✓ Artifact evaluated by FSE

On-Demand Strong Update Analysis via Value-Flow Refinement

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

ABSTRACT

We present a new Strong UPdate Analysis for C programs, called Supa, that enables computing points-to information on-demand 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 a value-flow graph representation of the program, so that strong updates are performed where needed, as long as the total analysis budget is not exhausted. Supa facilitates efficiency and precision tradeoffs by allowing different pointer analyses to be applied in a hybrid multi-stage analysis framework.

We have implemented Supa in LLVM and evaluated Supa by choosing uninitialized pointer detection as a major client on 12 open-source C programs. As the analysis budget increases, Supa achieves improved precision, with its single-stage flow-sensitive analysis reaching 97% of that achieved by whole-program flow-sensitive analysis by consuming about 0.19 seconds and 36KB of memory per query, on average (with a budget of at most 10000 value-flow edges per query).

CCS Concepts

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

Keywords

strong updates, value flow, pointer analysis, flow sensitivity

1. INTRODUCTION

Strong updates, where stores overwrite, i.e., kill the previous contents of their abstract destination locations with new values, is an important factor in the precision of pointer analysis [14, 15, 23]. In the case of *weak updates*, these locations are assumed conservatively to also retain their old contents.

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. A flow-sensitive analysis can strongly update an abstract location written at a store if and only if that location refers to exactly one concrete memory address. By applying strong updates where needed, an analysis can improve precision, thereby providing significant benefits to many clients, such as change impact analysis [2], bug detection [58], security analysis [4], type state verification [12], compiler optimization, and symbolic execution [5].

In this paper, we investigate how to perform strong updates effectively in analyzing large C programs, for which flow-sensitivity is important in achieving the precision required by the afore-mentioned client applications. For object-oriented languages like Java, context-sensitivity is essential in achieving useful precision [24, 26, 27, 28, 32, 33, 42, 53, 54, 56].

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 ignored, this problem is intractable due to potentially unbounded number of paths that must be analyzed [21, 38].

Instead, traditional flow-sensitive pointer analysis (FS) for C [17, 18] 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. Recently, sparse flow-sensitive pointer analysis (SFS) [15, 25, 35, 59, 60] 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.

Recently, an approach [23] 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 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 [23] are less precise, due to both missing strong updates and propagating the points-to information flow-insensitively along all the weakly-updated (abstract) locations.

In practice, a client application may require only parts of the program to be analyzed. In addition, some queries may demand precise answers while others can be answered as

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950296

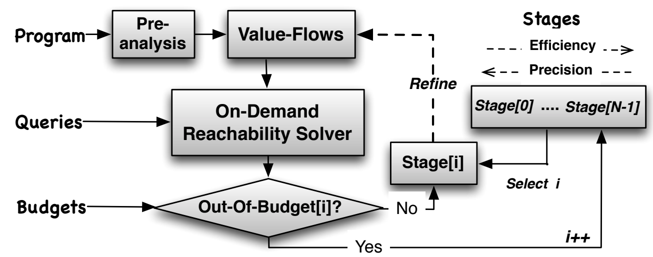


Figure 1: Overview of Supa

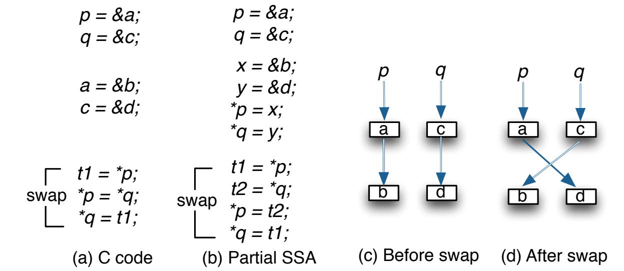


Figure 2: A swap example and its partial SSA (with the points-to relations for p and q at run time)

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? Existing demand-driven analyses for C [16, 61, 64] and Java [29, 41, 44, 47, 57] are flow-insensitive and thus cannot perform strong updates to produce the precision needed by some clients. In addition, recent advances in whole-program flow-sensitive analysis for C have exploited some form of sparsity to improve performance [15, 25, 35, 59, 60]. However, how to replicate this success for demand-driven flow-sensitive analysis 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 value-flow based demand-driven **S**trong **UP**date **A**nalysis for C, designed to support flexible yet effective tradeoffs between efficiency and precision in answering client queries, in environments with small time and memory budgets such as IDEs. As shown in Figure 1, its novelty 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 inter-procedural value-flow graph that captures the def-use chains of the program obtained conservatively by a pre-analysis. Such over-approximated value-flows can be obtained by applying Andersen's analysis [3] (flow-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 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 final results are.

Supa handles large programs by staging analyses in increasing efficiency but decreasing precision in a hybrid manner. Presently, 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 alloted 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.

This paper makes the following contributions:

- We present the first strong update analysis for C that enables computing precise points-to information on-demand, with strong updates applied where needed, 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 with its artifact available at [1]. We choose uninitialized pointer detection as a practical client using 12 open-source C programs. As the analysis budget increases, Supa achieves improved precision, with its single-stage flow-sensitive analysis reaching 97% of that achieved by whole-program flow-sensitive analysis, by consuming about 0.19 seconds and 36KB of memory per query, on average (with a per-query budget of at most 10000 value-flow edges traversed).

2. BACKGROUND

We describe the partial SSA form used for representing a C program and the sparse value-flow graph used for representing conservatively its value-flows, i.e., def-use chains.

2.1 Partial SSA Form

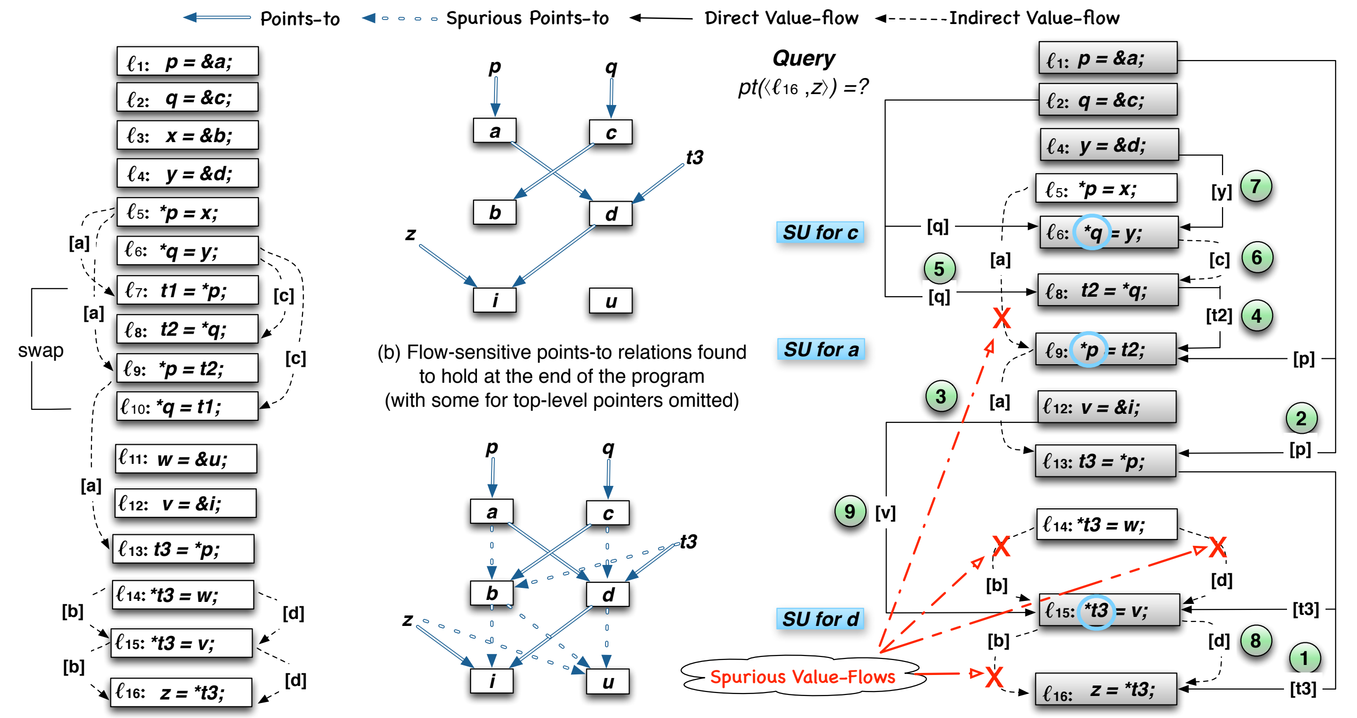
We represent a program by putting it into LLVM's partial SSA form, following [15, 23, 25, 59, 50]. The set of all variables $\mathcal{V}$ are separated into two subsets: $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer and $\mathcal{T}$ containing all *top-level variables*, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$.

After the SSA conversion, a program is represented by five types of statements: $p = \&a$ (AddrOf), $p = q$ (Copy), $p = *q$ (Load), $*p = q$ (Store), and $p = \phi(q, r)$ (Phi). Top-level variables are put directly in SSA form, while address-taken variables are accessed indirectly via Load or Store. Passing arguments into and returning results from functions are modeled by copies. For an AddrOf statement $p = \&a$, known as an *allocation site*, a is a stack or global variable or a dynamically created abstract heap object.

Figure 2 shows a swap program in C and its corresponding partial SSA form, where $p, q, x, y, t1, t2 \in \mathcal{T}$ and $a, b, c, d \in \mathcal{A}$. Here, $x, y, t1$ and $t2$ are new temporaries introduced.

2.2 Sparse Value-Flow Graph

Given a program in partial SSA form, a *sparse value-flow graph* (SVFG) $G = (N, E)$ is a multi-edged directed graph



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

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

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

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

Figure 3: A motivating example for illustrating the Supa analysis (with SU standing for "Strong Update")

that captures its def-use chains conservatively. N is the set of nodes representing all statements and E is the set of edges representing all potential def-use chains. In particular, an edge $\ell_1 \xrightarrow{v} \ell_2$, where $v \in \mathcal{V}$, from statement ℓ_1 to statement ℓ_2 signifies a potential def-use chain for v with its def at ℓ_1 and use at ℓ_2. This representation is sparse since the intermediate program points between ℓ_1 and ℓ_2 are omitted.

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

As address-taken variables are not (yet) in SSA form, their indirect uses at loads may be defined indirectly at multiple stores. We can build their def-use chains in several steps by following [15, 51], with an illustrating example given in Section 3. First, the points-to information in the program is computed by a pre-analysis. Second, a load $p = *q$ is annotated with a function $\mu(a)$ for each variable $a \in \mathcal{A}$ that may be pointed to by q to represent a potential use of a at the load. Similarly, a store $*p = q$ is annotated with a function $a = \chi(a)$ for each variable $a \in \mathcal{A}$ that may be pointed to by p to represent a potential def and use of a at the store. If a can be strongly updated, then a receives whatever q points to and the old contents in a are killed. Otherwise, a must also incorporate its old contents, resulting in a weak update to a. Third, we convert all the address-taken variables into SSA form, with each $\mu(a)$ treated as a use of a and each $a = \chi(a)$ as both a def and use of a. Finally, we obtain the indirect def-use chains for an address-taken variable $a \in \mathcal{A}$ as follows. For a use of a identified as a_n (with its version identified by n) at a load or store ℓ, its unique definition in SSA form is a_n at a store ℓ'. Then, an indirect def-use chain $\ell' \xrightarrow{a} \ell$ is added to represent potentially the *indirect value-flow* of a from ℓ' to ℓ. Note that the ϕ functions introduced for address-taken variables will now be ignored as the value a that appears in $\ell' \xrightarrow{a} \ell$ is not versioned.

3. A MOTIVATING EXAMPLE

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

Figure 3(b) depicts the points-to relations for the six address-taken variables and some top-level ones found at the *end* of the code sequence by a whole-program flow-sensitive analysis (with strong updates) like SFS [15]. Due to flow-sensitivity, multiple solutions for a pointer are maintained. In this example, these are the true relations observed at the end of program execution. Note that SFS gives rise to Figure 2(c) by analyzing $\ell_1 - \ell_6$, Figure 2(d) by analyzing also $\ell_7 - \ell_{10}$, and finally, Figure 3(b) by analyzing $\ell_{11} - \ell_{16}$ further. As z points to i but not u, no warning is issued for z.

Figure 3(c) shows how the points-to relations in Figure 3(b) are over-approximated flow-insensitively by applying Andersen's analysis [3]. In this case, a single solution is computed conservatively for the entire program. Due to the lack of strong updates in analyzing the two stores performed by swap, the points-to relations in Figures 2(c) and 2(d) are merged, causing $*a$ and $*c$ to become spurious aliases. When ℓ_{11} – ℓ_{16} are analyzed, the seven spurious points-to relations (shown in dashed arrows in Figure 3(c)) are introduced. Since z points to i (correctly) and u (spuriously), a false positive for z will be issued. Failing to consider flow-sensitivity, Andersen's analysis is not precise for this 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 3(a). For its top-level variables, their direct value-flows, i.e., def-use chains are explicit and thus omitted to avoid cluttering. For example, q has three def-use chains $\ell_2 \xrightarrow{q} \ell_6$, $\ell_2 \xrightarrow{q} \ell_8$ and $\ell_2 \xrightarrow{q} \ell_{10}$. For its address-taken variables, we first apply Andersen's analysis to find flow-insensitively their points-to relations, which are given in Figure 3(c). We then obtain the nine indirect value-flows, i.e., def-use chains depicted in Figure 3(a), as described in Section 2. 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 ***Stage[N-1]*** = ***Stage[0]*** = FS in Figure 1. Figure 3(d) shows how Supa computes $pt(\langle \ell_{16}, z\rangle)$ on-demand, starting from ℓ_{16}, by performing a backward reachability analysis on the SVFG, with the visiting order of def-use chains marked as ① – ⑨. Formally, this is done in Figure 5. The def-use chains for only the relevant top-level variables are shown. By filtering out the four spurious value-flows (marked by ×), Supa finds that only i at ℓ_{12} is backward reachable from z at ℓ_{16}. Thus, $pt(\langle \ell_{16}, z\rangle) = \{i\}$. So no warning for z will be issued.

Supa differs from prior work in three major aspects:

- ***On-Demand Strong Updates***

  A whole-program flow-sensitive analysis like SFS [15] can answer $pt(\langle \ell_{16}, z\rangle)$ precisely but must accomplish this task by analyzing all the 16 statements, resulting in six strong updates at the six stores, with some done unnecessarily for this query. Unfortunately, existing whole-program FSCS or even just FS algorithms do not scale well for large C programs [2].

  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 Supa performs a strong update during its reachability analysis, the fewer the number of statements traversed. After ① – ⑧, Supa finds that $t3$ points to d only. With a strong update done at ℓ_{15} : $*t3 = v$ (⑨), Supa concludes that $pt(\langle \ell_{16}, z\rangle) = \{i\}$.

- ***Value-Flow Refinement***

  Existing demand-driven analyses [41, 44, 57, 61, 64] 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 positive for z is issued, as discussed earlier.

  However, Supa performs strong updates flow-sensitively by filtering out the four spurious pre-computed value-flows marked by ×. As $t3$ points to d only, $\ell_{15} \xrightarrow{b} \ell_{16}$ is spurious and not traversed. In addition, a strong update is enabled at ℓ_{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 done 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 $pt(\langle \ell_{16}, z\rangle)$ in, say, three steps, Supa will stop its traversal from ℓ_9 to ℓ_8 (at ④) in Figure 3(d) and falls 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 on-demand strong update analysis (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 examine our hybrid multi-stage analysis framework (Section 4.3). All our analyses are field-sensitive, as discussed in Section 5.1.

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 as a pre-analysis, as discussed in Section 2.2 and illustrated in Section 3.

Let $\mathbb{V} = \mathcal{L} \times \mathcal{V}$ be the set of labeled variables lv, where $\mathcal{L}$ is the set of statement labels and $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$. Supa conducts a backward reachability analysis flow-sensitively on G_{vfg} by computing a reachability relation, $\hookleftarrow \subseteq \mathbb{V} \times \mathbb{V}$. In our formalism, $\langle \ell, v\rangle \hookleftarrow \langle \ell', v'\rangle$ signifies a value-flow from a def of v' at ℓ' to a use of v at ℓ through one or multiple value-flow paths in G_{vfg}. For an object o created at an AddrOf statement, i.e., an allocation site at ℓ', identified as $\langle \ell', o\rangle$, we must distinguish it from $\langle \ell, o\rangle$ accessed elsewhere at ℓ in our inference rules. Our abbreviation for $\langle \ell', o\rangle$ is $\widehat{o}$.

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

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

Despite flow-sensitivity, our formalization in Figure 4 makes no explicit references to program points. As Supa operates on the def-use chains in G_{vfg}, each variable $\langle \ell, v\rangle$ mentioned in a rule appears at the point just after ℓ, where v is defined.

Let us examine our rules in detail. By [ADDR], an object $\widehat{o}$ created at an allocation site ℓ is backward reachable from p at ℓ (or precisely at the point after ℓ). The pre-computed direct value-flows across the top-level variables in G_{vfg} are always precise ([COPY] and [PHI]). In partial SSA form, [PHI] exists only for top-level variables (Section 2.2).

However, the indirect value-flows across the address-taken variables in G_{vfg} can be imprecise; they need to be refined on

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

$$\texttt{[LOAD]}\ \frac{\ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle \ell'', q\rangle \hookleftarrow \hat{o} \quad \ell' \xrightarrow{o} \ell}{\langle \ell, p\rangle \hookleftarrow \langle \ell', o\rangle} \qquad \texttt{[STORE]}\ \frac{\ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle \ell'', p\rangle \hookleftarrow \hat{o} \quad \ell' \xrightarrow{q} \ell}{\langle \ell, o\rangle \hookleftarrow \langle \ell', q\rangle}$$

$$\texttt{[SU/WU]}\ \frac{\ell : *p = \_ \quad \ell' \xrightarrow{o} \ell \quad o \in \mathcal{A} \setminus kill(\ell, p)}{\langle \ell, o\rangle \hookleftarrow \langle \ell', o\rangle} \qquad \texttt{[COMPO]}\ \frac{lv \hookleftarrow lv' \quad lv' \hookleftarrow lv''}{lv \hookleftarrow lv''}$$

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

Figure 4: Single-stage flow-sensitive Supa analysis with demand-driven strong updates

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

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

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

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

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

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

Figure 5: Reachability derivations for $pt(\langle \ell_{16}, z\rangle)$ shown in Figure 3(d) (with reuse of cached points-to results inside each box)

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 $\langle \ell'', q\rangle \hookleftarrow \hat{o}$ 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 $\cdots = *q$ is aliased with $*p$ in a store $*p = \cdots$ 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 $\ell : p = \_$. 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 [23]. 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, $\langle \ell, o\rangle \hookleftarrow \langle \ell', o\rangle$ is falsified. In Case (2), the points-to set of p is empty. Again, further traversal to $\langle \ell', o\rangle$ must be prevented to avoid dereferencing a null pointer as is standard [14, 15, 23]. In Case (3), a weak update is performed to o so that its old contents at ℓ' are preserved. Thus, $\langle \ell, o\rangle \hookleftarrow \langle \ell', o\rangle$ is established, which implies that the backward traversal along $\ell' \xrightarrow{o} \ell$ must continue.

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

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

Example 1. *Figure 5 shows how we apply the rules of* Supa *to answer* $\mathrm{pt}(\langle \ell_{16}, z\rangle)$ *illustrated in Figure 3(d).* `[SU/WU]` *(implicit in these derivations) is applied to ℓ_6, ℓ_9 and ℓ_{15} to cause a strong update at each store. At ℓ_6, $\mathrm{pt}(\langle \ell_6, q\rangle) = \{c\}$, the old contents in c are killed. At ℓ_9, $\ell_5 \xrightarrow{a} \ell_9$ becomes* *spurious since* $\langle \ell_9, a\rangle \hookleftarrow \langle \ell_5, a\rangle$ *is falsified. At* ℓ_{15}, $\ell_{14} \xrightarrow{b} \ell_{15}$ *and* $\ell_{14} \xrightarrow{d} \ell_{15}$ *are filtered out since* $\langle \ell_{15}, b\rangle \hookleftarrow \langle \ell_{14}, b\rangle$ *and* $\langle \ell_{15}, d\rangle \hookleftarrow \langle \ell_{14}, d\rangle$ *are falsified. Finally,* $\ell_{15} \xrightarrow{b} \ell_{16}$ *is ignored since* $t3$ *points to* d *only (*`[LOAD]`*).* □

SUPA improves performance by caching points-to results to reduce redundant traversal, with reuse happening in the marked boxes in Figure 5. For example, in Figure 5(c), $pt(\langle \ell_{13}, t3\rangle) = \{\widehat{d}\}$ computed in `[LOAD]` is reused in `[STORE]`.

Example 2. *Let us consider a weak update example in Figure 6 by computing* $\mathrm{pt}(\langle \ell_{11}, z\rangle)$ *on-demand. At the confluence point* ℓ_9, $p3$ *receives the points-to information from both* $p1$ *and* $p2$ *in its two branches:* $\langle \ell_9, p_3\rangle \hookleftarrow \widehat{a}$ *and* $\langle \ell_9, p_3\rangle \hookleftarrow \widehat{e}$. *Thus, a weak update is performed to the two locations* a *and* e *at* ℓ_{10}. *Let us focus on* $\widehat{a}$ *only. By applying* `[STORE]`, $\langle \ell_{10}, a\rangle \hookleftarrow \langle \ell_4, r\rangle \hookleftarrow \widehat{d}$. *By applying* `[SU/WU]`, $\langle \ell_{10}, a\rangle \hookleftarrow \langle \ell_6, a\rangle \hookleftarrow \langle \ell_3, y\rangle \hookleftarrow \widehat{c}$. *Thus,* $\mathrm{pt}(\langle \ell_{11}, a\rangle) = \{c, d\}$, *which excludes* b *due to a strong update performed at* ℓ_6. *As* $\mathrm{pt}(\langle \ell_7, q\rangle) = \{a\}$, *we obtain* $\mathrm{pt}(\langle \ell_{11}, z\rangle) = \{c, d\}$. □

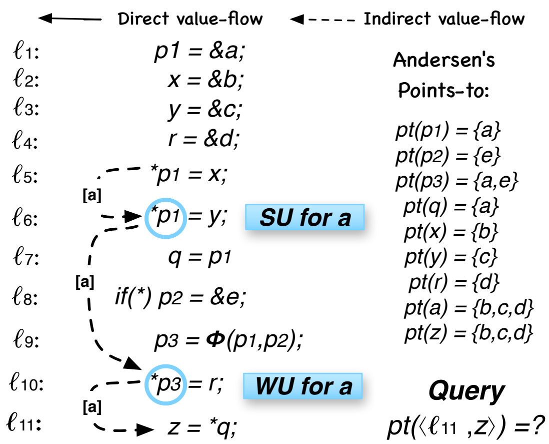


Figure 6: Resolving $pt(\langle \ell_{11}, z\rangle) = \{c, d\}$ **with a weak update**

Unlike [23], 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 given a sufficient budget. In Figure 6, 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, SUPA retraverses it whenever new points-to information is found until a fix point is reached.

Example 3. *Figure 7 shows a value-flow cycle formed by* $\ell_5 \xrightarrow{x} \ell_6$ *and* $\ell_6 \xrightarrow{z} \ell_5$. *To compute* $\mathrm{pt}(\langle \ell_6, z\rangle)$, *we must compute* $\mathrm{pt}(\langle \ell_5, x\rangle)$, *which requires the aliases of* $*z$ *at the load* $\ell_5 : x = *z$ *to be found by using* $\mathrm{pt}(\langle \ell_6, z\rangle)$. *SUPA computes* $\mathrm{pt}(\langle \ell_6, z\rangle)$ *by analyzing this value-flow cycle in two iterations. In the first iteration, a pointed-to target* $\widehat{b}$ *is found since* $\langle \ell_6, z\rangle \hookleftarrow \langle \ell_4, y\rangle \hookleftarrow \widehat{b}$. *Due to* $\langle \ell_2, q\rangle \hookleftarrow \widehat{b}$, $*z$ *and* $*q$

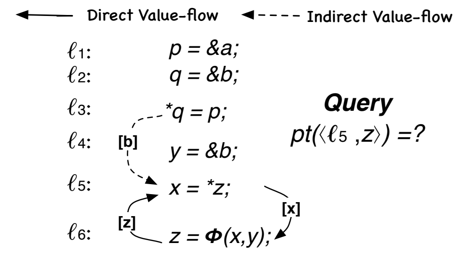


Figure 7: Resolving $pt(\langle \ell_5, z\rangle) = \{a, b\}$ **in a value-flow cycle**

are found to be aliases. In the second iteration, another target $\widehat{a}$ *is found since* $\langle \ell_6, z\rangle \hookleftarrow \langle \ell_5, x\rangle \hookleftarrow \langle \ell_3, b\rangle \hookleftarrow \langle \ell_1, p\rangle \hookleftarrow \widehat{a}$. *Thus,* $\mathrm{pt}(\langle \ell_6, z\rangle) = \{a, b\}$ *is obtained.* □

4.1.2 Call Graph Refinement

Unlike [15], which uses an imprecisely pre-computed call graph during its analysis, SUPA refines it on-the-fly. Let us consider how to resolve the points-to set of 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.

4.1.3 Properties

Theorem 1 (Soundness)**.** *SUPA is sound in analyzing a program as long as its pre-analysis is sound.*

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

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

Proof Sketch. Let $pt_{\text{SUPA}}(\langle \ell, v\rangle)$ and $pt_{\text{FS}}(\langle \ell, v\rangle)$ be the points-to sets computed by SUPA and FS, respectively. By Theorem 1, $pt_{\text{SUPA}}(\langle \ell, v\rangle) \supseteq pt_{\text{FS}}(\langle \ell, v\rangle)$, since SUPA is a demand-driven version of FS and thus cannot be more precise. To show that $pt_{\text{SUPA}}(\langle \ell, v\rangle) \subseteq pt_{\text{FS}}(\langle \ell, v\rangle)$, we note that SUPA operates on the SVFG of the program to improve its efficiency, by also filtering out value-flows imprecisely pre-computed by the pre-analysis. For the top-level variables, their direct value-flows are precise. So SUPA proceeds exactly the same as FS (`[ADDR]`, `[COPY]`, `[PHI]` 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]` 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, $pt_{\text{SUPA}}(\langle \ell, v\rangle) = pt_{\text{FS}}(\langle \ell, v\rangle)$. □

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

$$\text{[C-LOAD]}\ \frac{c, \ell : p = *q \quad \ell'' \xrightarrow{q} \ell \quad \langle c, \ell'', q\rangle \hookleftarrow (c', \widehat{o}) \quad \ell' \xrightarrow{o} \ell}{\langle c, \ell, p\rangle \hookleftarrow \langle c', \ell', o\rangle} \qquad \text{[C-STORE]}\ \frac{c, \ell : *p = q \quad \ell'' \xrightarrow{p} \ell \quad \langle c, \ell'', p\rangle \hookleftarrow (c', \widehat{o}) \quad \ell' \xrightarrow{q} \ell}{\langle c', \ell, o\rangle \hookleftarrow \langle c, \ell', q\rangle}$$

$$\text{[C-SU/WU]}\ \frac{c, \ell : *p = \_ \quad \ell' \xrightarrow{o} \ell \qquad (c', o) \in \mathcal{A} \setminus kill(c, \ell, p)}{\langle c', \ell, o\rangle \hookleftarrow \langle c', \ell', o\rangle} \qquad \text{[C-COMPO]}\ \frac{lv \hookleftarrow lv' \quad lv' \hookleftarrow lv''}{lv \hookleftarrow lv''}$$

$$\text{[C-CALL]}\ \frac{c, \ell_{cal_\kappa} : v = v' \quad c' = c \ominus \kappa \quad \ell' \xrightarrow{v'} \ell_{cal_\kappa}}{\langle c, \ell_{cal_\kappa}, v\rangle \hookleftarrow \langle c', \ell', v'\rangle} \qquad \text{[C-RET]}\ \frac{c, \ell_{ret_\kappa} : v = v' \quad c' = c \oplus \kappa \quad \ell' \xrightarrow{v'} \ell_{ret_\kappa}}{\langle c, \ell_{ret_\kappa}, v\rangle \hookleftarrow \langle c', \ell', v'\rangle}$$

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

Figure 8: Single-stage flow- and context-sensitive Supa analysis with demand-driven strong updates

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 [29, 40, 41, 44, 57]. A context stack c is encoded as a sequence of callsites, $[\kappa_1 \dots \kappa_m]$. $c \oplus \kappa$ denotes an operation for pushing a callsite κ into c. $c \ominus \kappa$ pops κ from c if c contains κ as its top value or is empty since a realizable path may start and end in different functions.

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

Given $\langle c, \ell, v\rangle$, Supa computes its points-to set context-sensitively by applying the rules given in Figure 8:

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

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

Passing parameters to and returning results from a callee invoked at a callsite κ are modeled by copies ($v = v'$) [15, 51, 60]. In [C-CALL], $v' \in \mathcal{V}$ denotes a variable passed into the callee directly or indirectly via parameter passing. Similarly, v' in [C-RET] represents a value returned directly or indirectly from the callee to its caller. Such def-use chains are built in the same way as others (Section 2.2), based on the points-to information obtained by pre-analysis.

With context-sensitivity, Supa will filter out more spurious value-flows, thereby producing more precise points-to information to enable more strong updates ([C-SU/WU]). At a store $c, \ell : *p = \_$, its kill set is context-sensitive. A strong update is applied if p points to a *context-sensitive singleton* $(c', o') \in cxtSingletons$, where o' is a (non-heap) singleton defined in Section 4.1 or a heap object with c' being a *concrete* context, i.e., one not involved in recursion or loops.

For a given 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 [44].

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 from the program. Let the N stages of Supa, ***Stage[0]***, $\cdots$, ***Stage[N-1]***, be configured with budgets $\eta_0, \cdots, \eta_{N-1}$, 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 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 5. Let $\mathcal{Q}$ be the set of queried variables issued from a program. Let $\mathcal{I} \supseteq \mathcal{Q}$ be the set of variables reached from $\mathcal{Q}$ during the analysis. Let $(\ell, v) \in \mathcal{Q}$ be a queried variable. We write $pt^i_{\eta_i}(\langle \Delta_i, \ell, v\rangle)$ 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, $pt^N_{\eta_N}(\langle \Delta_N, \ell, v\rangle)$ denotes the points-to set obtained by pre-analysis, at ***Stage[N]*** (conceptually).

When resolving $pt^i_{\eta_i}(\langle \Delta_i, \ell, v\rangle)$ 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 $*(\leqslant \eta_i)$ 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:

$\boxed{\textbf{Backward Reuse: } (\ell', v') \in \mathcal{I}}$ If $pt^j_*(\langle \Delta_j, \ell', v'\rangle)$, where $j \leqslant i$, 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.

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

Of course, many other schemes are possible with or without precision loss and will be investigated in future work.

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 (for the first time) 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 [63]) 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 always 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 used are field-sensitive. Each field instance of a struct is treated as a separate object. Analyzing a field operation, e.g., $x' = x$ GetElementPtr f in the LLVM IR is similar as handling a [COPY] statement. The only difference is that $pt(x')$ must include the field objects at the offset f of the pointed-to targets in $pt(x)$: $pt(x') = \{o.f | o \in pt(x)\}$. Arrays are considered monolithic. Positive weight cycles that arise from processing fields of struct objects are collapsed [36]. Distinct allocation sites (i.e., AddrOf statements) are modeled by distinct abstract objects as in [15].

We build the SVFG for a program based on our open-source software, SVF [49]. The def-use chains are pre-computed by Andersen's algorithm flow-insensitively.

To compare Supa with whole-program analysis, we have implemented a sparse flow-sensitive (SFS) analysis described in [15] also in LLVM, as SFS is a recent solution yielding exactly the flow-sensitive precision with good scalability. However, there are some differences. In [15], SFS was implemented in LLVM (2.5.0), by using imprecisely pre-computed call graphs and representing points-to sets with binary decision diagrams (BDDs). In this paper, just like Supa, SFS is implemented in LLVM (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 [2], 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 uninitialization at a stack or heap allocation site $\ell : p = \&a$ for an uninitialized pointer a, we add a special store $*p = u$ immediately after ℓ, where u points to an *unknown abstract object* (UAO), u_a. Given a load $x = *y$, we can issue a points-to query for x to detect its potential uninitialization. If x points to a u_a (for some a), then x may be uninitialized. By performing strong updates more often, a flow-sensitive analysis can find more UAO's that do not reach any pointer and thus prove more pointers to be initialized. Note that SFS can yield false positives since, for example, path correlations are not modeled.

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

We generate meaningful points-to queries, one query for the top-level variable x at each load $x = *y$. However, we ignore this query if x is found not to point to any UAO by pre-analysis. This happens only when x points to either default-initialized objects or unmodeled local variables in recursion cycles or arrays. The number of queries issued in each program is listed in the last column in Table 1.

Table 1: Program characteristics

Program	KLOC	Statements	Pointers	Alloc Sites	Queries
milc-v6	15	11713	29584	865	3
less-451	27.1	6766	22835	1135	100
hmmer-2.3	36	27924	74689	1472	2043
make-4.1	40.4	14926	36707	1563	1133
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
bash-4.3	155.9	59442	191413	6359	5103
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	1807.6	670761	2158377	49999	33232

5.3 Experimental Setup

We use a machine with a 3.7G Hz Intel Xeon 8-core CPU and 64 GB memory. As shown in Table 1, we have selected 12 open-source programs (including nine recently released applications) from a variety of areas: `milc-v6` (quantum chromodynamics), `less-451` (a terminal pager), `hmmer-2.3` (sequence similarity searching), `make-4.1`, `a2ps-4.14` (a postScript filter), `bison-3.04` (a parser), `grep-2.2.1`, `tar-1.28`, `bash-4.3`, `sendmail-8.15.1`, `vim74`, and `emacs-24.4`.

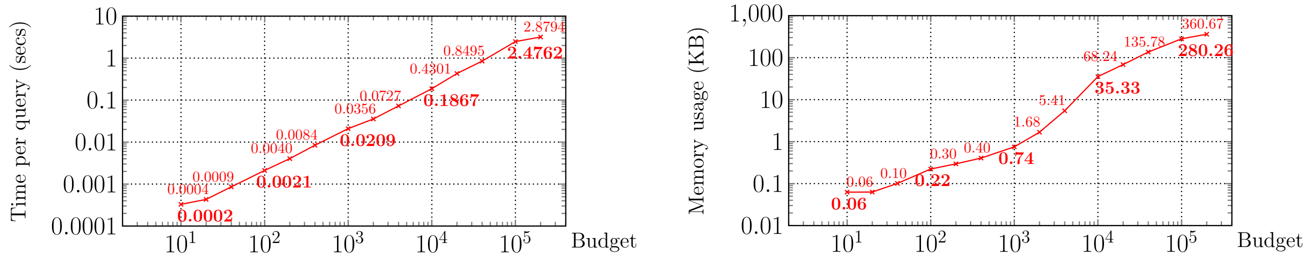
For each program, Table 1 lists its number of lines of code, statements, pointers, allocation sites (or `AddrOf` statements), and queries issued (as discussed in Section 5.2).

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 criteria: effi-



(a) Analysis Time (b) Memory Usage

Figure 9: Average analysis time and memory usage per query consumed by Supa-FS under different budgets

ciency (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 traversed def-use chains. 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 0.19 seconds and 36KB of memory per query, on average.

Table 2: 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	
milc	0.42	0.1	0.52	0.16
less	0.42	0.37	0.79	1.94
hmmer	1.57	0.46	2.03	1.07
make	1.74	1.17	2.91	13.94
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
bash	53.48	44.07	97.55	2590.69
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

Efficiency. Figure 9(a) shows the average analysis time per query for all the programs under a given budget, with about 0.19 seconds when $B = 10000$ and about 2.88 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 70% – 80% across the programs) take much less than the average cases. For emacs, 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 (Figure 10).

For SUPA, its pre-analysis is lightweight, as shown in Table 2. with `vim` taking the longest at 531.57 seconds. The same pre-analysis is shared by SFS to enable its sparse analysis. The additional time taken by SFS for analyzing each program entirely is given in the last column.

Figure 9(b) shows the average memory usage per query under different budgets. Following the common practice, we measure the real-time memory usage by reading the virtual memory information (`VmSize`) from the linux kernel file (`/proc/self/status`). The memory monitor starts after the pre-analysis to measure the memory usage for answer-

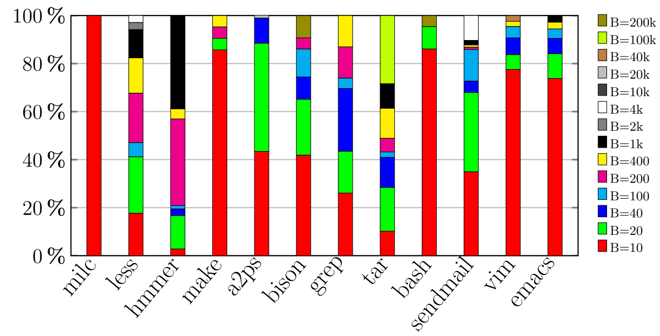


Figure 10: Percentage of queried variables proved to be initialized by Supa-FS over SFS under different budgets

ing queries only. The average amount of memory consumed per query is small, with about 36KB when $B = 10000$ and about 360KB when $B = 200000$. Even under the largest budget $B = 200000$ evaluated, SUPA-FS never uses more than 3MB for any single query processed.

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

Figure 10 reports our results. As B increases, the precision of SUPA-FS generally improves. With $B = 10000$, SUPA-FS can answer correctly 97% of all the queries from the 12 programs. These results indicate that our analysis is highly accurate, even under tight budgets. For the 12 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 11 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.

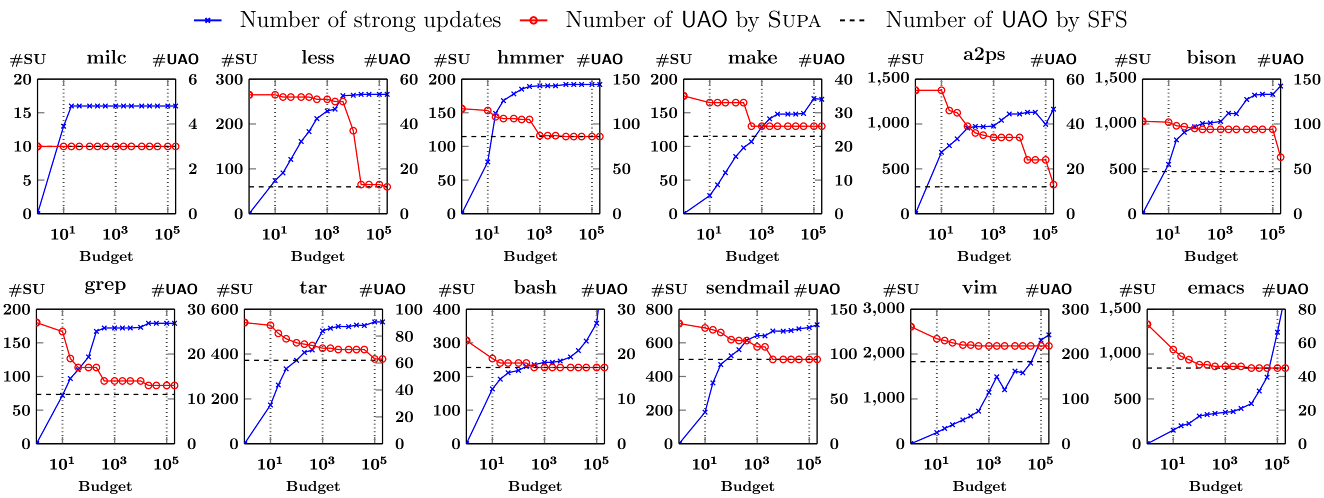


Figure 11: Correlating the number of strong updates with the number of UAO's under different budgets

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

When $B = 200000$, SUPA-FS gives the same answers as SFS in all the 12 programs except `bison` and `vim`, which causes SUPA-FS to report 16 and 35 more, respectively.

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

For programs like `bison`, `bash` and `emacs`, quite a few strong updates take place when $B > 1000$. There are two main reasons. First, these programs have many indirect callsites (with 293 in `bison`, 126 in `bash` and 446 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` 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. Unfortunately, whole-program analysis does not scale well to large programs when both are considered (Section 5.1).

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

In general, SUPA-FSCS has longer query response times than SUPA-FS due to the larger budgets used in our setting and the times taken in handling context-sensitivity. In `milc`, `hmmer`, `a2ps`, `bison`, `tar` and `sendmail`, SUPA-FSCS reports fewer UAO's than SUPA-FS, for two reasons. First, SUPA-FSCS can perform strong updates context-sensitively, 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 Sec-

Table 3: Average analysis times 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
milc	0.02	3	14.52	**0**
less	15.15	37	92.41	37
hmmer	11.41	86	135.05	**71**
make	124.40	26	229.44	26
a2ps	126.01	34	448.26	**32**
bison	465.54	94	529.20	**86**
grep	124.46	14	197.66	14
tar	26.31	70	83.10	**68**
bash	188.69	17	327.16	17
sendmail	200.32	94	250.19	**85**
vim	168.67	218	473.25	218
emacs	159.22	45	222.65	45

tion 4.2, by eliminating eight UAO's in `bison`, 1 in `tar` and 1 in `sendmail`, which have been reported by SUPA-FS.

6. 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.

6.1 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. Earlier, Choi et al. [6] and Emami et al. [10] gave some formulations in an iterative data-flow framework [18]. Wilson and Lam [55] 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 [14, 15, 20, 35, 60], 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 [7, 39] as in [17] and various SSA representations such as HSSA [8] and partial SSA [22]. 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 [14]. Recently, the idea of staged analysis [12, 15] that uses pre-computed points-to information to bootstrap a later more precise analysis has been leveraged to make pointer analysis full-sparse for both top-level and address-taken variables [15, 35, 48, 52, 59, 60]. This paper is the first to exploit sparsity to improve the performance of a demand-driven strong update analysis.

There are several parallel implementations of Andersen's flow-insensitive algorithm on multicore CPUs [31, 37], GPUs [30], and heterogeneous CPU-GPU systems [46], with no strong updates performed. However, a flow-sensitive parallel implementation of Andersen's algorithm that supports strong updates on multi-core CPUs also exists [34].

6.2 Demand-Driven Pointer Analysis

All the existing demand-driven pointer analyses for C [16, 64, 61] and Java [29, 41, 57, 44, 47] are flow-insensitive, formulated in terms of CFL (Context-Free-Language) reachability [40]. Heintze and Tardieu [16] introduced the first on-demand Andersen-style pointer analysis for C. Later, Zheng and Rugina [64] 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 [45] and later with context-sensitivity [45, 44]. Shang et al. [41] and Yan et al. [57] investigated how to summarize points-to information discovered during the CFL-reachability analysis to improve performance for Java programs. Lu et al. [29] introduced an incremental pointer analysis with a CFL-reachability formulation for Java. Su et al. [47] demonstrated that the CFL-reachability formulation is highly amenable to parallelisation on multi-core CPUs. Recently, Feng et al. [11] focused on answering demand queries for Java programs in a context-sensitive analysis framework (without strong updates). Unlike these flow-insensitive analyses, which are not effective for many clients like Uninit, SUPA can perform strong updates on-demand flow and context-sensitively.

`Boomerang` [43], a very recent IFDS-based flow- and context-sensitive pointer analysis for Java, also demonstrates the importance of demand-driven pointer analysis for security clients, such as FlowDroid [4].

6.3 Hybrid Pointer Analysis

The basic idea is to find a right balance between efficiency and precision. For C programs, the one-level approach [9] 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 [19]. In [13], how to adjust the analysis precision according to a client's needs is discussed. Zhang et al. [62] focus on finding effective abstractions for whole-program analyses written in Datalog via abstraction refinement. Lhoták and Chung [23] 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 [23], SUPA is capable of achieving the same precision as whole-program flow-sensitive analysis, subject to a given budget.

7. CONCLUSION

We have introduced, SUPA, an on-demand strong update analysis that enables computing precise points-to information for C programs flow- and context-sensitively 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 12 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.

8. ACKNOWLEDGEMENTS

We thank all the reviewers for their constructive comments on an earlier version of this paper. This research has been supported by ARC grants, DP130101970 and DP150101970.

9. ARTIFACT DESCRIPTION

9.1 Artifact Package

You can find this package, together with instructions, on how to use SUPA at http://www.cse.unsw.edu.au/~corg/supa.

A brief checklist:.

- `index.html`: the detailed instructions for reproducing the experimental results in the paper.
- `SUPA.ova`: a virtual image file (~5GB) containing installed Ubuntu OS and SUPA project (http://corg-pluto.cse.unsw.edu.au/supa/SUPA.ova).
- SUPA implementation developed on top of the SVF framework: http://unsw-corg.github.io/SVF.
- Scripts for reproducing all the data in the paper, including:
  - `./run.sh`,
  - `./figure_9.sh`.
  - `./figure_10.sh`,
  - `./figure_11.sh`,
  - `./table_2.sh`,
  - `./table_3.sh`.
- Micro-benchmarks to validate pointer analysis results.

Platform:.

All the results related to analysis times and memory usage in our paper are obtained on a 3.7G Hz Intel Xeon 8-core CPU with 64 GB memory. The OS in the virtual machine image is Ubuntu 12.04. A user account has been created with both its username and password as "pta".

To run SUPA, you are advised to allocate at least 16GB memory to the virtual machine. The whole-program sparse flow analysis, denoted SFS in the paper, requires more memory, as a lower memory budget may force OS to kill the running process when it is used to analyze some large programs, e.g., `vim`, `gdb` and `emacs`. Finally, a VirtualBox with version 4.1.12 or newer is required to run the image.

License:.

GPLv3 (www.gnu.org/licenses/gpl-3.0.en.html)

9.2 Quick Guidelines

To run the experiments as we did for in our paper, open a terminal and do:

- **cd /home/pta/pta/** # Go to the SUPA project directory, denoted as $SUPAHOME
- **. ./setup.sh** # Set up environment variables (please note that there is a white space between the two dots)
- **cd SUPA-exp** # Go to the experiment directory

To run the three analyses for all 12 benchmarks, execute the following scripts:

- **./run.sh DFS** # Run SUPA-FS
- **./run.sh CXT** # Run SUPA-FSCS
- **./run.sh SFS** # Run whole-program SFS

On our platform, obtaining the results for SUPA may take about 30 mins with a budget of 1000, and obtaining the results for SFS may take more than five hours (especially for large programs, such as `bash`, `vim` and `emacs`).

Initially, the users are advised to analyze a few benchmarks with small budgets using the default configuration files 'budget' and 'benchmarks'. To analyze all the benchmarks, please use another configuration file containing all the benchmarks (and remember to re-run everything if the configuration files have been changed):

After all the analyses are complete, you can collect the data included in our tables and figures using the following scripts in the same folder:

- **./figure_9.sh** # Data in Figure 9
- **./figure_10.sh** # Data in Figure 10
- **./figure_11.sh** # Data in Figure 11
- **./table_2.sh** # Data in Table 2
- **./table_3.sh** # Data in Table 3

For comparison purposes, we have also provided the experimental data presented in the paper under "$SUPAHOME/SUPA-exp/supa-fse/".

To reproduce the results shown in tables and figures with the provided data, issue the following commands:

- **./figure_9.sh supa-fse** # Data in Figure 9
- **./figure_10.sh supa-fse** # Data in Figure 10
- **./figure_11.sh supa-fse** # Data in Figure 11
- **./table_2.sh supa-fse** # Data in Table 2
- **./table_3.sh supa-fse** # Data in Table 3

9.3 More Experiments and Developer Guide

Please refer to

- The website of SUPA (http://www.cse.unsw.edu.au/~corg/supa)
- The Wiki site of our SVF framework (http://unsw-corg.github.io/SVF).

10. REFERENCES

[1] SUPA. http://www.cse.unsw.edu.au/~corg/supa.

[2] M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE '11*, pages 746–755, 2011.

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

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

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

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

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

[8] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267, 1996.

[9] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI '00*, pages 35–46, 2000.

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

[11] Y. Feng, X. Wang, I. Dillig, and C. Lin. EXPLORER: query- and demand-driven exploration of interprocedural control flow properties. In *OOPSLA '15*, pages 520–534, 2015.

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

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

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

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

[16] N. Heintze and O. Tardieu. Demand-driven pointer analysis. In *PLDI '01*, pages 24–34, 2001.

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

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

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

[20] U. P. Khedker, A. Mycroft, and P. S. Rawat. Liveness-based pointer analysis. In *SAS '12*, pages 265–282. 2012.

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

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

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

[24] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. CC '03, pages 153 – 169.

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

[26] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP '14*, pages 27–53.

[27] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *SAS '15*, pages 162–180.

[28] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program Tailoring: Slicing by Sequential Criteria. In *ECOOP '16*, pages 15:1–15:27, 2016.

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

[30] M. Méndez-Lojo, M. Burtscher, and K. Pingali. A GPU implementation of inclusion-based points-to analysis. In *PPoPP '12*, pages 107–116, 2012.

[31] M. Méndez-Lojo, A. Mathew, and K. Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA '10*, pages 428–443, 2010.

[32] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. ISSTA '02.

[33] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[34] V. Nagaraj and R. Govindarajan. Parallel flow-sensitive pointer analysis by graph-rewriting. In *PACT '13*, pages 19–28, 2013.

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

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

[37] S. Putta and R. Nasre. Parallel replication-based points-to analysis. In *CC '12*, pages 61–80, 2012.

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

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

[40] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

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

[42] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL'11*, pages 17–30, 2011.

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

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

[45] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. In *OOPSLA '05*, pages 59–76, 2005.

[46] Y. Su, D. Ye, and J. Xue. Accelerating inclusion-based pointer analysis on heterogeneous CPU-GPU systems. In *HiPC '13*, pages 149–158, 2013.

[47] Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with cfl-reachability. In *ICPP '14*, pages 451–460, Sept 2014.

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

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

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

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

[52] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171, 2011.

[53] Q. Sun, J. Zhao, and Y. Chen. Probabilistic points-to analysis for java. In *CC '11*, pages 62–81, 2011.

[54] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *SAS '16*. 2016.

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

[56] X. Xiao and C. Zhang. Geometric encoding: forging the high performance context sensitive points-to analysis for Java. In *ISSTA '11*, pages 188–198, 2011.

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

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

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

[60] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229, 2010.

[61] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for C. In *PLDI '14*, pages 829–845, 2014.

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

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

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