified as the maximum number of def-use chains traversed in the SVFG of the program.

SUPA answers a query on-demand by applying its N analyses successively, starting from **Stage**[0]. If the query is not answered after budget η_i has been exhausted at stage i , SUPA re-issues the query at stage $i+1$, and eventually falls back to the results that are pre-computed by pre-analysis.

SUPA caches fully computed points-to information in a query and reuses it in subsequent queries, as illustrated in Figure 8. Let Q be the set of queried variables issued from a program. Let $\mathcal{I} \supseteq Q$ be the set of variables reached from Q during the analysis. Let $(\ell, v) \in Q$ be a queried variable. We write $pt_{\eta_i}^i(\langle \Delta_i, \ell, v \rangle)$ to represent the points-to

set of a variable $\langle \ell, v \rangle$ computed at stage i under budget η_i , where Δ_i is a contextual qualifier at stage i (e.g., c in FSCS). By convention, $pt_{\eta_N}^N(\langle \Delta_N, \ell, v \rangle)$ denotes the points-to set obtained by pre-analysis, at **Stage[N]** (conceptually).

When resolving $pt_{\eta_i}^i(\langle \Delta_i, \ell, v \rangle)$ at stage i , suppose SUPA has reached a variable $\langle \ell', v' \rangle \in \mathcal{I}$ and needs to compute $pt_*^i(\langle \Delta_i, \ell', v' \rangle)$, where $*(\leq \eta_i)$ represents an unknown budget remaining, with (ℓ', v') being (ℓ, v) possibly (in a cycle).

Presently, SUPA exploits two types of reuse to improve efficiency with no loss of precision in a hybrid manner:

Backward Reuse: $(\ell', v') \in \mathcal{I}$. If $pt_*^j(\langle \Delta_j, \ell', v' \rangle)$, where $j \leq i$, was previously cached, then $pt_*^i(\langle \Delta_i, \ell', v' \rangle) = pt_*^j(\langle \Delta_j, \ell', v' \rangle)$, provided that $pt_*^j(\langle \Delta_j, \ell', v' \rangle)$ is a sound representation of $pt_*^i(\langle \Delta_i, \ell', v' \rangle)$. For example, if **Stage[i]** = *FS* and **Stage[j]** = *FSCS*, then $pt_*^{FSCS}(\langle c', \ell', v' \rangle)$ can be reused for $pt_*^{FS}(\langle \ell', v' \rangle)$ if c' is *true*, representing a context-free points-to set.

Forward Reuse: $(\ell', v') \in \mathcal{Q}$. If $pt_{\eta_j}^j(\langle \Delta_j, \ell', v' \rangle)$, where $j > i$, was previously computed and cached but $pt_{\eta_k}^k(\langle \Delta_k, \ell', v' \rangle)$ was not, where $0 \leq k < j$, then SUPA will also fail for $pt_*^k(\langle \Delta_k, \ell', v' \rangle)$, where $i \leq k < j$, since $* \leq \eta_k$. Therefore, SUPA will exploit the second type of reuse by setting $pt_*^i(\langle \Delta_i, \ell', v' \rangle) = pt_{\eta_j}^j(\langle \Delta_j, \ell', v' \rangle)$.

Of course, many other schemes are possible with or without precision loss.

5. EVALUATION

We evaluate SUPA by choosing detection of uninitialized pointers as a major client. The objective is to show that SUPA is effective in answering client queries, in environments with small time and memory budgets such as IDEs, by facilitating efficiency and precision tradeoffs in a hybrid multi-stage analysis framework. 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.

5.1. Implementation

We have implemented SUPA in LLVM (3.5.0). The source files of a program are compiled under “-O0” (to facilitate detection of undefined values [Zhao et al. 2012]) 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 [Pearce et al. 2007]. Arrays are considered monolithic so that the elements in an array are not distinguished. Distinct allocation sites (i.e., `ADDR_OF` statements) are modeled by distinct abstract objects.

We build the SVFG for a program based on our open-source software, SVF [Sui and Xue 2016]. 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 [Hardekopf and Lin 2011] also in LLVM, as SFS is a recent solution yielding exactly the flow-sensitive precision with good scalability. However, there are some differences. In [Hardekopf and Lin 2011], SFS was imple-

mented 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 (3.5.0), by building a program’s call graph on the fly (Section 4.1) 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 [Acharya and Robinson 2011], 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.

5.2. Methodology

We choose uninitialized pointer detection as a major client, named Uninit, which requires strong update analysis to be effective. 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.

We will show that SUPA can answer Uninit’s on-demand queries efficiently while achieving nearly the same precision as SFS. For C, global and static variables are default initialized, but local variables are not. In order to mimic the default uninitialized state 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 uninitialized state. 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 III.

5.3. Experimental Setup

We use a machine with a 3.7GHz Intel Xeon 8-core CPU and 64 GB memory. As shown in Table III, 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 III 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 5.2).

Table III: Program characteristics.

Program	KLOC	Statements	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

5.4. Results and Analysis

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 applied in that order.

5.4.1. 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 14(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 15 and explained below).

For SUPA, its pre-analysis is lightweight, as shown in Table IV, 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 14(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

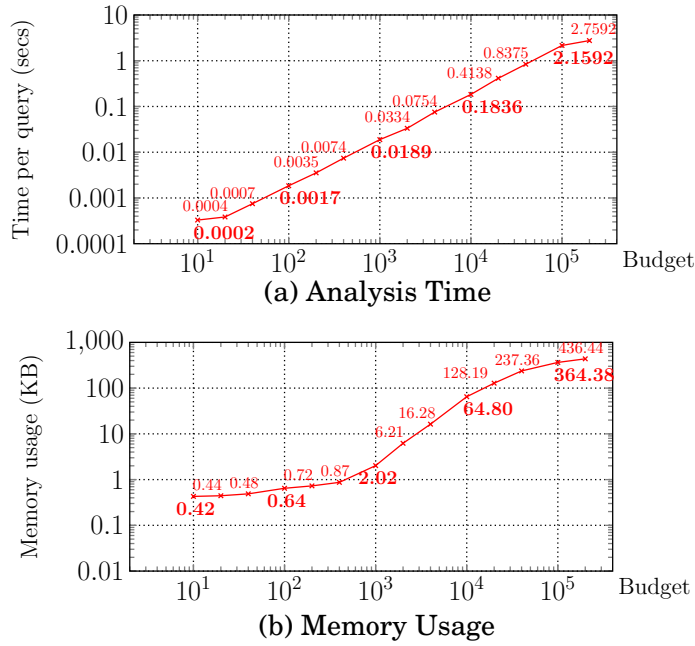


Fig. 14: Average analysis time and memory usage per query consumed by SUPA-FS under different analysis budgets (with both axes being logarithmic).

(/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.

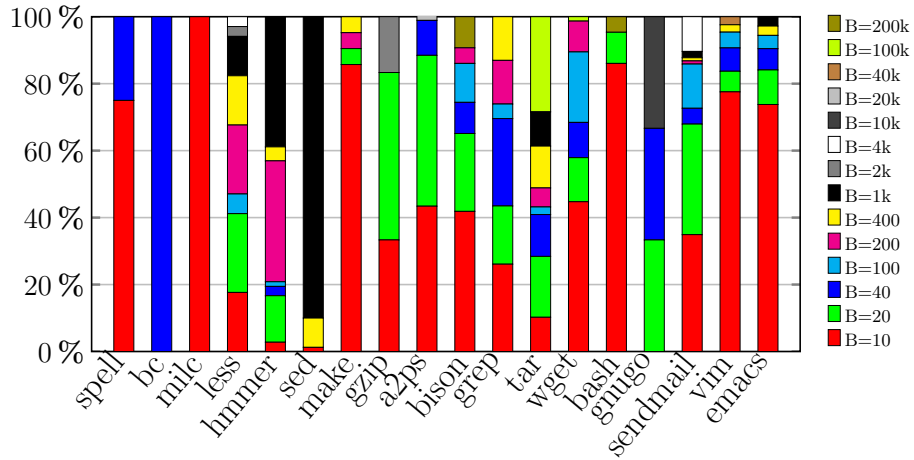


Fig. 15: Percentage of queried variables proved to be initialized by SUPA-FS over SFS under different budgets.

Table IV: 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’s Analysis	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
hmmmer	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

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 15 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 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.

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 16 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.

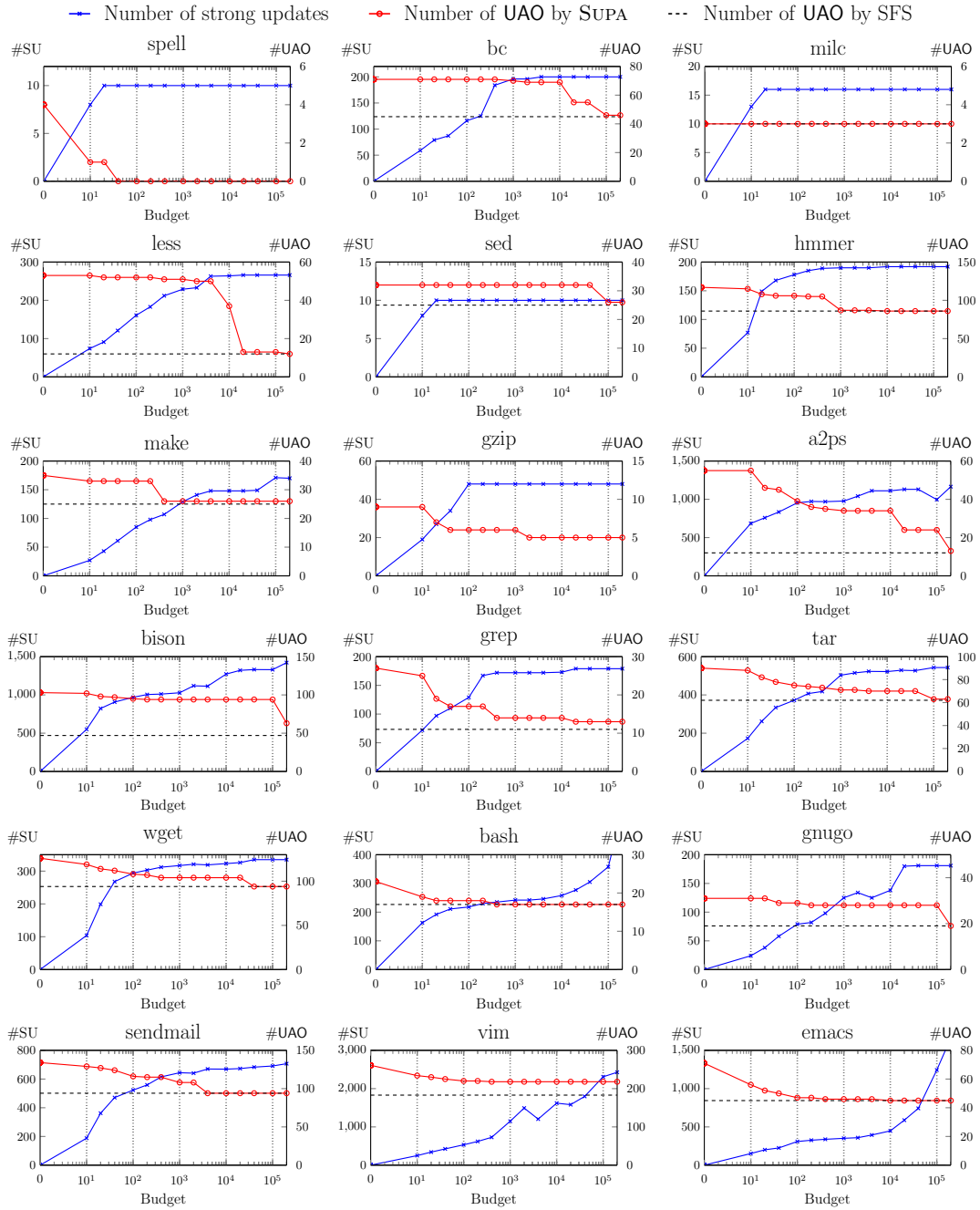


Fig. 16: Correlating the number of strong updates with the number of UAO's under SUPA-FS with different analysis budgets.

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, respectively.

For some programs such as `spell`, `bc`, `milc`, `hmmmer` and `grep`, most of their strong updates happen under small budgets (e.g., $B = 1000$). In `hmmmer`, 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 4.3. 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 .

5.4.2. Evaluating SUPA-FSCS. 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 5.1).

Table V: 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
<code>spell</code>	0.01	0	0.01	0
<code>bc</code>	18.35	69	287.23	69
<code>milc</code>	0.02	3	14.52	0
<code>less</code>	15.15	37	92.41	37
<code>sed</code>	355.60	32	4725.42	32
<code>hmmmer</code>	11.41	86	135.05	71
<code>make</code>	124.40	26	229.44	26
<code>gzip</code>	0.64	5	4.28	5
<code>a2ps</code>	126.01	34	448.26	32
<code>bison</code>	465.54	94	529.20	86
<code>grep</code>	124.46	14	197.66	14
<code>tar</code>	26.31	70	83.10	68
<code>wget</code>	24.51	104	84.90	104
<code>bash</code>	188.69	17	327.16	17
<code>gnugo</code>	72.73	28	80.08	27
<code>sendmail</code>	200.32	94	250.19	85
<code>vim</code>	168.67	218	473.25	218
<code>emacs</code>	159.22	45	222.65	45

In this section, we demonstrate that SUPA can exploit both flow- and context-sensitivity effectively *on-demand* in a hybrid multi-stage analysis framework, providing improved precision needed by some programs. Table V 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`, `hmmmer`, `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 4.2, by eliminating 8 UAO's in `bison`, 1 in `tar` and 1 in `sendmail`, which have been reported by SUPA-FS.

6. CASE STUDIES

<pre> 114 // symtab.c 115 static 116 void symbols_sort(symbol **first, symbol **second) { 117 ... 118 symbol* tmp = *first; 119 *first = *second; 120 121 *second = tmp; 122 ... 123 } 124 623 static void 624 user_token_number_redeclaration(...) { 125 ... 627 symbols_sort (&st, &nd); 126 ... 628 complain_indent (&nd->location, ...); 127 ... 635 } </pre> <p style="text-align: center;">(a) Code snippet from bison-3.0.4</p>	<pre> // 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; 82 ... 84 m->m_ifile = curr_ifile; 85 break; 108 case '\n': 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 } </pre> <p style="text-align: center;">(b) Code snippet from less-4.5.1</p>
<pre> //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 } </pre> <p style="text-align: center;">(c) Code snippet from milc-v6</p>	<pre> //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 } </pre> <p style="text-align: center;">(d) Code snippet from tar-1.28</p>

Fig. 17: Selected code snippets.

We examine some real code to see how client queries are answered precisely with on-demand strong updates under four different scenarios.

Figure 17(a). There is a swap from `bison`. In line 121, `second` points to a singleton stack object `nd` passed from line 627. So a strong update is applied. When querying `nd->location` in line 628, SUPA knows that `nd` points to what `st` pointed to before.

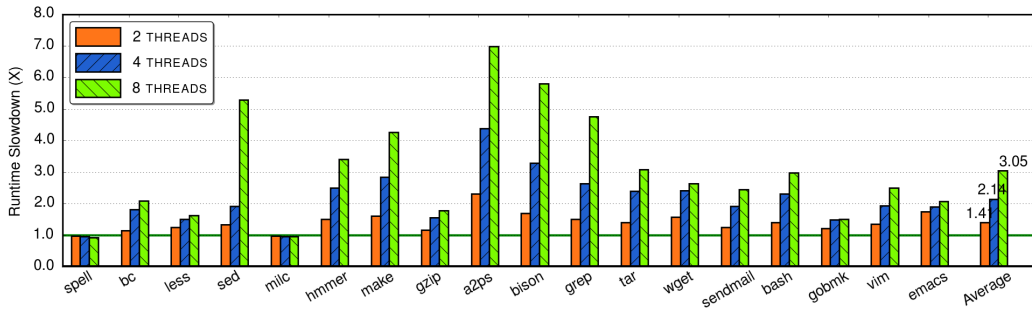


Fig. 18: Speedups of SUPA-FS when parallelized over its sequential version with two, four and eight threads ($B = 10000$).

Figure 17(b). In the code fragment from `less`, `m->mifile` 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->mifile` in line 208 is initialized.

Figure 17(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 17(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.

7. PARALLELIZING SUPA

To demonstrate that SUPA is amenable to parallelization as a demand-driven analysis, we have parallelized SUPA-FS by using Intel Threading Building Blocks (TBB). A *concurrent_queue* is used to store all the queries issued from a program. We use a *task_group* to allocate tasks for computing the queries from *concurrent_queue* in parallel. The cached points-to information is shared with a *concurrent_hash_map*.

Figure 18 shows the speedups achieved by parallelization over the sequential setting with $B = 10000$. With eight threads, the average speedup for the 18 programs is 3.05x and the maximum speedup observed is 6.9x at `a2ps`. The time for each setting excludes the pre-analysis time. Some programs enjoy better speedups than others. There are three main reasons. First, some programs, such as `spell`, `less` and `milc`, have relatively few queries issued. Therefore, the performance benefits achieved from query parallelization can be small. Second, different queries take different times to answer, resulting in different degrees of workload imbalance in different programs. Third, different programs suffer from different synchronization overheads in accessing the cached points-to information in *concurrent_hash_map*.

8. 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). Below we limit our discussion to the work that is most relevant to SUPA.

8.1. 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. [Choi et al. \[1993\]](#) and [Emami and Hendren \[1994\]](#) gave some formulations in an iterative data-flow framework [[Kam and Ullman 1977](#)]. [Wilson and Lam \[1995\]](#) 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 [[Hardekopf and Lin 2009, 2011](#); [Oh et al. 2012](#); [Yu et al. 2010](#)], 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) [[Choi et al. 1991](#); [Ramalingam 2002](#)] as in [[Hind and Pioli 1998](#)] and various SSA representations such as HSSA [[Chow et al. 1996](#)], partial SSA [[Lattner and Adve 2004](#)] and SSI [[Ananian 1999](#); [Tavares et al. 2014](#)]. 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 [[Hardekopf and Lin 2009](#)]. Later, the idea of staged analysis [[Fink et al. 2008](#)] 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 [[Hardekopf and Lin 2011](#); [Sui et al. 2016a](#); [Ye et al. 2014b](#)]. 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 [[Balatsouras and Smaragdakis 2016](#)] 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.

8.2. Demand-Driven Pointer Analysis

Demand-driven pointer analyses for C [[Heintze and Tardieu 2001](#); [Zhang et al. 2014a](#); [Zheng and Rugina 2008](#)] and Java [[Lu et al. 2013](#); [Shang et al. 2012](#); [Sridharan and Bodík 2006](#); [Su et al. 2016](#); [Yan et al. 2011](#)] are flow-insensitive, formulated in terms of CFL (Context-Free-Language) reachability [[Reps et al. 1995](#)]. [Heintze and Tardieu \[2001\]](#) introduced the first on-demand Andersen-style pointer analysis for C. Later, [Zheng and Rugina \[2008\]](#) 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 [[Sridharan et al. 2005](#)] and later with context-sensitivity [[Sridharan and Bodík 2006](#)]. [Shang et al. \[2012\]](#) and [Yan et al. \[2011\]](#) investigated how to summarize points-to information discovered during the CFL-reachability analysis to improve performance for Java programs. [Lu et al. \[2013\]](#) introduced an incremental pointer analysis with a CFL-reachability formulation for Java. [Su et al. \[2014\]](#) demonstrated that the CFL-reachability formulation is highly amenable to parallelization on multi-core CPUs. Recently, [Feng et al. \[2015\]](#) focused on answering demand queries for Java programs in a context-sensitive analysis framework (without performing strong updates). Unlike these flow-

insensitive analyses, which are not effective for many clients like Uunit, SUPA can perform strong updates on-demand flow and context-sensitively.

BOOMERANG [Späth et al. 2016] represents a 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, where a and b are different top-level variables. Let us explain this by using the following straight-line Java code and its corresponding C code.

Java Code	C Code
ℓ_1 : $q = \text{new } A()$ // o1	ℓ_1 : $q = \text{malloc}()$ // o1
ℓ_2 : $p = q$	ℓ_2 : $p = q$
ℓ_3 : $p.f = \text{new } A()$ // o2	ℓ_3 : $*p = \text{malloc}()$ // o2
ℓ_4 : $q.f = \text{new } A()$ // o3	ℓ_4 : $*q = \text{malloc}()$ // o3
ℓ_5 : $x = p.f$	ℓ_5 : $x = *p$

Let us consider BOOMERANG first. At ℓ_3 , a strong update is performed to $p.f$ to make it point to $o2$ only. At ℓ_4 , a strong update is performed to $q.f$ to make it point to $o3$ but a weak update is performed to all its aliases so that $p.f$ now points to not only $o2$ as before but also $o3$. As a result, x points-to both $o2$ and $o3$ at ℓ_5 . Let us consider now SUPA. With both flow- and context-sensitivity enforced, a strong update is performed to $o1$ pointed p and q at both ℓ_3 and ℓ_4 , respectively. Thus, x points to $o3$ only at ℓ_5 .

8.3. Hybrid Pointer Analysis

The basic idea is to find a right balance between efficiency and precision. For C programs, the one-level approach [Das 2000] 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 [Kastrinis and Smaragdakis 2013]. In [Guyer and Lin 2003], how to adjust the analysis precision according to a client’s needs is discussed. Zhang et al. [2014b] focus on finding effective abstractions for whole-program analyses written in Datalog via abstraction refinement. Lhoták and Chung [Lhoták and Chung 2011] 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 in a hybrid multi-stage analysis framework. Unlike [Lhoták and Chung 2011], SUPA can achieve the same precision as whole-program flow-sensitive analysis, subject to a given budget.

8.4. Parallel Pointer Analysis

Méndez-Lojo et al. [2010] introduced a parallel implementation of Andersen’s analysis for C based on graph rewriting. Their parallel analysis is flow- and context-insensitive, achieving a speedup of up to 3X on an 8-core CPU. Su et al. [2016] introduces an improvement of this parallel implementation on GPUs. The whole-program sparse flow-sensitive pointer analysis [Hardekopf and Lin 2009] has also been parallelized on multi-core CPUs [Nagaraj and Govindarajan 2013] and GPUs [Nasre 2013]. The speedups are up to 2.6X on a 8-core CPU. This paper presents the first parallel implementation of demand-driven pointer analysis with strong updates for C programs, achieving an average speedup of 3.05X on a 8-core CPU.

9. CONCLUSION

We have introduced, SUPA, a demand-driven pointer analysis that enables computing precise points-to information for C programs flow- and context-sensitively with strong

updates by refining away imprecisely pre-computed value-flows, subject to some analysis budgets. SUPA handles large C programs effectively by allowing pointer analyses with different efficiency and precision tradeoffs to be applied in a hybrid multi-stage analysis framework. SUPA is particularly suitable for environments with small time and memory budgets such as IDEs. We have evaluated SUPA by choosing uninitialized pointer detection as a major client on 18 C programs. SUPA can achieve nearly the same precision as whole-program flow-sensitive analysis under small budgets.

One interesting future work is to investigate how to allocate budgets in SUPA to its stages to improve the precision achieved in answering some time-consuming queries for a particular client. Another direction is to add more stages to its analysis, by considering, for example, path correlations.

REFERENCES

- Acharya, M. and Robinson, B. (2011). Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE '11*, pages 746–755.
- Ananian, C. S. (1999). *The static single information form*. PhD thesis, Master's Thesis, MIT.
- Andersen, L. (1994). *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen.
- Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Oceau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI '14*, pages 259–269.
- Balatsouras, G. and Smaragdakis, Y. (2016). Structure-sensitive points-to analysis for c and c++. In *SAS '16*.
- Blackshear, S., Chang, B.-Y. E., and Sridharan, M. (2013). Thresher: Precise refutations for heap reachability. In *PLDI '13*, pages 275–286.
- Choi, J.-D., Burke, M., and Carini, P. (1993). Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93*, pages 232–245.
- Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic construction of sparse data flow evaluation graphs. In *POPL '91*, pages 55–66.
- Chow, F., Chan, S., Liu, S., Lo, R., and Streich, M. (1996). Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267.
- Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. (1991). Efficiently computing static single assignment form and the control dependence graph. *TOPLAS'91*, 13(4):490.
- Das, M. (2000). Unification-based pointer analysis with directional assignments. In *PLDI '00*, pages 35–46.
- Emami, M., Ghiya, R. and Hendren, J. (1994). Context-sensitive interprocedural points-to analysis in presence of function pointers. In *PLDI '94*, pages 242–256.
- Feng, Y., Wang, X., Dillig, I., and Lin, C. (2015). EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. In *OOPSLA '15*, pages 520–534.
- Fink, S. J., Yahav, E., Dor, N., Ramalingam, G., and Geay, E. (2008). Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2):9.
- Guyer, S. Z. and Lin, C. (2003). Client-driven pointer analysis. In *SAS '03*, pages 1073–1073.
- Hardekopf, B. and Lin, C. (2009). Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238.
- Hardekopf, B. and Lin, C. (2011). Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *CGO '11*, pages 289–298.

- Heintze, N. and Tardieu, O. (2001). Demand-driven pointer analysis. In *PLDI '01*, pages 24–34.
- Hind, M. and Pioli, A. (1998). Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98*, pages 57–81.
- ISO90 (1990). ISO/IEC. international standard ISO/IEC 9899, programming languages - C.
- Kam, J. B. and Ullman, J. D. (1977). Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317.
- Kastrinis, G. and Smaragdakis, Y. (2013). Hybrid context-sensitivity for points-to analysis. In *PLDI '13*, pages 423–434.
- Landi, W. (1992). Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337.
- Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86.
- Lhoták, O. and Chung, K.-C. A. (2011). Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16.
- Lhoták, O. and Hendren, L. (2003). Scaling Java points-to analysis using Spark. *CC '03*, pages 153 – 169.
- Li, L., Cifuentes, C., and Keynes, N. (2011). Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE '11*, pages 343–353.
- Li, Y., Tan, T., Sui, Y., and Xue, J. (2014). Self-inferencing reflection resolution for Java. In *ECOOP '14*, pages 27–53.
- Lu, Y., Shang, L., Xie, X., and Xue, J. (2013). An incremental points-to analysis with CFL-reachability. In *CC'13*.
- Méndez-Lojo, M., Mathew, A., and Pingali, K. (2010). Parallel inclusion-based points-to analysis. In *OOPSLA '10*, pages 428–443.
- Milanova, A., Rountev, A., and Ryder, B. G. (2002). Parameterized object sensitivity for points-to and side-effect analyses for Java. *ISSTA '02*.
- Milanova, A., Rountev, A., and Ryder, B. G. (2005). Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41.
- Nagaraj, V. and Govindarajan, R. (2013). Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT '13*, pages 19–28.
- Nasre, R. (2013). Time- and space-efficient flow-sensitive points-to analysis. *TACO '13*, 10(4):39:1–39:27.
- Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and implementation of sparse global analyses for C-like languages. In *PLDI '12*, pages 229–238.
- Pearce, D., Kelly, P., and Hankin, C. (2007). Efficient field-sensitive pointer analysis of C. *ACM TOPLAS*, 30(1):4–es.
- Ramalingam, G. (1994). The undecidability of aliasing. *ACM TOPLAS*, 16(5):1467–1471.
- Ramalingam, G. (2002). On sparse evaluation representations. *Theoretical Computer Science*, 277(1):119–147.
- Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61.
- Shang, L., Xie, X., and Xue, J. (2012). On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274.
- Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick your contexts well: understanding object-sensitivity. In *POPL'11*, pages 17–30.
- Späth, J., Do, L. N. Q., Ali, K., and Bodden, E. (2016). Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. *ECOOP*.
- Sridharan, M. and Bodík, R. (2006). Refinement-based context-sensitive points-to analysis for Java. *PLDI '06*, pages 387–400.

- Sridharan, M., Gopan, D., Shan, L., and Bodík, R. (2005). Demand-driven points-to analysis for Java. In *OOPSLA '05*, pages 59–76.
- Su, Y., Ye, D., and Xue, J. (2014). Parallel pointer analysis with cfl-reachability. In *ICPP '14*, pages 451–460.
- Su, Y., Ye, D., Xue, J., and Liao, X. (2016). An efficient GPU implementation of inclusion-based pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 27(2):353–366.
- Sui, Y., Di, P., and Xue, J. (2016a). Sparse flow-sensitive pointer analysis for multi-threaded programs. In *CGO '16*, pages 160–170. ACM.
- Sui, Y., Fan, X., Zhou, H., and Xue, J. (2016b). Loop-oriented array-and field-sensitive pointer analysis for automatic simd vectorization. In *LCTES '16*, pages 41–51. ACM.
- Sui, Y., Li, Y., and Xue, J. (2013). Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11.
- Sui, Y. and Xue, J. (2016). SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266.
- Sui, Y., Ye, D., and Xue, J. (2012). Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264.
- Sui, Y., Ye, D., and Xue, J. (2014a). Detecting memory leaks statically with full-sparse value-flow analysis. *TSE '14*, 40(2):107–122.
- Sui, Y., Ye, S., Xue, J., and Zhang, J. (2014b). Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *SPE '14*, 44(12):1485–1510.
- Sun, Q., Zhao, J., and Chen, Y. (2011). Probabilistic points-to analysis for Java. In *CC '11*, pages 62–81.
- SUPA (2016). SUPA. <http://www.cse.unsw.edu.au/~corg/supa>.
- Tavares, A., Boissinot, B., Pereira, F., and Rastello, F. (2014). Parameterized construction of program representations for sparse dataflow analyses. In *CC '14*, pages 18–39. Springer.
- Wilson, R. and Lam, M. (1995). Efficient context-sensitive pointer analysis for C programs. *PLDI '95*, pages 1–12.
- Xiao, X. and Zhang, C. (2011). Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA '11*, pages 188–198.
- Yan, D., Xu, G., and Rountev, A. (2011). Demand-driven context-sensitive alias analysis for Java. In *ISSTA '11*, pages 155–165.
- Yan, H., Sui, Y., Chen, S., and Xue, J. (2016). Automated memory leak fixing on value-flow slices for c programs. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16*, pages 1386–1393, New York, NY, USA. ACM.
- Ye, D., Sui, Y., and Xue, J. (2014a). Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In *CGO '14*.
- Ye, S., Sui, Y., and Xue, J. (2014b). Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336.
- Yu, H., Xue, J., Huo, W., Feng, X., and Zhang, Z. (2010). Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229.
- Zhang, Q., Xiao, X., Zhang, C., Yuan, H., and Su, Z. (2014a). Efficient subcubic alias analysis for C. In *PLDI '14*, pages 829–845.
- Zhang, X., Mangal, R., Grigore, R., Naik, M., and Yang, H. (2014b). On abstraction refinement for program analyses in Datalog. In *PLDI '14*, pages 239–248.
- Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2012). Formalizing the LLVM intermediate representation for verified program transformations. In *POPL '12*, pages 427–440.
- Zheng, X. and Rugina, R. (2008). Demand-driven alias analysis for C. In *POPL '08*, pages 197–208.