The Flow-Insensitive Precision of Andersen's Analysis in Practice

Sam Blackshear[1], Bor-Yuh Evan Chang[1], Sriram Sankaranarayanan[1], and Manu Sridharan[2]

[1] University of Colorado Boulder
{samuel.blackshear,evan.chang,sriram.sankaranarayanan}@colorado.edu
[2] IBM T.J. Watson Research Center
msridhar@us.ibm.com

Abstract. We present techniques for determining the precision gap between Andersen's points-to analysis and precise flow-insensitive points-to analysis in practice. While previous work has shown that such a gap may exist, no efficient algorithm for precise flow-insensitive analysis is known, making measurement of the gap on real-world programs difficult. We give an algorithm for precise flow-insensitive analysis of programs with finite memory, based on a novel technique for refining any points-to analysis with a search for flow-insensitive witnesses. We give a compact symbolic encoding of the technique that enables computing the search using a tuned SAT solver. We also present extensions of the algorithm that enable computing lower and upper bounds on the precision gap in the presence of dynamic memory allocation. In our experimental evaluation over a suite of small- to medium-sized C programs, we never observed a precision gap between Andersen's analysis and the precise analysis. In other words, Andersen's analysis computed a precise flow-insensitive result for all of our benchmarks. Hence, we conclude that while better algorithms for the precise flow-insensitive analysis are still of theoretical interest, their practical impact for C programs is likely to be negligible.

1 Introduction

Programming languages such as C and Java make extensive use of pointers. As a result, many program analysis questions over these languages require pointer analysis as a primitive to find the set of all memory locations that a given pointer may address. This problem is of fundamental importance and has been widely studied using numerous approaches [8]. Recently, Andersen's analysis [1] has been increasingly employed to analyze large programs [7, 19]. However, it is also well known that Andersen's analysis falls short of being a *precise* flow-insensitive analysis [5, 9, 17]. A precise flow-insensitive analysis reports only the points-to relationships that are realizable via executing some sequence of program statements, assuming arbitrary control flow between statements. There are two key reasons for the precision gap between Andersen's analysis and a precise flow-insensitive analysis (discussed further in Sect. 2):

SAS 2011.

- Andersen's analysis assumes that any set of points-to edges can occur *simultaneously*, whereas program variables must point to a single location at any program state. This discrepancy may cause Andersen's to generate spurious points-to edges.
- Andersen's analysis transforms pointer assignments to contain at most one dereference by rewriting complex statements using fresh temporary variables. However, temporary variables can introduce spurious points-to edges [9].

These observations lead to two tantalizing and long-standing questions:

1. Is there an *efficient* algorithm for precise flow-insensitive pointer analysis?
2. Does a precision gap exist, *in practice*, for real-world programs?

Regarding the first question, precise flow-insensitive analysis is NP-hard for arbitrary finite-memory programs [9], and no polynomial-time algorithm is known even for programs with only Andersen-style statements [5]. In the presence of dynamic memory, the decidability of the problem remains unknown [5].

This paper addresses the second question by presenting techniques for computing the precision gap between Andersen's and precise flow-insensitive points-to analysis in practice. We introduce an algorithm for computing the precise flow-insensitive analysis for programs with finite memory. This algorithm refines Andersen's analysis results by searching for an appropriate sequence of statements to *witness* each edge in the points-to graph obtained from Andersen's analysis. The search is encoded symbolically and carried out using efficient modern SAT solvers. Although the worst-case performance of our algorithm is exponential, our SAT encoding enables analysis of medium-sized C programs within reasonable time/memory bounds. We then extend our techniques to investigate the precision gap in the presence of dynamic memory.

We performed an experimental evaluation to measure the precision gap between Andersen's and precise flow-insensitive analysis on a suite of C programs. Perhaps surprisingly, we found the results of the two analyses to be identical over our benchmarks: *a precision gap seems to be non-existent, in practice*. Thus, we conclude that better algorithms for precise flow-insensitive points-to analysis, while retaining theoretical interest, are unlikely to have a large impact on the analysis of C programs. Instead, our conclusions suggest efforts spent on refining Andersen's analysis with flow or context sensitivity may be more fruitful. Interestingly, our witness search algorithm may offer a basis for such efforts.

This paper makes the following contributions:

- We present an algorithm for precise flow-insensitive analysis for programs with finite memory based on refining Andersen's analysis with a *witness search* for each computed points-to fact (Sect. 3.1).
- We describe extensions for handling dynamic memory over- and under-approximately in order to evaluate the precision gap resulting from the lack of a fully precise treatment of dynamic memory (Sect. 3.2).
- We also give a compact symbolic encoding of the witness search algorithm, enabling the use of highly-tuned SAT solvers for the search (Sect. 3.3).

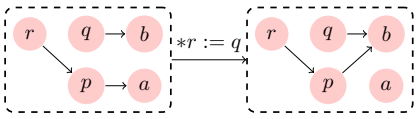
– We implemented our algorithms and performed an experimental evaluation, showing that the precision gap seems non-existent for small- to medium-sized C programs (Sect. 4).

2 Flow-Insensitive Imprecision in Andersen's Analysis

In this section, we examine the sources of imprecision in Andersen's analysis compared to a precise flow-insensitive points-to analysis. Most of this discussion is a reformulation of known concepts.

We first define the notion of a precise flow-insensitive points-to analysis. A (flow-insensitive) points-to analysis problem consists of a finite set of variables X along with a set of assignments A. The simplest variant considers only *finite memory*, that is, each assignment has one of the following forms: $*^d p := \&q$ or $*^{d_1} p := *^{d_2} q$ where p and q are variables. The expression $*^d p$ denotes the application of $d \geq 0$ dereferences to pointer p, while $\&q$ takes the address of q. Note that $*^0 p$ is the same as p. The *dynamic memory* points-to analysis problem adds a statement $*^d p := \texttt{malloc}()$ for allocation. The goal of a precise flow-insensitive points-to analysis is to answer queries of the form $p \mapsto q$: is there a sequence of assignments from A that causes p to point to q (i.e., that causes variable p to contain the address of q)? The problem is flow-insensitive, as program control flow is ignored to produce a set of assignments as input.

The result of a points-to analysis can be captured as a *points-to graph*. A points-to graph $G\colon (V, E)$ consists of a set of vertices V and directed edges E. The set of vertices represents memory cells and thus includes the program variables (i.e., $V \supseteq X$). To conservatively model constructs like aggregates (e.g., arrays or structures), dynamically allocated memory, and local variables in a recursive context, a vertex may model more than one concrete memory cell (which is referred to as a *summary location*). An edge $v_1 \mapsto v_2$ says that v_1 may point to v_2 (i.e., a concrete cell represented by v_1 may contain an address from v_2) under some execution of assignments drawn from A. For convenience, we use the notation $V(G)$ or $E(G)$ to indicate the vertices and edges of G.



An exact abstraction of a concrete memory configuration can be modeled by a points-to graph where each vertex represents a single memory cell and thus each vertex can have at most one outgoing points-to edge. We call such graphs *exact points-to graphs*. A points-to graph obtained as a result of some may points-to analysis may be viewed as the join of some number of exact points-to graphs. With exact point-to graphs, we can define the operational semantics of pointer assignments from a points-to analysis problem. We write $G \xrightarrow{a} G'$ for the one-step transition relation that says assignment a transforms exact graph G to exact graph G'. A formal definition is provided in our companion technical report [3]. The inset figure illustrates the transformation of an exact points-to graph through an assignment. We can now define *realizability* of points-to edges.

Definition 1 (Realizable Graphs, Edges, and Subgraphs). *A graph G is* realizable *iff there exists a sequence of assignments $a_1, \ldots, a_N$ such that $G_0 \xrightarrow{a_1} G_1 \rightarrow \cdots \xrightarrow{a_N} G_N \equiv G$ where $G_0 \colon (X, \emptyset)$ is the* initial graph *of the points-to-analysis problem with variables X. An edge $v_1 \mapsto v_2 \in V \times V$ is* realizable *iff there exists a realizable graph G such that $v_1 \mapsto v_2 \in E(G)$. A subset of edges $E \subseteq V \times V$ is (simultaneously) realizable if there exists a realizable graph G such that $E \subseteq E(G)$.*

A *precise flow-insensitive points-to analysis* derives all edges that are realizable and no other edges.

Andersen's analysis [1], well studied in the literature, is an over-approximate flow-insensitive points-to analysis computable in polynomial time. In essence, Andersen's analysis works by deriving a graph with all points-to relations using the inference rules shown below:

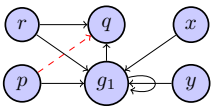
$$\frac{p := \&q}{p \mapsto q} \qquad \frac{p := q \quad q \mapsto r}{p \mapsto r} \qquad \frac{p := *q \quad q \mapsto r \quad r \mapsto s}{p \mapsto s} \qquad \frac{*p := q \quad p \mapsto r \quad q \mapsto s}{r \mapsto s}$$

where an assignment a (e.g., $p := \&q$) in the rule states that a is in the set of program assignments A and a points-to edge e (e.g., $p \mapsto q$) states that e is in the derived points-to graph G (i.e., $e \in E(G)$). Observe that Andersen's analysis requires that an input problem be transformed so that all statements contain at most one dereference. This transformation itself may introduce imprecision, as we shall discuss shortly. Finally, Andersen's analysis handles dynamic memory over-approximately by essentially translating each statement $p := \texttt{malloc}()$ into $p := \&m_i$, where m_i is a fresh *summary location* representing all memory allocated by the statement.

Imprecision: Simultaneous Points-To. Previous work has pointed out that Andersen's is not a precise flow-insensitive points-to analysis [5, 9]. One source of imprecision in Andersen's analysis is a lack of reasoning about what points-to relationships can hold *simultaneously* in possible statement sequences.

Example 1. Consider the following set of pointer assignments:

$$\{p := *r,\ r := \&q,\ r := *x,\ x := \&g_1,\ y := x,\ *x := r,\ *x := y\}\,.$$

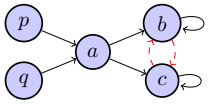


The inset figure shows the Andersen's analysis result for this example (for clarity, graphs with outlined blue nodes are used for analysis results). Notice that while $r \mapsto g_1$ and $g_1 \mapsto q$ are individually realizable, they cannot be realized simultaneously in any statement sequence, as this would require either: (1) pointer r to point to g_1 and q simultaneously; or (2) pointer g_1 to point to g_1 and q simultaneously (further illustration in Sect. 3.1). Andersen's does not consider simultaneous realizability, so with given the statement $p := *r$ and the aforementioned points-to edges, the analysis concludes that p may point to q (shown dashed in red), when in fact this edge is not realizable. The finite heap abstraction employed by Andersen's analysis may lead to conflation of multiple heap pointers, possibly worsening the simultaneous realizability issue.

Imprecision: Program Transformation. Imprecision may also be introduced due to the requisite decomposition of statements with multiple dereferences:

Example 2. Consider the following set of pointer assignments: $\{a := \&b,\ a := \&c,\ p := \&a,\ q := \&a,\ **p := *q\}$. The statement $**p := *q$ may make either b or c point to itself, but in no statement sequence can it make b point to c (as shown in the inset below). However, when decomposed for Andersen's analysis, the statement is transformed into statements introducing fresh variables t_1 and t_2: $t_1 := *p,\ t_2 := *q,\ *t_1 := t_2$. Then, the following sequence causes $b \mapsto c$:

$$a := \&b;\ p := \&a;\ t_1 := *p;\ a := \&c;\ q := \&a;\ t_2 := *q;\ *t_1 := t_2;$$



Hence, the transformation to Andersen's-style statements may create additional realizable points-to relationships among the original variables (i.e., the transformation adds imprecision even for precise flow-insensitive analysis). The goal of this work is to determine whether simultaneous realizability or program transformation issues cause a precision gap, in practice.

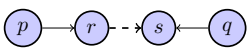
3 Precise Analysis via Witness Search

In this section, we present a witness search algorithm that yields a precise flow-insensitive points-to analysis for the finite-memory problem (Sect. 3.1). Then, we discuss two extensions to the algorithm that respectively provide over- and under-approximate handling of dynamic memory allocation and other summarized locations (Sect. 3.2). Finally, we describe a SAT-encoding of the search algorithm that yields a reasonably efficient implementation in practice (Sect. 3.3).

3.1 A Precise Algorithm for Finite Memory

Here, we describe our witness search algorithm, which computes a precise flow-insensitive analysis for programs with finite memory. Given the result of a conservative flow-insensitive points-to analysis, such as Andersen's [1], we first create *edge dependency rules* that capture ways a points-to edge may arise. These edge dependency rules are effectively instantiations of the Andersen inference rules. Next, we search for witness sequences for a given edge, on demand, using the edge dependency rules while taking into account constraints on simultaneous realizability. We may find no witness for an edge, in which case we have a *refutation* for the realizability of that points-to fact. Essentially, the dependency rules leverage the Andersen result to constrain a goal-directed search for realizability.

Generating Edge Dependency Rules. We first illustrate edge dependency rule construction through an example. Let G be a points-to graph derived as the result of a conservative points-to analysis. Consider the assignment $a\colon *p := q$, wherein edges $p \mapsto r$, $q \mapsto s$, and $r \mapsto s$ exist in G (as illustrated inset). In terms of realizability, the following claim can be made in this situation:



Edge $r \mapsto s$ is realizable (using assignment a) if the edge set $\{p \mapsto r,\ q \mapsto s\}$ is simultaneously realizable.

Note that the converse of this statement need not be true—the edge $r \mapsto s$ may be realizable using another set of edges and/or a different pointer assignment. In our framework, this assertion is represented by a *dependency rule*:

$$r \mapsto s \xleftarrow{a:\ *p:=q} \{p \mapsto r, q \mapsto s\}$$

This dependency rule indicates that the edge $r \mapsto s$ can be produced as a result of the assignment a whenever the edges $p \mapsto r$ and $q \mapsto s$ can be realized simultaneously.

The dependency rules can be created by examining a points-to graph G that results from a conservative analysis. Let us first consider assignments of the form $*^m p := *^n q$. For each such assignment, we generate a set of rules as follows:

- Let $\text{paths}(p, m)$ denote the set of all paths of length m starting from p in G, and let $\text{paths}(q, n+1)$ be the set of all paths of length $n+1$ starting from q.
- Consider each pair of paths $\pi_1 \colon p \rightsquigarrow_m p' \in \text{paths}(p, m)$ and $\pi_2 \colon q \rightsquigarrow_{n+1} q' \in \text{paths}(q, n+1)$.
- We generate the dependency rule: $\left(p' \mapsto q' \xleftarrow{*^m p:=*^n q} E(\pi_1) \cup E(\pi_2)\right)$ where $E(\pi_i)$ denotes the edges in the path π_i for $i \in \{1, 2\}$.

The case for assignments of the form $*^m p := \&q$ is essentially the same, so we elide it here. Overall, we obtain the set of rules for a finite-memory problem by taking all such rules generated from all assignments $a \in A$.

Note that the time taken for rule generation and the number of rules generated can be shown to be a polynomial in the size of the problem and the number of edges in the points-to graph (which is in turn at most quadratic in the number of variables) [3]. The time taken is exponential in the number of dereferences in the pointer assignments, but usually this number is very small in practice (it is at most one for Andersen-style statements).

This rule generation can be done offline as described above to take advantage of an optimized, off-the-shelf points-to analysis, but it can also be performed online during the execution of Andersen's analysis. Consider a points-to edge e discovered in the course of Andersen's analysis while processing an assignment a. The edges traversed at this step to produce e are exactly the dependence edges needed to create an edge dependency rule (as in the rule construction algorithm described above).

Example 3. Figure 1 shows the edge dependency rules derived from the result of Andersen's Analysis for the problem in Example 1.

Witness Enumeration. Once edge dependency rules are generated, witness search is performed via witness *enumeration*, which constructs possible partial witnesses. Consider a rule $r \colon e \xleftarrow{a} E$. Rule r states that we can realize edge e

$$r \mapsto q \xleftarrow{r:=\&q} \emptyset \qquad x \mapsto g_1 \xleftarrow{x:=\&g_1} \emptyset \qquad y \mapsto g_1 \xleftarrow{y:=x} x \mapsto g_1$$

$$g_1 \mapsto q \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto q \qquad g_1 \mapsto g_1 \xleftarrow{*x:=r} x \mapsto g_1, r \mapsto g_1$$

$$g_1 \mapsto g_1 \xleftarrow{*x:=y} x \mapsto g_1, y \mapsto g_1 \qquad r \mapsto g_1 \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto g_1$$

$$r \mapsto q \xleftarrow{r:=*x} x \mapsto g_1, g_1 \mapsto q \qquad p \mapsto g_1 \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto g_1 \qquad p \mapsto q \xleftarrow{p:=*r} r \mapsto g_1, g_1 \mapsto q$$

Fig. 1. The edge dependency rules for the problem in Example 1.

via assignment a if we can realize the set of edges E simultaneously (i.e., in a state satisfying E, executing a creates the points-to edge e). Intuitively, we can realize the set E if we can find a chain of rules realizing each edge in E. Thus, enumeration proceeds by repeatedly rewriting edge sets based on dependency rules until reaching the empty set; the statements associated with the rules employed become the candidate witness (see [3] for a detailed definition).

Example 4. We describe a witness enumeration step for Example 1. Starting from the set $E\colon \{r \mapsto g_1, g_1 \mapsto g_1\}$ and using the rule $r\colon g_1 \mapsto g_1 \xleftarrow{*x:=y} \{x \mapsto g_1, y \mapsto g_1\}$, we can rewrite set E to a set E' as follows:

$$E\colon \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \xrightarrow{r} \quad E'\colon \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\}\,.$$

Often, we will write such transitions using the same format as the rule itself:

$$E\colon \{r \mapsto g_1, g_1 \mapsto g_1\} \quad \xleftarrow{*x:=y} \quad E'\colon \{x \mapsto g_1, y \mapsto g_1, r \mapsto g_1\}\,.$$

Not all rewriting steps lead to valid witnesses. In essence, we need to ensure that the witness search respects the concrete semantics of the statements. Recall the definition of realizability (Definition 1), which states that a set of edges E is realizable if it is a subset of edges in a realizable graph. A realizable graph must be an exact points-to graph. Therefore, we simply detect when the exactness constraint is violated, which we call a *conflict set*.

Definition 2 (Conflict Set). *A set of edges E is a* conflict set *iff there exist two or more outgoing edges $v \mapsto v_1, v \mapsto v_2 \in E$ for some vertex v.*

In addition to conflict detection, we guarantee termination in the finite-memory problem by stopping cyclic rewriting of edge sets. Intuitively, if we have $E_1 \xrightarrow{r_1} E_2 \xrightarrow{r_2} \ldots \xrightarrow{r_n} E_n$, wherein $E_n \supseteq E_1$, the corresponding statements have simply restored the points-to edges in E_1. Hence no progress has been made toward a complete witness. Since all cyclic rewriting is truncated, and we have a finite number of possible edge sets (since memory is finite), termination follows.

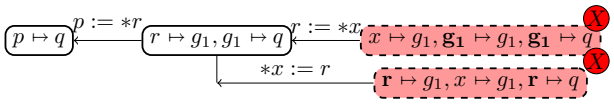
Performing witness enumeration with conflict set detection for each points-to fact derived by an initial analysis yields a precise flow-insensitive points-to analysis as captured by the theorem below. Proofs of all theorems are given in the appendix of our companion technical report [3].

Theorem 1 (Realizability). *(A) An edge e is realizable iff there exists a sequence of rewrites w: $E_0 : \{e\} \xrightarrow{r_1} E_1 \xrightarrow{r_2} \cdots \xrightarrow{r_N} E_N : \emptyset$, such that none of the sets $E_0, \ldots, E_N$ are conflicting. (B) Furthermore, it is also possible to find w such that $E_i \not\supseteq E_j$ for all $i > j$.*

Example 5. Considering the problem from Example 1, the following sequence of rule applications demonstrates the realizability of the edge $r \mapsto g_1$:

$$\{r \mapsto g_1\} \xleftarrow{r:=*x} \{x \mapsto g_1, g_1 \mapsto g_1\} \xleftarrow{*x:=y} \{x \mapsto g_1, y \mapsto g_1\}$$
$$\xleftarrow{y:=x} \{x \mapsto g_1\} \xleftarrow{x:=\&g_1} \emptyset\,.$$

The sequence of assignments corresponding to the set of rule applications provides the witness sequence: $x := \&g_1; y := x; *x := y; r := *x;$.



The converse of Theorem 1 can be applied to show that a given edge is not realizable. To do so, we search over the sequence of applicable rules, stopping our search when a conflicting set or a superset of a previously encountered set of edges is encountered. A refutation tree for the non-realizability of edge $p \mapsto q$ from Example 1 is shown inset. In one path, the search terminates with a conflict on g_1, and in the other, the conflict is on r.

Possible Extensions. Looking beyond precise flow-insensitive points-to analysis, our algorithm can be extended to provide greater precision by introducing additional validation of the produced witnesses. For example, context sensitivity could be added by ensuring that each witness respects call-return semantics. One could add flow or even path sensitivity in a similar manner. This additional checking could be performed on partial witnesses during the search, possibly improving performance by reducing the size of the search space. Further study of these extensions is promising future work.

3.2 Handling Summarized Locations

In practice, problems arising from programming languages such as C will contain complications such as union types, structure types handled field insensitively, local variables in a recursive function, thread local variables, and dynamic memory allocations. Such constructs are often handled conservatively through *summary locations*, which model a (possibly unbounded) collection of concrete memory locations. As noted in Sect. 2, to conservatively model the potentially unbounded number of allocated cells with dynamic memory, Andersen's analysis uses one summary location per allocation site in the program.

The decidability of the precise flow-insensitive analysis in the presence of dynamic memory is unknown [5]. Here, we present two extensions to our algorithm that respectively handle summary locations in an over- and under-approximate

manner, thereby yielding lower and upper bounds on the precision gap with a fully precise treatment of dynamic memory and other summarized locations.

Over-Approximating Summaries. To handle summary variables over-approximately, we can simply augment the search algorithm with weak update semantics for summaries. In particular, on application of a rule $r\colon e \xleftarrow{a} E$, if the source of edge e is a summary location, then e is not replaced in the rewriting (i.e., $E_0 \xrightarrow{r} E_0 \cup E$ for initial edge set E_0). Additionally, the definition of a conflict set (Definition 2) is modified to exclude the case when the conflict is on a summary location (i.e., two edges $v \mapsto v_1$ and $v \mapsto v_2$ where v is a summary location), as a summary may abstract an unbounded number of concrete cells. This handling clearly yields an over-approximate handling of summaries, as it is possible for the algorithm to generate witnesses that are not realizable by Definition 1. Hence, comparing Andersen's analysis and this algorithm yields a lower bound on the precision gap with a fully precise analysis.

Under-Approximating Summaries. To obtain an upper bound on the precision gap between Andersen's and the fully precise analysis, we define a straightforward under-approximating algorithm—during witness search, we treat summaries as if they were concrete memory locations. In essence, this approximation looks for witnesses that require only one instance of a summary (e.g., only one cell from a dynamic memory allocation site). This algorithm is unsound, as a points-to relation may be realizable even when this algorithm does not find a witness. However, if this algorithm finds a witness for a points-to relation, that relation is indeed realizable, and thus this algorithm yields an upper bound on the precision gap.

3.3 A Symbolic Encoding

In this section, we discuss a symbolic encoding for witness search and proving unrealizability. The idea is to encode the search for witnesses whose depths are bounded by some constant k using a Boolean formula $\varphi(e, k)$ such that any solution leads to a witness for edge e. We then adapt this search to infer the absence of witnesses by encoding subsumption checks. Crucially, our encoding allows *parallel updates* of unrelated pointer edges during witness search so that longer witnesses can be found at much smaller depths.

Propositions. For each edge $e \in E$ and depth $i \in [1, k+1]$, the Boolean variable $\mathsf{Edg}(e, i)$ denotes the presence of edge e in the set obtained at depth i. Similarly, for depths $i \in [1, k]$, the Boolean variable $\mathsf{Rl}(r, i)$ will be used to denote the application of the rule r at depth i (to obtain the set at depth $i+1$). Note that there is no rule application at the last step.

Boolean Encoding. Some of the key assertions involved in the Boolean encoding are summarized in Table 1. The assertion $\mathsf{init}(e)$ describes the edge set at depth 1, which is required to be the singleton $\{e\}$. Recall that a pair of edges conflict if they have the same source location (which is not a summary location). The assertion $\mathsf{edgeConflict}(e_A, e_B, i)$ is used for such conflicting edges. Similarly, we

Table 1. Overview of the boolean encoding for witness search.

Name	Definition	Remarks
$\mathsf{init}(e)$	$\mathsf{Edg}(e,1) \wedge \bigwedge_{e' \neq e} \neg\mathsf{Edg}(e',1)$	Start from edge set $\{e\}$
$\mathsf{edgeConflict}(e_A, e_B, i)$	$\neg\mathsf{Edg}(e_A,i) \vee \neg\mathsf{Edg}(e_B,i)$	Edges e_A, e_B cannot both be edges at depth i
$\mathsf{ruleConflict}(r_1, r_2, i)$	$\neg\mathsf{Rl}(r_1,i) \vee \neg\mathsf{Rl}(r_2,i)$	Rules r_1, r_2 cannot both be simultaneously applied at depth i
$\mathsf{someRule}(i)$	$\bigvee_{r \in R} \mathsf{Rl}(r,i)$	Some rule applies at depth i
$\mathsf{ruleApplicability}(r,i)$	$\mathsf{Rl}(r,i) \Rightarrow \mathsf{Edg}(e,i)$	Applying rule $r\colon e \leftarrow E$ at depth i creates edge e
$\mathsf{notSubsumes}(i,j)$	$\neg(\bigwedge_{e \in E} \mathsf{Edg}(e,i) \Rightarrow \mathsf{Edg}(e,j))$	Edge set at depth i does not contain set at depth j

define a notion of a conflict on the rules that enables parallel application of non-conflicting rules. Rules $r_1\colon e_1 \xleftarrow{a_1} E_1$ and $r_2\colon e_2 \xleftarrow{a_2} E_2$ are *conflicting* iff one of the following conditions holds: (a) $e_1 = e_2$, or (b) e_1 conflicts with some edge in E_2, or (c) e_2 conflicts with some edge in E_1. If two rules r_1, r_2 are not conflicting, then they may be applied in "parallel" at the same step and "serialized" arbitrarily, enabling the solver to find much longer witnesses at shallower depths. The corresponding assertion is $\mathsf{ruleConflict}(r_1, r_2, i)$. Assertion $\mathsf{someRule}(i)$ says some rule applies at depth i, and $\mathsf{ruleApplicability}(r,i)$ expresses the application of a rule r at depth i.

The assertion $\mathsf{ruleToEdge}(e,i)$ enforces that a rule $r\colon e \leftarrow E$ is applicable at depth i only if the corresponding edge e is present at that depth, which we define as follows (and is not shown in Table 1:

$$\mathsf{Edg}(e,i+1) \Leftrightarrow \left(\begin{array}{ll} (\mathsf{Edg}(e,i) \wedge \left(\bigwedge_{(r\colon e \leftarrow E) \in R} \neg\,\mathsf{Rl}(r,i)\right)) & \textit{/e existed previously/} \\ \quad \vee \bigvee_{(r'\colon e' \leftarrow E) \in R \text{ s.t. } e \in E} \mathsf{Rl}(r',i) & \textit{/or rule } r' \textit{ creates } e/ \end{array} \right)$$

During the witness search, if we encounter an edge set E_i at depth i, such that $E_i \supseteq E_j$ for a smaller depth $j < i$, then the search can be stopped along that branch and a different set of rule applications should be explored. This aspect is captured by $\mathsf{notSubsumes}(i,j)$, and in the overall encoding below, we have such a clause for all depths $i > j$.

Overall Encoding. The overall encoding for an edge e_{query} is the conjunction:

$$\varphi(e_{\text{query}}, k)\colon \bigwedge_{i \in [1,k]} \left[\begin{array}{ll} & \mathsf{init}(e_{\text{query}}) \\ \bigwedge_{e_1,e_2 \text{ conflicting}} & \mathsf{edgeConflict}(e_1,e_2,i) \\ \bigwedge_{r_1,r_2 \text{ conflicting}} & \mathsf{ruleConflict}(r_1,r_2,i) \\ \wedge & \mathsf{someRule}(i) \\ \wedge \bigwedge_{r \in R} & \mathsf{ruleApplicability}(r,i) \\ \wedge \bigwedge_{e \in E} & \mathsf{ruleToEdge}(e,i) \\ \wedge \bigwedge_{j \in [1,i-1]} & \mathsf{notSubsumes}(i,j) \end{array} \right].$$

The overall witness search for edge e_{query}, consists of increasing the depth bound k incrementally until either (A) $\varphi(e_{\text{query}}, k)$ is unsatisfiable indicating a proof of unrealizability of the edge e_{query}, or (B) $\varphi(e_{\text{query}}, k) \wedge \mathsf{emptySet}(k+1)$ is satisfiable yielding a witness, wherein, the clause $\mathsf{emptySet}(i)\colon \bigwedge_{e \in E} \neg \mathsf{Edg}(e, i)$ encodes an empty set of edges.

Lemma 1. *(A) If $\varphi(e, k)$ is unsatisfiable then there cannot exist a witness for e for any depth $l \geq k$; and (B) If $\varphi(e, k) \wedge \mathsf{emptySet}(k+1)$ is satisfiable then there is a witness for the realizability of the edge e.*

4 Is There a Precision Gap in Practice?

We now describe our implementation of the ideas described thus far and the evaluation of these ideas to determine the size of the precision gap between Andersen's analysis and precise flow-insensitive analysis.

Implementation. Our implementation uses the C language front-end CIL [13] to generate a set of pointer analysis constraints for a given program. The constraint generator is currently field insensitive. Unions, structures, and dynamic memory allocation are handled with summary locations. To resolve function pointers, our constraint generator uses CIL's built-in Steensgaard analysis [18]. The constraints are then analyzed using our own implementation of Andersen's analysis. Our implementation uses a *semi-naive iteration* strategy to handle changes in the pointer graphs incrementally [14]. Other optimizations such as cycle detection have not been implemented, since our implementation of Andersen's analysis is not the scalability bottleneck for our experiments.

Our implementation of witness generation uses the symbolic witness search algorithm outlined in Sect. 3.3. Currently, our implementation uses the SMT solver Yices [6]. Note that the witness search directly handles statements with multiple dereferences from the original program, so the additional temporaries generated to run Andersen's analysis do not introduce imprecision in the search.

Evaluation Methodology. We performed our experiments over a benchmark suite consisting of 12 small- to medium-sized C benchmarks representing various Linux system utilities including network utilities, device drivers, a terminal application, and a system daemon. All measurements were taken on an 2.93 GHz Intel Xeon X7350 using 3 GB of memory.

To measure the precision gap for points-to analysis, we ran our witness search for all of the Andersen's points-to results for the benchmarks, both with over- and under-approximate handling of summary locations (yielding a lower and upper bound on the precision gap respectively, as described in Sect. 3.2). The primary result of this paper is that we found *no precision gap* between Andersen's analysis and the precise flow-insensitive analysis in either of these experimental configurations. In other words, our witness search never produced a refutation over our 12 benchmarks, no matter if summary locations were handled over- or under-approximately, and with precise handling of statements with multiple dereferences.

Following our observation that no precision gap exists for points-to queries, it is natural to consider if there is a precision gap between using Andersen's analysis to resolve alias queries and a precise flow-insensitive alias analysis. We say that p *aliases* q if there is a common location r such that both p and q may simultaneously point to r. We adapted the witness search encoding to search for witnesses for aliasing between pairs of variables that Andersen's analysis indicated were may-aliased. For aliasing experimental configurations, we ran the alias witness search for 1000 randomly chosen pairs of variables for each of our benchmarks (whereas for points-to configurations, we exhaustively performed witness search on all edges reported by Andersen's). Even though realizability of alias relations is more constrained than that of points-to relations, the search still produced a witness for all alias queries. This observation provides evidence that there is also likely no precision gap for alias analysis.

Results. As stated above, we found a flow-insensitive witness for *every points-to relation* and *every alias query* for our benchmarks in each experimental configuration. We found refutations for small hand-crafted examples that demonstrate the precision gap (like Examples 1 and 2), but not in real programs.

Table 2 gives details about the benchmarks and the execution of our witness-generating analyses. We show the statistics for two experimental configurations: the over- and under-approximating analyses for points-to queries with search over the original program statements.

Witness Search with Weak-Update Witnesses (Weak). For each points-to edge computed by Andersen's analysis, we performed a symbolic witness search using edge dependency rules derived with the original program statements until either a witness or a refutation for the edge was found. Weak-update semantics were used for summaries (see Sect. 3.2), yielding an over-approximate analysis and a lower bound on the precision gap.

Witness Search with Concretization (Conc). Here, we performed an under-approximate witness search that treated summaries as concrete locations, as described in Sect. 3.2. As refutations produced in this configuration may be invalid (due to the under-approximation), the configuration gives an upper bound on the precision gap.

The benchmarks are organized by function and sorted by number of lines of code in ascending order. The first set of columns gives statistics on the problem size, while the second set shows number of rules, analysis times, and search depths for each configuration. We note that running time depends primarily on the number of rules available to the witness search.

In Fig. 2, we show the per-benchmark distribution of discovered witness lengths for both the Weak configuration (left) and Conc configuration (right). Comparing each benchmark across the two configurations, we see relatively little change in the distribution. This result is a bit surprising, as one may expect that the more constraining Conc configuration would be forced to find longer witnesses. We hypothesize that the flow-insensitive abstraction allows so much flexibility in witness generation that there are always many possible witnesses

Table 2. Data from experiments using the WEAK and CONC configurations. The "Program Size" columns give the number of thousands of lines of code (kloc), variables (vars), and pointer constraints (cons). Note that the number of variables includes all program variables (pointer type or non-pointer type), as any type may be used a pointer in C. The "Problem Size" columns give the number of rules generated (rules) and number of points-to edges found by Andersen's (edges). For the WEAK and CONC experiments, we give the average search depth required and total running time.

	Program Size			Problem Size		WEAK		CONC	
Benchmark	kloc	vars	cons	rules	edges	depth	time (s)	depth	time (s)
-NETWORK UTILITIES-									
aget (ag)	1.1	198	86	21	21	1.4	0.0	1.4	0.0
arp (ar)	3.1	1052	144	31	30	1.5	0.1	1.5	0.0
slattach (sl)	3.4	1046	164	31	31	1.5	0.1	1.5	0.0
netstat (ne)	4.5	1333	205	85	80	1.5	0.1	1.5	0.1
ifconfig (if)	8.8	1334	702	224	195	1.9	0.4	1.9	0.5
plip (pl)	18.4	4298	1556	167	146	2.5	1.0	2.7	1.2
-DEVICE DRIVERS-									
knot (kn)	1.3	243	125	22	21	1.7	0.0	1.7	0.0
esp (es)	10.9	3805	1475	6979	413	3.9	12937.0	4.2	734.0
ide-disk (id)	12.6	4684	1290	422	274	5.0	42.1	5.1	53.4
synclink (sy)	23.6	5221	2687	164	157	1.2	0.2	1.2	0.2
-TERMINAL APPLICATIONS-									
bc (bc)	6.2	658	615	1098	244	3.6	129.7	3.6	124.0
-DAEMONS-									
watchdog (wa)	9.4	1189	760	196	163	2.7	1.1	2.7	1.1

for each points-to relation regardless of whether we use the WEAK or CONC configuration. The median witness length for WEAK and CONC was 4, while the mean lengths were 8.41 and 8.58, respectively. Note that the mean lengths significantly exceeded the mean search depth for the benchmarks, indicating the effectiveness of parallel rule application in the search (see Sect. 3.3). The longest witness found in either configuration was of length 54.

4.1 Discussion: Why is There No Precision Gap in Practice?

We note that the phenomena reported here as such defy a straightforward explanation that reflects directly on the way pointers are typically used in C programs.

From our experiments, we observe that not only does every edge discovered by Andersen's analysis have a witness, but it potentially has a *large number* of witnesses. This hypothesis is evidenced by the fact that we can (a) deploy non-conflicting rules in parallel and (b) discover long witnesses at a much smaller search depth. As a result, each witness consists of *parallel threads* of unrelated

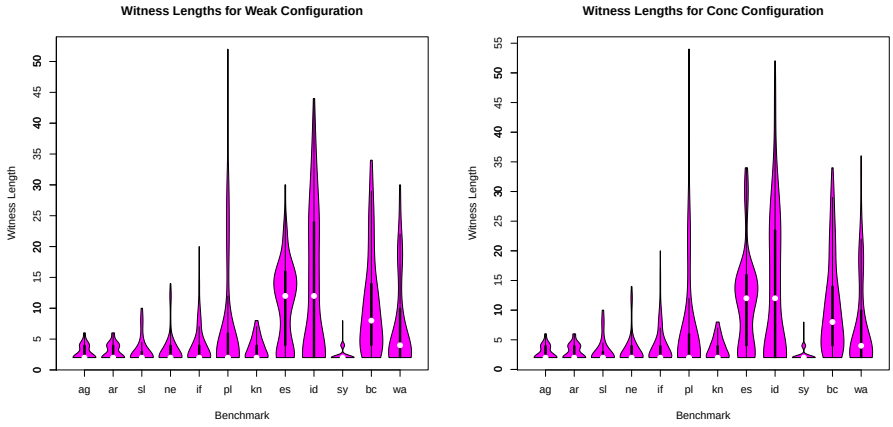


Fig. 2. Distribution of witness lengths for the WEAK and CONC configurations. The white dot shows the median length, the endpoints of the thick line give the first and last quartiles, and the thin line indicates the first and last deciles. The width of the plot indicates the (relative) density of witnesses for a given length.

pointer assignments that contribute towards the final goal but can themselves be *interleaved* in numerous ways.

Recall that the rules obtained from Andersen's analysis are of the form $e \xleftarrow{a} \{e_1, e_2\}$, stating that if e_1, e_2 are simultaneously realizable then e is realizable by application of assignment a. Therefore, unrealizability of e means that for every such rule that can realize e, the corresponding RHS set $\{e_1, e_2\}$ are simultaneously unrealizable. In turn, this indicates that any sequence of assignments that realizes e_1 destroys e_2 and vice versa. Such "mutually-destructive" pairs of points-to relations are easy to create and maintain in programs. However, these examples depend on sequential control flow to produce the desired behavior. When analyzed under flow-insensitive semantics wherein statements can occur multiple times under varying contexts, the behavior changes drastically.

Other examples of imprecisions in points-to analysis depend on the proper modeling of function calls and returns. For example, the following code may be used to initialize a linked list:

```
void initList(List* l) { l->header->next = l->header->prev = l->header; }
```

If this function were invoked at multiple contexts with different arguments, Andersen's analysis could conflate the internal list pointers while a precise flow-insensitive analysis would not (assuming a context-insensitive treatment of the function). However, note that this precision gain would require the ability to distinguish the list nodes themselves, for which flow-insensitive analysis is often insufficient. Furthermore, the precision gain would be quite fragile; if the above source is rewritten to store `l->header` in a temporary variable, the gain disappears. Stepping out of pure flow-insensitive analysis, a partially-flow-sensitive

analysis [17] would be more robust to such changes and may be worth future investigation.

4.2 Threats to Validity

One threat to the validity of our results is that they may be sensitive to how various C language constructs are modeled by our constraint generator. It is possible that field sensitivity, (partial) context sensitivity, or a more precise treatment of function pointers would expose a precision gap. However, given the exacting conditions required for a gap to arise, we believe it is unlikely that these other axes of precision would affect our results in any significant way.

It is also possible that our benchmarks are not representative of small- to medium-sized C programs. To mitigate this concern, we chose benchmarks from several domains: network utilities, device drivers, a command-line application, and a system daemon. We also attempted to select programs of different sizes within the spectrum of small- to medium sized programs. Although no benchmark suite can be representative of all programs, our intent was to choose a reasonable number of programs with diverse sizes and uses to comprise a set that adequately represents small- to medium-sized C programs.

Finally, it may be that the precision gap only manifests itself on larger programs than the ones we considered. We have tried to perform measurements on examples in the 25 to 200 kloc range, but such examples are presently beyond the reach of our implementation. We are currently investigating implementing ideas along the lines of bootstrapping [10], wherein the witness search may focus on a smaller subset of edges in the points-to graph and allow our experiments to scale to larger programs. Despite our inability to scale to programs beyond 25k lines, we hypothesize that our conclusion generalizes to larger programs based on the intuitions outlined in Sect. 4.1.

5 Related Work

Our work was partially inspired by previous work on the complexity of precise points-to analysis variants. Horwitz [9] discussed the precision gap between Andersen's analysis and precise flow-insensitive analysis and proved the NP-hardness of the precise problem. Chakaravarthy [5] gave a polynomial-time algorithm for precise flow-insensitive analysis for programs with well-defined types.

Muth and Debray [12] provide an algorithm for a variant of precise flow-sensitive points-to analysis (for programs without dynamic memory) that can be viewed as producing witnesses by enumerating all possible assignment sequences and storing the exact points-to graph, yielding a proof of PSPACE-completeness. Others have studied the complexity and decidability of precise flow-sensitive and partially-flow-sensitive points-to analysis [11, 15, 17].

The edge reduction rules derived in our approach are similar, in spirit, to the reduction from pointer analysis problems to graph reachability as proposed by Reps [16]. However, a derivation in this CFL for a points-to edge need not

always yield a witness. In analogy with Andersen's analysis, the derivation may ignore conflicts in the intermediate configurations. Finding a derivation in a CFL without conflicting intermediate configurations reduces to temporal model checking of push-down systems. This observation, however, does not seem to yield a better complexity bound [4].

Our work employs SAT solvers to perform a symbolic search for witnesses to points-to edges. Symbolic pointer analysis using BDDs have been shown to outperform explicit techniques in some cases by promoting better sharing of information [2, 19].

6 Conclusion

We have presented techniques for measuring the precision gap between Andersen's analysis and precise flow-insensitive points-to analysis in practice. Our approach is based on refinement of points-to analysis results with a witness search and a symbolic encoding to perform the search with a tuned SAT solver. Our experimental evaluation showed that for medium-sized C programs, the precision gap between Andersen's and precise flow-insensitive analysis is (as far as we can observe) non-existent. Future work includes improving the scalability of our witness search algorithm and applying our techniques to other languages. We also plan to extend the witness search algorithm to incorporate higher levels of precision, including context sensitivity and some form of flow sensitivity.

Acknowledgments. We thank Jeffrey S. Foster for fruitful discussions on an earlier draft of this paper, as well as the anonymous reviewers for their helpful comments. The authors are also grateful to Gogul Balakrishnan, Franjo Ivancic, and Aarti Gupta at NEC Laboratories America in Princeton, NJ for helping us with the Linux device driver benchmarks used in our experiments. We also thank Jan Wen Voung, Ranjit Jhala, and Sorin Lerner for including a large set of C benchmarks in their publicly available Relay/Radar tool. This research was supported in part by NSF under grants CCF-0939991 and CCF-1055066.

7 References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[2] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Programming Language Design and Implementation (PLDI)*, pages 103–114, 2003.

[3] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of Andersen's analysis in practice (extended version). Technical Report CU-CS-1082-11, Department of Computer Science, University of Colorado Boulder, 2011.

[4] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Concurrency Theory (CONCUR)*, pages 135–150, 1997.

[5] V. T. Chakaravarthy. New results on the computability and complexity of points-to analysis. In *Principles of Programming Languages (POPL)*, pages 115–125, 2003.

[6] B. Dutertre and L. de Moura. The YICES SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`.

[7] B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, pages 290–299, 2007.

[8] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

[9] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1), 1997.

[10] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 249–259, 2008.

[11] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[12] R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *Principles of Programming Languages (POPL)*, pages 67–80, 2000.

[13] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction (CC)*, pages 213–228, 2002.

[14] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[15] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[16] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

[17] N. Rinetzky, G. Ramalingam, M. Sagiv, and E. Yahav. On the complexity of partially-flow-sensitive alias analysis. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[18] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996.

[19] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.