# Fast and Precise Handling of Positive Weight Cycles for Field-sensitive Pointer Analysis

Yuxiang Lei and Yulei Sui

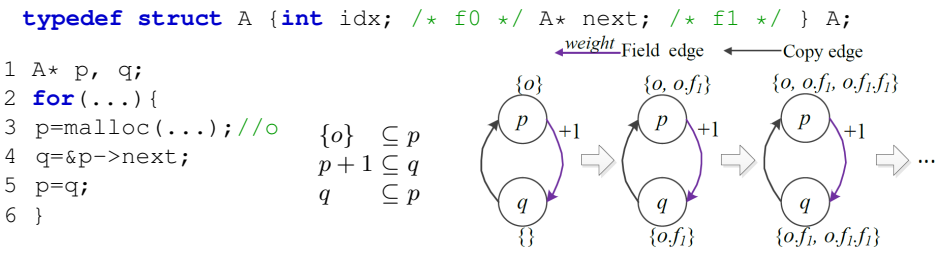
School of Computer Science, University of Technology Sydney, Australia

**Abstract.** By distinguishing the fields of an object, Andersen's field-sensitive pointer analysis yields better precision than its field-insensitive counterpart. A typical field-sensitive solution to inclusion-based pointer analysis for C/C++ is to add positive weights to the edges in Andersen's constraint graph to model field access. However, the precise modeling is at the cost of introducing a new type of constraint cycles, called *positive weight cycles* (*PWC*s). A *PWC*, which contains at least one positive weight constraint, can cause infinite and redundant field derivations of an object unless the number of its fields is bounded by a pre-defined value. *PWC*s significantly affect analysis performance when analyzing large C/C++ programs with heavy use of structs and classes.
This paper presents Dea, a fast and precise approach to handling of *PWC*s that significantly accelerates existing field-sensitive pointer analyses by using a new field collapsing technique that captures the *derivation equivalence* of fields derived from the same object when resolving a *PWC*. Two fields are derivation equivalent in a *PWC* if they are always pointed to by the same variables (nodes) in this *PWC*. A stride-based field representation is proposed to identify and collapse derivation equivalent fields into one, avoiding redundant field derivations with significantly fewer field objects during points-to propagation. We have conducted experiments using 11 open-source C/C++ programs. The evaluation shows that Dea is on average 7.1X faster than Pearce et al.'s field-sensitive analysis (Pkh), obtaining the best speedup of 11.0X while maintaining the same precision.

**Keywords:** Pointer analysis · Field-sensitive · Cycle elimination · Positive weight cycle.

## 1 Introduction

Pointer analysis, which statically approximates the runtime values of a pointer, is an important enabling technology that paves the way for many other program analyses, such as program understanding, bug detection and compiler optimisations. Andersen's analysis (or inclusion-based analysis) represents one of the most commonly used pointer analyses for Java and C/C++ programs. Field-sensitivity is an important precision enhancement that is naturally used in Andersen's analysis for analyzing Java [1–4], but is rarely used in many Andersen's analyses for C/C++ [5–9].




(a) C code (b) Constraints (c) Solving a *PWC* on the constraint graph

Fig. 1: A positive weight cycle example in Pearce's field-sensitive analysis.

Developing field-sensitive analysis for C/C++ is much harder than that for Java. The key difficulty, as also mentioned in [10, 11], is that the address of a field can be taken in C/C++ (via an *address-of-field* instruction `q=&p→f`), whereas Java does not permit taking the address of a field. Accessing the value of a field in Java is through the load/store instruction associated with an extra field specifier in Java's bytecode given its strongly-typed language feature. However, in the C/C++ intermediate representation (e.g., LLVM IR), a load/store only accepts a single pointer operand without a field specifier even for the reading/writing values of a field. The address taken by the pointer operand needs to be computed by the analysis itself to identify which field of an object the load/store may access.

To simplify the complicated field-sensitivity in C/C++, the majority of the works on Andersen's analysis are field-insensitive (i.e., accessing a field of an object is treated as accessing the entire object). One representative field-sensitive analysis proposed by Pearce et al. [10] offers a field-index-based object modeling, which distinguishes the fields of an object by their unique indices (with nested structs expanded), yielding better precision than field-insensitive analysis [10, 11]. The approach extends Andersen's inclusion constraints [12] to differentiate an *address-of-field* instruction `q=&p→f`$_i$ from a simple *copy* instruction `q=p` by adding a positive weight $i$ to the field-insensitive constraint $p \subseteq q$ to obtain the field-sensitive one $p+i \subseteq q$, indicating that $q$ points to $i$-th field of an object $o$ that $p$ points to. In contrast, field-insensitive analysis imprecisely assumes that $p$ and $q$ both point to object $o$ based on the non-weighted constraint $p \subseteq q$.

The field-sensitive points-to relations are resolved by computing the dynamic transitive closure on top of the extended Andersen's constraint graph, where each node represents a variable and each edge denotes an inclusion constraint between two variables. One key challenge for field-sensitive analysis is to detect and resolve a new type of cycles, called *positive weight cycles* (*PWCs*) on the constraint graph. A *PWC* is a cycle containing at least one positive weighted constraint edge. A *PWC* differs from a normal constraint cycle (or non-*PWC* containing only copy constraints) in two fundamental ways: (1) the points-to sets of variables in a non-*PWC* are identical after constraint resolution, but the points-to sets of variables in a *PWC* can be different, and (2) computing the

transitive closure of a non-$PWC$ terminates once a fixed-point is reached, but a $PWC$ can cause infinite derivations unless a maximum number of fields of each object is specified.

Figure 1 gives an example from [10, §4.1] to illustrate a PWC that incurs infinite derivations during constraint resolution. Figure 1(b) gives the constraints transformed from the code via Pearce et al.'s modeling [10]. Figure 1(c) shows its corresponding constraint graph with a $PWC$ containing a positive weighted edge from $p$ to $q$ ($p{+}1 \subseteq q$) and a simple copy edge from $q$ back to $p$ ($q \subseteq p$). An abstract object $o$ allocated at line 3 is initially added to $p$'s points-to set. Note that the object is modeled per allocation site (e.g., `malloc`) in Andersen's analysis. The constraint $p{+}1 \subseteq q$ derives a new field object given each object that $p$ points to. The new object is then propagated back to $p$ via $q \subseteq p$ for a new round of field derivation due to this $PWC$, resulting in infinitely deriving fields $o.f_1, o.f_1.f_1, ...$ from the base object $o$.

To avoid infinite derivations, Pearce et al. [10] set a maximum number of fields for each object to ensure that field access via an index is always within the scope of an object. For a stack and global object, its number of fields can be statically determined based on its declared types. However, a dynamically allocated heap object may have an unknown number of fields and is thus assumed to have as many as the largest struct in the program, causing redundant derivations.

To accelerate the constraint resolution, cycle elimination is a commonly used technique that merges nodes within a cycle into one node if the point-to sets of the nodes in this cycle are identical. However, the existing cycle elimination approaches [13, 14, 5, 6] in field-insensitive analysis can not be directly applied to solve $PWC$s in field-sensitive analysis. Unlike nodes in a non-$PWC$, nodes in a $PWC$ may not have identical points-to sets, thus collapsing all nodes in a $PWC$ leads to precision loss. Collapsing only non-$PWC$s following previous algorithms cannot solve the infinite derivation problem in field-sensitive analysis.

This paper presents Dea, a fast and precise approach to handling of $PWC$s in field-sensitive Andersen's analysis. Rather than cycle elimination, we present a field collapsing technique to solve $PWC$s by capturing *derivation equivalence*. Two fields derived from the same object are derivation equivalent when solving a $PWC$ if these fields are always pointed to by the same variables (nodes) in this $PWC$. A new *stride-based field representation* (Sfr) is proposed to identify and collapse derivation equivalent fields when field-sensitive constraints.

Our handling of $PWC$s significantly boosts the performance of existing field-sensitive analysis (e.g., [10] proposed by Pearce et al.), while achieving the same precision. By capturing derivation equivalence, Dea avoids redundant field derivations with greatly reduced overhead during points-to propagation, making constraint solving converge more quickly. Our precision-preserving handling of $PWC$s can be easily integrated into existing Andersen's field-sensitive analyses, and is also complementary to the state-of-the-art cycle elimination methods for non-$PWC$s. Our evaluation shows that Dea on average achieves a speed up of 7.1X over Pkh equipped with a recent cycle elimination technique, wave propagation [6] for analyzing 11 open-source large-scale C/C++ programs.

Table 1: Analysis domains, LLVM instructions, and constraint edges

| Analysis Domains | | | Instruction | Constraint | Type |
|---|---|---|---|---|---|
| $i, j, w$ | $\in \mathbb{Z}$ | Integer constants | $\mathtt{p} = \&\mathtt{o}$ | $p \xleftarrow{\text{AddrOf}} o$ | AddrOf |
| $o$ | $\in \mathcal{O}$ | Abstract objects | $\mathtt{p} = \mathtt{q}$ | $p \xleftarrow{\text{Copy}} q$ | Copy |
| $o.f_i$ | $\in \mathcal{F}$ | Abstract field objects | $\mathtt{p} = \&\mathtt{q} \rightarrow \mathtt{f_i}$ | $p \xleftarrow{\text{Field}_i} q$ | Field |
| $a, b, c$ | $\in \mathcal{A} = \mathcal{O} \cup \mathcal{F}$ | Address-taken variables | $\mathtt{p} = *\mathtt{q}$ | $p \xleftarrow{\text{Load}} q$ | Load |
| $p, q, r$ | $\in \mathcal{P}$ | Top-level variables | $*\mathtt{p} = \mathtt{q}$ | $p \xleftarrow{\text{Store}} q$ | Store |
| $u, v$ | $\in \mathcal{V} = \mathcal{A} \cup \mathcal{P}$ | Variables | | | |

The key contributions of this paper are:

- We present a fast and precise handling of positive weight cycles to significantly boost the existing field-sensitive Andersen's analysis by capturing derivation equivalence when solving $PWC$s.
- We propose a new stride-based field abstraction to identify and collapse a sequence of derivation equivalent fields.
- We have implemented Dea in LLVM-7.0.0 and evaluated using 11 real-world large C/C++ programs. The results show that Dea on average is 7.1X faster than Pearce et al.'s field-sensitive analysis with the best speedup of 11.0X.

## 2 Background and Motivating Example

This section introduces the background of field-sensitive Andersen's analysis, including program representation, abstract object modeling and inference rules. We then give a motivating example to explain the key idea of derivation equivalence when resolving $PWC$s.

### 2.1 Program Representation and Field-sensitive Analysis

We perform our pointer analysis on top of the LLVM-IR of a program, as in [15–17, 11, 18]. The domains and the LLVM instructions relevant to field-sensitive pointer analysis are given in Table 1. The set of all variables $\mathcal{V}$ is separated into two subsets, $\mathcal{A} = \mathcal{O} \cup \mathcal{F}$ which contains all possible abstract objects and their fields, i.e., *address-taken variables* of a pointer, and $\mathcal{P}$ which contains all *top-level variables*, including stack virtual registers (symbols starting with "%") and global variables (symbols starting with "@") which are explicit, i.e., directly accessed. Address-taken variables in $\mathcal{A}$ are implicit, i.e., accessed indirectly at LLVM's load or store instructions via top-level variables.

After the SSA conversion, a program is represented by five types of instructions: $p = \&o$ (AddrOf), $p = q$ (Copy), $p = \&q \rightarrow f_i$ (Field or Address-of-field) $p = *q$ (Load) and $*p = q$ (Store), where $p, q \in \mathcal{P}$ and $o \in \mathcal{O}$. Top-level variables are put directly in SSA form, while address-taken variables are only accessed indirectly via Load or Store. For an AddrOf $p = \&o$, known as an *allocation site*, $o$ is a stack or global variable with its address taken or a dynamically created

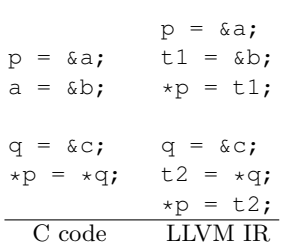



Fig. 2: C code fragment and its LLVM IR.

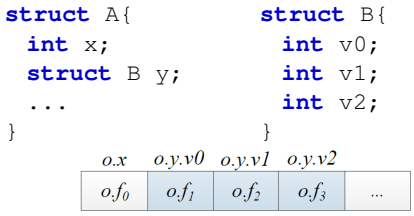



Fig. 3: The flattened fields with their unique indices (i.e., $o.f_0$, $o.f_1$, $o.f_2$, $o.f_3$, ... ) for object $o$ of type **struct A**

.

abstract heap object (e.g., via `malloc()`). Parameter passings and returns are treated as Copys.

Figure 2 shows a code fragment and its corresponding partial SSA form, where $\mathtt{p}, \mathtt{q}, \mathtt{t1}, \mathtt{t2} \in \mathcal{P}$ and $\mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathcal{A}$. Note that `a` is indirectly accessed at a store $*\mathtt{p} = \mathtt{t1}$ by introducing a top-level pointer `t1` in the partial SSA form. Complex statements such as $*\mathtt{p} = *\mathtt{q}$ are decomposed into basic instructions by introducing a top-level pointer `t2`.

Our handling of field-sensitivity is ANSI-compliant [19]. For each struct allocation e.g., $p = \&o$, a field-insensitive object $o$ is created to represent the entire struct object. The fields of a struct are distinguished by their unique indices [10, 11] with the fields of nested structs flattened as illustrated in Figure 3. A field object denoted by $o.f_i$ is derived from $o$ when analyzing Field $\mathtt{q} = \&\mathtt{p} \rightarrow \mathtt{f_i}$ (LLVM's getelementptr instruction), where $f_i$ denotes the $i$-th field of $o$ and $i$ is a constant value. Following [10], the address of $o$ is modeled by the address of its first field with index 0. All other fields are modeled using distinct subobjects. Two pointer dereferences are aliased if one refers to $o$ and another refers to one of its fields e.g., $o.f_i$, since it is the sub component of $o$. However, dereferences refer to distinct fields of $o$ (e.g., $o.f_2$ and $o.f_3$) which are distinguished and not aliased.

For a C pointer arithmetic (e.g., $q = p+j$), if $p$ points to a struct object $o$, we conservatively assume that $q$ can point to any field of this struct object, i.e., the entire object $o$. This is based on the assumption that the pointer arithmetic is not across the boundary of the object. Similar to previous practices for analyzing C/C++, the analysis can be unsound if a pointer arithmetic used to access an aggregate object is out of the boundary or arbitrary castings between a pointer and an integer. Arrays are treated monolithically, i.e., accessing any element of an array is treated as accessing the entire array object.

In Andersen's analysis [12], resolving the points-to sets $pts(v)$ of a variable $v$ is formalized as a set-constraint problem on top of the constraint graph $G = \langle V, E \rangle$, where each node $v \in V$ represents a variable, and an edge $e \in E$ between two nodes represents one of the five types of constraints (Table 1). Figure 4 gives the inference rules of field-sensitive analysis, which solves a dynamic transitive closure on $G$ by propagating points-to information following the es-

$$\texttt{[ADDROF]}\ \frac{p \xleftarrow{\mathsf{AddrOf}} o}{o \in pts(p)} \qquad \texttt{[COPY]}\ \frac{v \xleftarrow{\mathsf{Copy}} u}{pts(u) \subseteq pts(v)}$$

$$\texttt{[FIELD-1]}\ \frac{p \xleftarrow{\mathsf{Field}_i} q \quad o \in pts(q)}{o.f_i \in pts(p)} \qquad \texttt{[FIELD-2]}\ \frac{p \xleftarrow{\mathsf{Field}_i} q \quad o.f_j \in pts(q)}{o.f_{i+j} \in pts(p)}$$

$$\texttt{[STORE]}\ \frac{\mathtt{p} \xleftarrow{\mathsf{Store}} \mathtt{q} \quad a \in pts(p)}{a \xleftarrow{\mathsf{Copy}} q} \qquad \texttt{[LOAD]}\ \frac{p \xleftarrow{\mathsf{Load}} q \quad a \in pts(q)}{p \xleftarrow{\mathsf{Copy}} a}$$

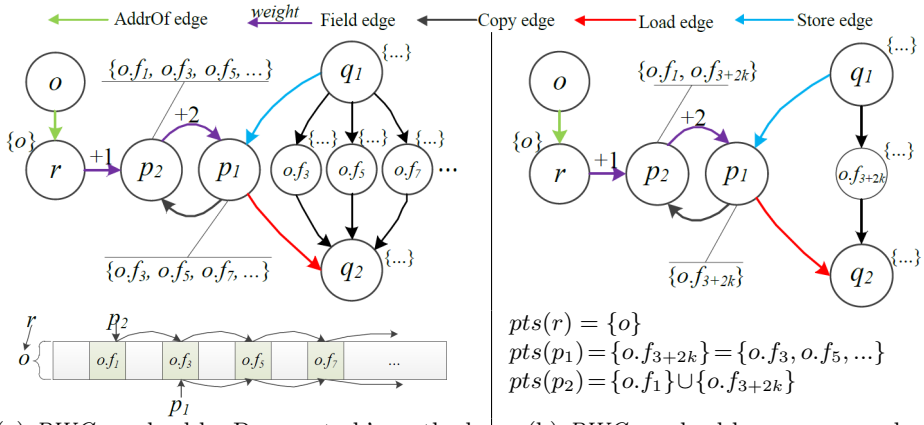
Fig. 4: Inference rules of Pearce et al.'s field-sensitive Andersen's analysis

tablished Copy/Field edges and by adding new Copy edges until a fixed-point is reached [12].

### 2.2 A Motivating Example

Figure 5 gives an example to show the redundant derivations when solving a $PWC$ on the constraint graph by PKH [10] (Pearce et al.'s field-sensitive analysis) based on its inference rules (Figure 4). We illustrate how our idea captures the derivation equivalence by using a stride-based representation to collapse fields which are always pointed to by all the pointers in this $PWC$. The example consists of five types of constant edges corresponding to the five types of instructions in Table 1 with one $PWC$ involving nodes $p_1$ and $p_2$. Pointer $r$ initially points to $o$ ([ADDROF]). The points-to set of $p_2$ has the field $o.f_1$ derived from the object $o$ when resolving $p_2 \xleftarrow{\mathsf{Field}_1} r$ ([FIELD-1]). Since $p_1 \xleftarrow{\mathsf{Field}_2} p_2$ and $p_2 \xleftarrow{\mathsf{Copy}} p_1$ form a $PWC$ with a positive weight $+2$, a sequence of field objects starting from $o.f_3$ with a stride 2 are iteratively derived and added into $p_1$'s points-to set ([FIELD-2]) and then propagated back to $p_2$ ([COPY]). These field objects are derivation equivalent because all the fields are always pointed to by both $p_1$ and $p_2$ in this $PWC$, incurring redundant derivations. Even worse, the edge $p_1 \xleftarrow{\mathsf{Store}} q_1$ flowing into and the edge $q_2 \xleftarrow{\mathsf{Load}} p_1$ going out of this $PWC$ add redundant Copy edges (e.g., $o.f_3 \xleftarrow{\mathsf{Copy}} q_1$ and $q_2 \xleftarrow{\mathsf{Copy}} o.f_3$)) based on [STORE] and [LOAD], causing redundant points-to propagation, as also illustrated in Figure 5(a).

To avoid redundant field derivations and unnecessary Copy edges when resolving Load and Store. Our idea is to merge derivation equivalent fields into a stride-based polynomial representation $o.f_{i+ks}$, where $i$ is the starting field, $s$ is the stride corresponding to the weight of the $PWC$, and $k \in \mathbb{N}$. Figure 5(b) illustrates the new representation $o.f_{3+2k}$ for collapsing equivalent fields $\{o.f_3, o.f_5, ...\}$ in Figure 5(a). The new representation successfully reduces the number of points-to targets during points-to propagation and the number of Copy edges added into the constraint graph when solving Store/Load edges, while maintaining the same precision, i.e., the points-to sets of $r, p_1, p_2$ (after expanding the fields based on the polynomial representation) are identical to those produced by PKH.




(a) *PWC* resolved by Pearce et al.'s method (b) *PWC* resolved by our approach

Fig. 5: A motivating example.

## 3 Our Approach

This section details our approach to handling of *PWC*s in field-sensitive pointer analysis, including the stride-based field abstraction to represent derivation equivalent fields and the inference rules based on the new field representation.

### 3.1 Stride-based Field Representation

**Definition 1 (Stride-based Field Representation (SFR)).** *We use $\sigma = \langle o, i, S\rangle$ to denote a single object or a sequence of fields in Pearce et al.'s modeling starting from $i$-th field following the strides in $S$. The field expansion of $\langle o, i, S\rangle$ is as follows:*

$$FX(\langle o,i,S\rangle) = \begin{cases} \{o\} & \text{if } S=\emptyset \wedge i=0 \\ \{o.f_j \,|\, j = i + \sum_{n=1}^{|S|} k_n s_n,\ j \le max, k_n \in \mathbb{N}, s_n \in S\} & \text{otherwise} \end{cases}$$

where $max$ denotes the maximum number of fields of object $o$ and $s_n$ is the $n$-th element of the stride set $S$ which models precisely field derivations when a Field edge resides in one or multiple *PWC*s. We use $\langle o, 0, \emptyset\rangle$ to represent the entire object $o$ and its single field $o.f_i$ is denoted by $\langle o, i, \{0\}\rangle$. SFR unifies the notations of an object and its fields. The expansion of an SFR fully represents the objects and fields in Pearce et al.'s modeling, while it reduces the number of points-to targets during constraint solving. Two SFRs can be disjointed or overlapping (Definition 2).

$$\texttt{[E-ADDROF]}\ \frac{p \xleftarrow{\text{AddrOf}} o \quad \sigma = \langle o, 0, \emptyset \rangle}{\sigma \in pts(p)} \qquad \texttt{[E-COPY]}\ \frac{v \xleftarrow{\text{Copy}} u}{pts(u) \subseteq pts(v)}$$

$$\texttt{[E-FIELD]}\ \frac{p \xleftarrow{\text{Field}_i} q \quad \langle o, j, S \rangle \in pts(q) \quad S' = Strides(p \xleftarrow{\text{Field}_i} q) \quad \sigma = \langle o, i+j, S \cup S' \rangle}{\nexists\, \sigma' \in pts(p) : \sigma \sqsubseteq \sigma' \Rightarrow \sigma \in pts(p)}$$

$$\texttt{[E-STORE]}\ \frac{p \xleftarrow{\text{Store}} q \quad \sigma \in pts(p)}{\sigma \xleftarrow{\text{Copy}} q} \qquad \texttt{[E-LOAD]}\ \frac{p \xleftarrow{\text{Load}} q \quad \sigma \in pts(q)}{\forall \sigma' : \sigma \sqcap \sigma' \neq \emptyset \Rightarrow p \xleftarrow{\text{Copy}} \sigma'}$$

$$Strides(e) = \begin{cases} \{0\} & \text{if edge } e \text{ is not in any } PWC \\ \{\mathcal{W}_\mathcal{C} \mid \forall \mathcal{C} \subseteq E : e \in \mathcal{C}\} & \text{otherwise} \quad \text{(Definition 4)} \end{cases}$$

Fig. 6: Inference rules of our approach

**Definition 2 (Overlapping and disjointed SFRs).** *Two* SFR*s are overlapping, denoted as* $\sigma \sqcap \sigma' \neq \emptyset$ *if* $\sigma = \sigma'$ *or two different* SFR*s derived from the same object* $o$ *have at least one common field, i.e.,* $FX(\langle o, i, S \rangle) \cap FX(\langle o, i', S' \rangle) \neq \emptyset$*. A special case is the subset relation between two overlapping* SFR*s, denoted as* $\sigma \sqsubseteq \sigma'$*, i.e.,* $FX(\langle o, i, S \rangle) \subseteq FX(\langle o, i', S' \rangle)$*. We say that two* SFR*s are disjointed if* $\sigma \sqcap \sigma' = \emptyset$*.*

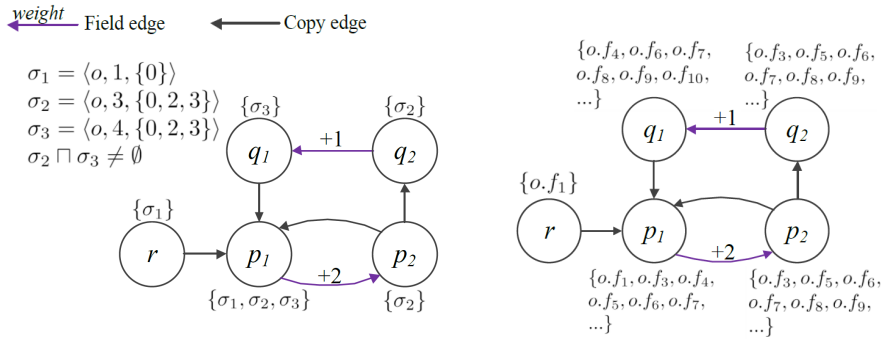
*Example 1 (Field expansion).* The expanded fields of $\langle o, 1, \{2\} \rangle$ are $FX(\sigma) = \{o.f_1, o.f_3, o.f_5 ...\}$. Likewise, the fields represented by $\langle o, 1, \{5, 6\} \rangle$ are $FX(\sigma) = \{o.f_j \mid j = 1 + 5k_1 + 6k_2, k_1, k_2 \in \mathbb{N}\} = \{o.f_1, o.f_6, o.f_7, o.f_{11}, o.f_{12}, ...\}$.

### 3.2 Inference Rules

Figure 6 gives the inference rules of our field-sensitive points-to analysis based on the stride-based field representation for resolving the five types of constraints. Object and field nodes on the constraint graph are now represented by the unified SFRs. Rule `[E-ADDROF]` initializes the points-to set of $p$ with object $o$ represented by $\langle o, 0, \emptyset \rangle$ (Definition 1) for each $p \xleftarrow{\text{AddrOf}} o$. Similar to `[COPY]` in Figure 4, `[E-COPY]` simply propagates the points-to set of $u$ to that of $v$ when analyzing $v \xleftarrow{\text{Copy}} u$.

**Definition 3 (Path and cycle).** *A path* $u \xLeftarrow{*} v$ *on the constraint graph* $G = \langle V, E \rangle$ *is a sequence of edges leading from* $v$ *to* $u$*. A path* $v \xLeftarrow{*} v$ *is called a closed path. A closed path* $v \xLeftarrow{*} v$ *is a* ***cycle*** *if all its edges are distinct and the only node to occur twice in this path is* $v$*.*

**Definition 4 (Weight of a *PWC*).** *A PWC, denoted as* $\mathcal{C}$*, is a cycle containing only* Copy *and* Field *edges and at least one edge is a* Field *with a positive weight. The weight of* $\mathcal{C}$ *is* $\mathcal{W}_\mathcal{C} = \sum_{e \in \mathcal{C}} wt_e$*, where* $e$ *is a* Field *or* Copy *and* $wt_e$ *is its weight (*$wt_e$ *is 0 if* $e$ *is a* Copy*). The set of weights of all the PWCs containing* $e$ *is* $\{\mathcal{W}_\mathcal{C} \mid \forall \mathcal{C} \subseteq E : e \in \mathcal{C}\}$




(a) DEA (b) PKH

Fig. 7: Solving the FIELD edge $p_2 \xleftarrow{\mathsf{Field}_2} p_1$ which involves in multiple $PWC$s

Unlike rule [FIELD-1] and [FIELD-2] in Figure 4 which generate a single field object when analyzing $p \xleftarrow{\mathsf{Field}_i} q$, [E-FIELD] generates an SFR $\sigma = \langle o, j + w, S \cup S' \rangle$ representing a sequence of fields starting from $(i+j)$-th field following strides $S \cup S'$, where $S' = \{0\}$ if $p \xleftarrow{\mathsf{Field}_i} q$ is not involved in any $PWC$, otherwise $S' = \{\mathcal{W}_\mathcal{C} \mid \forall \mathcal{C} \subseteq E : (p \xleftarrow{\mathsf{Field}_i} q) \in \mathcal{C}\}$, a set of the weights of all the positive weight cycles with each $\mathcal{C}$ containing $p \xleftarrow{\mathsf{Field}_i} q$ on the constraint graph (Definitions 3 and 4). If $p \xleftarrow{\mathsf{Field}_i} q$ is involved in multiple $PWC$s, $\sigma$ is derived to collapse as many equivalent fields as possible by considering the set of weights $S'$ of all the $PWC$s containing $p \xleftarrow{\mathsf{Field}_i} q$. The premise of [E-FIELD] ensures that $\sigma$ represents the derivation equivalent fields such that the targets added to the points-to sets of all these fields are always identical when solving each cycle $\mathcal{C}$. The conclusion of [E-FIELD] ensures early termination and avoids redundant derivations, since an SFR $\sigma$ can only be generated and added to $pts(p)$ if there no SFR $\sigma'$ already exists in $pts(p)$ such that $\sigma'$ can represent $\sigma$, i.e., $\sigma \sqsubseteq \sigma'$ (Definition 2). Examples 2 and 3 give two scenarios in which a Field edge resides in single and multiple $PWC$s.

*Example 2 ([E-FIELD] for a single PWC).* Let us revisit our motivating example in Figure 5 to explain [E-FIELD]. The Field edge $p_2 \xleftarrow{\mathsf{Field}_1} r$ is not involved in any $PWC$, therefore, [E-FIELD] generates an SFR $\sigma = \langle o, 1, \{0\}\rangle$ with $S' = \{0\}$, representing only field $o.f_1$ and it then adds $\sigma$ into $pts(p_2)$. Together with $p_2 \xleftarrow{\mathsf{Copy}} p_1$, the second Field edge $p_1 \xleftarrow{\mathsf{Field}_2} p_2 \in \mathcal{C}$ forms a positive weight cycle $\mathcal{C}$ with its weight $\mathcal{W}_\mathcal{C} = 2$. A new SFR $\sigma = \langle o, 1+2, \{0\} \cup \{2\}\rangle = \langle o, 3, \{0, 2\}\rangle$ is derived and added into $pts(p_1)$ given $\langle o, 1, \{0\}\rangle \in pts(p_2)$. The SFR $\langle o, 3, \{0, 2\}\rangle$ is then propagated back to $p_2$. In the second iteration for resolving $p_1 \xleftarrow{\mathsf{Field}_2} p_2$, the newly derived SFR $\langle o, 5, \{0, 2\}\rangle$ is discarded and not added into $pts(p_1)$ since $\langle o, 5, \{0, 2\}\rangle$ can be represented by $\langle o, 3, \{0, 2\}\rangle$, i.e., a subset relation $\langle o, 5, \{0, 2\}\rangle \sqsubseteq \langle o, 3, \{0, 2\}\rangle$ (Definition 2) holds.

*Example 3 (`[E-FIELD]` for multiple PWCs).* Figure 7 compares DEA with PKH to show that `[E-FIELD]` requires significantly fewer field derivations to resolve $p_2 \xleftarrow{\mathsf{Field}_2} p_1$ when it is involved in two *PWC*s, i.e., cycle $\mathcal{C}_1$ formed by $p_1$ and $p_2$, and $\mathcal{C}_2$ formed by $p_1$, $p_2$, $q_2$ and $q_1$. The weights of $\mathcal{C}_1$ and $\mathcal{C}_2$ are 2 and 3 respectively, therefore $S' = \{2, 3\}$. Initially, $p_1$ points to $\sigma_1 = \langle o, 1, \{0\}\rangle$, which is propagated to $p_1$ along $p_1 \xleftarrow{\mathsf{Copy}} r$. We first take a look at resolving $\mathcal{C}_1$. A new SFR $\sigma_2 = \langle o, 1{+}2, \{0\} \cup \{2, 3\}\rangle = \langle o, 3, \{0, 2, 3\}\rangle$ is derived and added to $pts(p_2)$ when analyzing $p_2 \xleftarrow{\mathsf{Field}_2} p_1$, as shown in Figure 7(a). $\sigma_2$ is then propagated back and added to $pts(p_1)$ along $p_1 \xleftarrow{\mathsf{Copy}} p_2$. The second iteration for analyzing $p_2 \xleftarrow{\mathsf{Field}_2} p_1$ avoids adding $\langle o, 5, \{0, 2, 3\}\rangle$ because it is a subset of $\sigma_2$, into $pts(p_2)$, resulting in early termination. Similarly, when resolving $\mathcal{C}_2$ which contains two Field edges, DEA generates $\sigma_3 = \langle o, 3{+}1, \{0, 2, 3\}\rangle$ when analyzing $q_1 \xleftarrow{\mathsf{Field}_1} q_2$ and then propagates $\sigma_3$ to $p_1$. Given this new $\sigma_3$ in $pts(p_1)$, $\langle o, 4{+}1, \{0, 2, 3\}\rangle$ is derived when again analyzing $p_2 \xleftarrow{\mathsf{Field}_2} p_1$ in $\mathcal{C}_2$. However, $\langle o, 4{+}1, \{0, 2, 3\}\rangle$, which is a subset of $\sigma_2$, is not added to $pts(p_2)$. Note that though $\sigma_2$ and $\sigma_3$ are overlapping due to the intersecting *PWC*s, $\sigma_2$ successfully captures the equivalent fields that are always pointed by $p_1, p_2, q_2$ and $\sigma_3$ captures the equivalent fields that are always pointed by $p_1, q_1$, avoiding redundant derivations. For each *PWC*, DEA generates only one SFR, requiring at most two iterations to converge the analysis. In contrast, PKH performs redundant derivations until it reaches the maximum number of fields of this object, as also illustrated in Figure 7(b).

Let us move to rules `[E-LOAD]` and `[E-STORE]`. Unlike `[STORE]` and `[LOAD]` in Figure 4, our handling of Store and Load is asymmetric for both efficiency and precision-preserving purposes. For $p \xleftarrow{\mathsf{Store}} q$, `[E-STORE]` is similar to `[STORE]` by propagating $pts(q)$ to $pts(\sigma)$, where $\sigma$ is pointed to by $p$. For an SFR $\sigma$ pointed to by $q$ at $p \xleftarrow{\mathsf{Load}} q$, `[LOAD]` propagates the points-to set of any $\sigma'$ which overlaps with $\sigma$ (Definition 2) to $pts(p)$. This is because a field $o.f_i$ in PKH may belong to one or multiple SFRs. For example, in Figure 7, $o.f_6$ belongs to $\sigma_2$ and $\sigma_3$ when resolving a Field edge which is involved in multiple cycles or in one *PWC* containing multiple Field edges. We use $\mathcal{M}_{o.f_i}$ to denote a set of all SFRs containing $o.f_i$, i.e., any two SFRs in $\mathcal{M}_{o.f_i}$ share common fields including at least $o.f_i$. According to Definition 1, any change to the point-to sets of $\sigma \in \mathcal{M}_{o.f_i}$ also applies to those of $o.f_i$ during our constraint resolution. If $*q$ at a Load refers to an SFR $\sigma$, it also refers $\sigma' \in \mathcal{M}_{o.f_i}$ that overlaps with $\sigma$ for each field $o.f_i \in FX(\sigma)$ (Definition 2). Therefore, `[LOAD]` maintains the correctness that $pts(o.f_i)$ obtains the union of the points-to sets of all SFRs in $\mathcal{M}_{o.f_i}$. Since a points-to target in $pts(\sigma)$ must be in the points-to set of every field in $FX(\sigma)$ (i.e, for any $\sigma \in \mathcal{M}_{o.f_i}$, $pts(\sigma)$ is always a subset of $pts(o.f_i)$), ensuring that no spurious points-to targets other than $pts(o.f_i)$ will be propagated to $p$ at the Load. Thus, our handling of *PWC*s is precision preserving, i.e., the points-to set of a variable after field expansion resolved by DEA is the same as that of PKH.

*Example 4 (`[E-LOAD]` and `[E-STORE]`).* Figure 8 illustrates the resolving of $p \xleftarrow{\mathsf{Store}} q$ and $r \xleftarrow{\mathsf{Load}} p$ with the initial points-to sets $pts(p) = \{\sigma_1\}$, $pts(q) = \{\sigma_3\}$

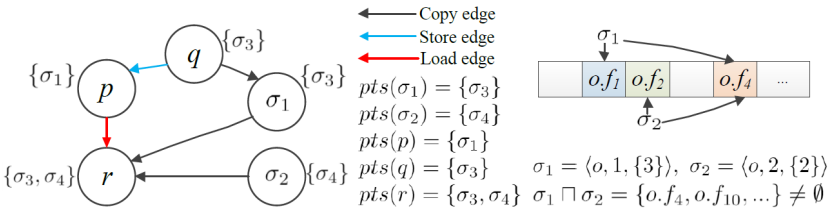



Fig. 8: Resolving Store $p \xleftarrow{\text{Store}} q$ and Load $r \xleftarrow{\text{Load}} p$ for overlapping SFRs

and $pts(\sigma_2) = \{\sigma_4\}$. $\sigma_1$ and $\sigma_2$ are both derived from object $o$ with overlapping fields, e.g., $o.f_4$, as highlighted in orange in Figure 8. When resolving $p \xleftarrow{\text{Store}} q$, `[E-STORE]` adds a new Copy edge $\sigma_1 \xleftarrow{\text{Copy}} q$, propagating $\sigma_3 \in pts(q)$ to $pts(\sigma_1)$, but not $pts(\sigma_2)$ though $\sigma_1 \sqcap \sigma_2 \neq \emptyset$. This avoids, for example, introducing the spurious target $\sigma_3$ to the points-to set of $o.f_2$ (in green), which only resides in $\sigma_2$ but not in $\sigma_1$. In contrast, `[E-LOAD]` resolves $r \xleftarrow{\text{Load}} p$ by adding two Copy edges $r \xleftarrow{\text{Copy}} \sigma_1$ and $r \xleftarrow{\text{Copy}} \sigma_2$, as also depicted in Figure 8. Since $\sigma_1 \sqcap \sigma_2 = \{o.f_4, ....\}$ and $\sigma_1 \in pts(p)$, if $*p$ at Load $r = *p$ refers to an overlapping field e.g., $o.f_4$ shared by $\sigma_1$ and $\sigma_2$, the points-to set of $r$ is the union of $pts(\sigma_1)$ and $pts(\sigma_2)$, i.e., $pts(r) = \{\sigma_3, \sigma_4\}$, achieving the precise field-sensitive results.

### 3.3 An Algorithm

Our precision-preserving handling of *PWC*s (i.e., the inference rules in Figure 6) can be integrated into existing constraint solving algorithms for field-insensitive Andersen's analysis, e.g., the state-of-the-art cycle elimination approaches [13, 14, 5, 6]. This section gives an overall algorithm of our approach by instantiating our inference rules on top of *wave propagation* [6], a constraint solving strategy with better or comparable performance as HCD/LCD [5] for analyzing large size programs.

In Algorithm 1, all the AddrOf edges are processed only once to initialize the worklist $W$ (lines 2-5), followed by a *while* loop for the main phase of constraint solving, which has three phases.

(1) SCC (strongly connected component) detection and weight calculation for *PWC*s (lines 7-9). We use Nuutilia et al.'s algorithm [20] to detect SCCs, which is an improvement over the original algorithm developed by Tarjan et al. [21]. The weight $W_{\mathcal{C}}$ of each positive weight cycle $\mathcal{C}$ is then calculated given the detected SCCs.

(2) Points-to propagation along Copy and Field edges (lines 10-24). We propagate points-to information along each Copy edge based on `[E-COPY]` (lines 12-15). New SFRs are derived and added to the points-to sets of the destination node of each Field edge based on `[E-FIELD]` (lines 16-22). A variable $v$ is pushed into a new worklist $W_{ind}$ if there exists an incoming Store edge to $v$ or an outgoing Load edge from $v$ for later handling of Loads/Stores (lines 23-24)

**Algorithm 1:** An Algorithm

```
1  Function DEA (G = ⟨V, E⟩)
2      W := ∅; W_ind := ∅
3      for each e : v ←AddrOf— o ∈ E do
4          ⟨o, 0, ∅⟩ ∈ pts(v)                                   } ▷[E-ADDROF]
5          W.push(v)                                            }
6      while W ≠ ∅ do
7          Compute SCC on G using Nuutilia's algorithm [20]
8          Collapse nodes in one SCC that contains only COPY edges
9          Calculate W_C for each cycle in SCCs
10         while W ≠ ∅ do
11             v := W.pop_front()
12             for each u ←Copy— v ∈ E do
13                 pts(v) ⊆ pts(u)                              } ▷[E-COPY]
14                 if pts(u) changed then                       }
15                     W.push(u)                                }
16             for each u ←Field_i— v ∈ E do
17                 S' := Strides(u ←Field_i— v)                 } ▷[E-FIELD]
18                 for each ⟨o, j, S⟩ ∈ pts(v) do               }
19                     σ := ⟨o, i+j, S∪S'⟩                      }
20                     if ∄ σ' ∈ pts(u) : σ ⊑ σ' then           }
21                         σ ∈ pts(u)                           }
22                         W.push(u)                            }
23             if ∃ v ←Store— u ∈ E or ∃ u ←Load— v ∈ E then
24                 push v into W_ind
25         while W_ind ≠ ∅ do
26             q := W_ind.pop_front()
27             for each v ←Store— u ∈ E do
28                 for each σ ∈ pts(v) do                       } ▷[E-STORE]
29                     if σ ←Copy— u then                       }
30                         E := E ∪ σ ←Copy— u                  }
31                         W.push(u)                            }
32             for each u ←Load— v ∈ E do
33                 for each σ' ∈ {σ' ⊓ σ ≠ ∅ | σ ∈ pts(v)} do   } ▷[E-LOAD]
34                     if u ←Copy— σ' then                      }
35                         E := E ∪ u ←Copy— σ'                 }
36                         W.push(σ')                           }
37         for each u ←Copy— v added by UPDATECALLGRAPH do
38             W.push(v)
```

(3) Processing Store and Load edges (lines 25-36). New Copy edges are added to $G$, and the source node of each newly added Copy edge is added to worklist $W$

Table 2: Basic characteristics of the benchmarks (IR's lines of code, number of pointers, number of five types of instructions on the initial constraint graph, and maximum number of fields of the largest struct in each program).

| | *LOC* | *#Pointers* | *MaxFields* | #Field | #Copy | #Store | #Load | #AddrOf |
|---|---|---|---|---|---|---|---|---|
| git-checkout | 1253K | 624K | 302 | 93201 | 88406 | 41620 | 60723 | 33380 |
| json-conversions | 355K | 264K | 64 | 27685 | 36557 | 37960 | 36872 | 43448 |
| json-ubjson | 330K | 233K | 64 | 24064 | 35813 | 34577 | 26288 | 34165 |
| llvm-as-new | 729K | 597K | 121 | 307167 | 77944 | 287634 | 41960 | 17435 |
| llvm-dwp | 1796K | 897K | 632 | 100877 | 101849 | 116205 | 142943 | 121541 |
| llvm-objdump | 728K | 353K | 121 | 61117 | 57743 | 56493 | 40314 | 16767 |
| opencv_perf_core | 1014K | 715K | 64 | 122744 | 192419 | 59599 | 79466 | 24450 |
| opencv_test_dnn | 889K | 635K | 64 | 105550 | 174080 | 52304 | 70332 | 22786 |
| python | 539K | 420K | 171 | 84779 | 74524 | 49215 | 56434 | 18340 |
| redis-server | 706K | 374K | 332 | 52178 | 60111 | 24542 | 39205 | 13175 |
| Xalan | 2192K | 807K | 133 | 110184 | 181804 | 35940 | 68812 | 53926 |

for points-to propagation in the next iteration. Lines 37-38 update the callgraph by creating new Copy edges (e.g., $u \xleftarrow{\text{Copy}} v$) for parameter/return passings when a new callee function is discovered at a callsite using the points-to results of function pointers obtained from this points-to resolution round. The source node $v$ of the Copy edge is added to $W$ to be processed in the next iteration until a fixed point is reached, i.e., no changes are made to the points-to set of any node.

Other field-sensitive analyses (e.g., PKH [10]) can also be implemented under the same constraint solving algorithm by simply replacing the lines for handling the five types of constraints with the inference rules in Figure 4.

## 4 Experimental Evaluation

The objective of our evaluation is to show that our field-sensitive analysis is significantly faster than Pearce et al.'s analysis (PKH) yet maintains the same precision in analyzing large size C/C++ programs.

### 4.1 Implementation and Experimental Setup

Our approach is implemented on top of LLVM-7.0.0 and its sub-project SVF [22, 18, 23]. A state-of-the-art constraint resolution algorithm, *wave propagation* [6] is used for cycle detection and computing dynamic transitive closures on top of the same constraint graph for both PKH and DEA. Indirect calls via function pointers are resolved on-the-fly during points-to resolution. A C++ virtual call $p \rightarrow foo()$ is translated into four low-level LLVM instructions for our pointer analysis. (1) a Load $vtptr = *p$, obtaining virtual table pointer $vtptr$ by dereferencing pointer $p$ to the object, (2) a Field $vfn = \&vtptr \rightarrow idx$, obtaining the entry in the vtable

at a designated offset *idx* for the target function, (3) a Load $fp = *vfn$, obtaining the address of the function, and (4) a function call $fp(p)$. Following [24, 25, 22], a white list is maintained to summarize all the side-effects of external calls (e.g., `memcpy`, `xmalloc` and `_Znwm` for C++ `new`) [26].

To evaluate the effectiveness of our implementation, we chose 11 large-scale open-source C/C++ projects downloaded from Github, including `git-checkout` (a sub project of Git for version control), `json-conversions` and `json-ubjson` (two main Json libraries for modern C++ environment, version 3.6.0), `llvm-as-new` and `llvm-dwp` (tools in LLVM-7.0.0 compiler), `opencv_perf_core` and `opencv_test_dnn` (two main libraries in OpenCV-3.4), `python` (version 3.4.2) and `redis-server` (a distributed database server, version 5.0). The source code of each program is compiled into bit code files Clang-7.0.0 [27] and then linked together using `WLLVM` [28] to produce whole program bc files.

Table 2 collects the basic characteristics about the 11 programs before the main pointer analysis phase. The statistics include the LLVM IR's lines of code (LOC) of a program, the number of pointers (#Pointers), the number of fields of the largest struct in the program, also known as the maximum number of fields using the upper bound for deriving fields of a heap object, and the number of each of the five types of constraint edges in the initial constraint graph. The reason that *#Field* is not much smaller than *#Copy* is twofold (1) Field refers to LLVM's getelementptr instruction, which is used to get the addresses of subelements of aggregates, including not only structs but also arrays and nested aggregates (Figure 3). (2) In low-level LLVM IR, a Copy only refers to an assignment between two virtual registers, such as casting or parameter passing (Section 2.1). An assignment "p = q" in high-level C/C++ is not translated into a Copy, but a Store/Load manipulated indirectly through registers on LLVM's partial SSA form.

All our experiments were conducted on a platform consisting of a 3.50GHz Intel Xeon Quad Core CPU with 128 GB memory, running Ubuntu Linux (kernel version 3.11.0).

### 4.2 Results and Analysis

Table 3 compares Dea with Pkh for each of the 11 programs evaluated in terms of the following three analysis results after constraint resolution, the total number of address-taken variables (*#AddrTakenVar*), the total number of fields derived when resolving all Field edges (*#Field*), and the number of fields derived only when resolving Field edges involving *PWC*s (*#FieldByPWC*). Both Dea and Pkh use `LLVM Sparse Bitvectors` as the points-to set implementation. The peak memory usage by Dea is 7.33G observed in git-checkout. Dea produces identical points-to results as those by Pkh, confirming that Dea's precision is preserved

From the results produced by Pkh, we can see that the number of fields (Column 4 in Table 3) occupies a large proportion of the total address-taken variables (Column 2) in modern large-scale C/C++ programs. On average, 72.5% of the address-taken variables are field objects. In programs `git-checkout` (written

Table 3: Comparing the results produced by Dea with those by Pkh, including the total number of address-taken variables, number of fields and the number of fields derived when resolving $PWC$s, and the number of Copy edges connected to/from the field object nodes derived when resolving $PWC$s

| | *#AddrTakenVar* | | *#Field* | | *#FieldByPWC* | |
|---|---|---|---|---|---|---|
| | Pkh | Dea | Pkh | Dea | Pkh | Dea |
| git-checkout | 135576 | 73967 | 121574 | 59965 | 68045 | 6436 |
| json-conversions | 62397 | 40993 | 40943 | 19539 | 22330 | 926 |
| json-ubjson | 60721 | 34987 | 49211 | 23477 | 27000 | 1266 |
| llvm-as-new | 24427 | 16124 | 19304 | 11001 | 9770 | 1467 |
| llvm-dwp | 145247 | 91945 | 109650 | 56348 | 62383 | 9081 |
| llvm-objdump | 16130 | 12007 | 11235 | 7112 | 5119 | 996 |
| opencv_perf_core | 60625 | 44061 | 40196 | 23632 | 18894 | 2330 |
| opencv_test_dnn | 53064 | 37957 | 35177 | 20070 | 17366 | 2259 |
| python | 30848 | 23713 | 21530 | 14395 | 9531 | 2396 |
| redis-server | 13109 | 9581 | 8165 | 4637 | 4234 | 706 |
| Xalan | 90314 | 62859 | 61466 | 34011 | 32226 | 4771 |
| *Max reduction* | 45.4% | | 52.3% | | 95.9% | |
| *Average reduction* | 32.4% | | 44.4% | | 86.6% | |

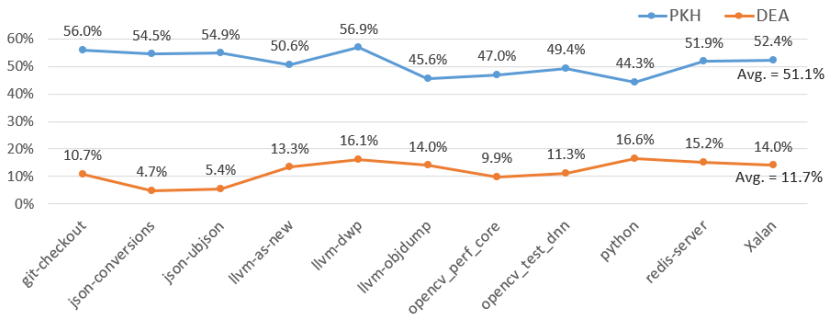



Fig. 9: Percentages of fields derived when solving $PWC$s out of the total number of fields, i.e., $\frac{\#FieldByPWC}{\#Field} * 100$

in C) and `json-ubjson` (written in C++) with heavy use of structs and classes, the percentages for both are higher than 80%. In 8 of the 11 programs, over 50% of the fields are derived from $PWC$s.

Columns 4-5 of Table 3 compare the total number of field objects produced by Pkh and Dea respectively. Columns 6-7 give more information about the number of fields derived only when resolving $PWC$s by Pkh and Dea, we can see that these fields are significantly reduced by Dea with an average reduction rate of 86.6%, demonstrating that Dea successfully captured the derivation equivalence to collapse a majority of fields into Sfrs when resolving $PWC$s.

Figure 9 further compares Dea with Pkh in terms of percentages of fields derived from resolving $PWC$s out of the total number of fields for the 11 pro-

Table 4: Constraint graph information (*#NodeInPWC* denotes the number of nodes involving *PWC*s by Pkh; *#SFR* denotes the number of stride-based field representatives, generated by Dea; *#CopyByPWC*, denotes the number of Copy edges flowing into and going out of fields derived when solving *PWC*s; *#CopyProcessed* denotes the number of processing times of Copy edges.)

| | *#NodeInPWC* | *#SFR* | *#CopyByPWC* | | *#CopyProcessed* | |
|---|---|---|---|---|---|---|
| | Pkh | Dea | Pkh | Dea | Pkh | Dea |
| git-checkout | 2840 | 2172 | 12372 | 2046 | 3868834 | 1128617 |
| json-conversions | 3631 | 1641 | 13490 | 2622 | 2253266 | 319960 |
| json-ubjson | 4271 | 1753 | 4311 | 1037 | 5621768 | 575884 |
| llvm-as-new | 1752 | 2085 | 9739 | 2789 | 2513940 | 688238 |
| llvm-dwp | 7263 | 1463 | 15062 | 2128 | 2802988 | 779424 |
| llvm-objdump | 1581 | 1761 | 7105 | 2013 | 2177990 | 647582 |
| opencv_perf_core | 1373 | 2030 | 4948 | 1973 | 4800563 | 655095 |
| opencv_test_dnn | 1007 | 777 | 4008 | 1577 | 5095795 | 460127 |
| python | 3817 | 1942 | 8530 | 3854 | 3495769 | 971376 |
| redis-server | 2783 | 1405 | 3380 | 1408 | 1288753 | 390783 |
| Xalan | 4874 | 2909 | 21935 | 7671 | 5143418 | 1554627 |
| *Max reduction* | | | 85.9% | | 91.0% | |
| *Avg. reduction* | | | 70.3% | | 77.3% | |

grams. The average percentage of 51.1% in Pkh (blue line) is reduced to only 11.7% (orange line) in Dea with a reduction of 39.4%.

In `git-checkout`, `json-conversions` and `json-ubjson`, Dea achieves over 90% reduction in solving *PWC*s because these programs have relatively large numbers of address-taken variables (Table 3) and relatively more nodes involving *PWC*s (Table 4). On average, over 85% of redundant field derivations involving *PWC*s are avoided with the maximum reduction rate of 95.9% in `json-conversions`, confirming the effectiveness of our field collapsing in handling *PWC*s.

Table 4 gives the constraint graph information after points-to resolution. Column 2 lists the number of nodes involving *PWC*s by Pkh. For each Sfr $\sigma$ generated by Dea, Column 3 gives the numbers of Sfrs generated by Dea. The average numbers of overlapping Sfrs for the 11 programs evaluated are all below 1, which means that the majority of the Sfrs either represent a single object/field or represent a sequence of fields that do not overlap with one another.

Columns 4-5 give the numbers of Copy edges flowing into and going out of field nodes derived when resolving *PWC*s by Pkh and Dea respectively. Dea on average reduces the Copy edges in Column 4 by 70.3% with a maximum reduction rate of 85.9% Columns 6-7 give the number of processing times of Copy edges during points-to propagation by the two approaches. Since the number of Copy edges is significantly reduced by Dea, the processing times of Copy edges are reduced accordingly with an average/maximum reduction rate of 77.3%/91.0%.

Table 5: Total analysis times and the times of the three analysis stages, including *CycleDec* cycle detection (Lines 7-9 of Algorithm 1), *PtsProp*, propagating points-to information via Copy and Field edges (Lines 11-24), *ProcessLdSt*, adding new Copy edges when processing Loads/Stores (Lines 25-36)

| | *CycleDec* | | *PtsProp* | | *ProcessLdSt* | | *TotalTime* | | *speed* |
|---|---|---|---|---|---|---|---|---|---|
| | PKH | DEA | PKH | DEA | PKH | DEA | PKH | DEA | *up* |
| git-checkout | 3117.8 | 4600.0 | 138233.5 | 26668.1 | 3870.2 | 1472.5 | 145221.6 | 32740.6 | 4.4 |
| json-conversions | 4436.2 | 561.6 | 12248.2 | 939.2 | 17.6 | 11.5 | 16702.0 | 1512.3 | 11.0 |
| json-ubjson | 25.1 | 6.0 | 18635.2 | 1817.3 | 52.4 | 23.2 | 18712.7 | 1846.6 | 10.1 |
| llvm-as-new | 22.6 | 11.9 | 10920.4 | 1728.9 | 541.9 | 221.2 | 11484.9 | 1962.0 | 5.9 |
| llvm-dwp | 3134.1 | 1457.7 | 120654.4 | 22177.2 | 1671.2 | 747.5 | 125459.8 | 24382.4 | 5.1 |
| llvm-objdump | 22.2 | 22.2 | 10617.3 | 2158.4 | 254.8 | 109.7 | 10894.4 | 2290.2 | 4.8 |
| opencv_perf_core | 338.5 | 299.3 | 30049.9 | 3018.5 | 2125.5 | 991.7 | 32513.9 | 4309.5 | 7.5 |
| opencv_test_dnn | 67.0 | 64.2 | 3145.5 | 248.8 | 366.1 | 122.2 | 3578.6 | 435.2 | 8.2 |
| python | 51.6 | 18.8 | 167556.9 | 22674.4 | 939.9 | 474.8 | 168548.3 | 23168.0 | 7.3 |
| redis-server | 525.1 | 428.6 | 11088.3 | 1315.2 | 99.8 | 49.8 | 11713.2 | 1793.5 | 6.5 |
| Xalan | 412.3 | 118.1 | 146617.8 | 21729.4 | 352.5 | 218.1 | 147382.7 | 22065.6 | 6.7 |
| *Average speedup* | | | | | | | | | 7.1 |

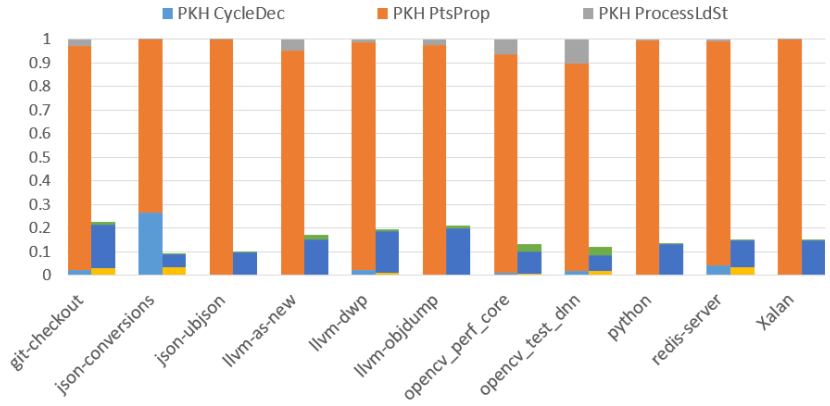



Fig. 10: Comparing the time distribution of the three analysis phases of DEA with that of PKH (normalized with PKH as the base).

Table 5 compares DEA with PKH in terms of the overall analysis times and the times collected for each of the three analysis phases. The total pointer analysis time consists of three major parts, as also discussed in Algorithm 1, and comprises (1) cycle detection, (2) propagating point-to sets via Copy and Field edges, and (3) processing Stores and Loads by adding new Copy edges into the constraint graph. Overall, DEA has a best speed up of 11.0X (observed in `json-conversions`) with an average speed up of 7.1X among the 11 programs.

Figure 10 gives the analysis time distributions of the three analysis phases in Table 5 for both PKH and DEA, where the phases are highlighted in different colors. The time cost of *PtsProp* (Columns 4-5) occupies a large percentage in resolution time by PKH. This is because *PtsProp* in field-sensitive pointer analysis needs to perform heavy set union operations for handling both Copy and Field edges. Worse, *PWC*s which need to be fully resolved by PKH incur a large number of redundant field derivations and unnecessary Copy edges until a pre-defined maximum number is reached, resulting in high analysis overhead in the *PtsProp* phase. In contrast, as depicted in Figure 10, the analysis overhead introduced by *PtsProp* is greatly reduced by DEA, though it occupies a noticeable portion of the total analysis time, showing that DEA effectively cuts down the overhead introduced by *PWC*s (i.e., redundant points-to propagation, and unnecessary Copy edges connecting to/from derivation equivalent fields) to help constraint resolution converge more quickly.

## 5 Related Work

Andersen's inclusion-based analysis [12] is one of the most commonly used pointer analyses. Resolving points-to relations in Andersen's analysis is formalized as a set-constraint problem by computing a dynamic transitive closure on top of the constraint graph of a program. The majority of works on Andersen's analysis for C/C++ programs are field-insensitive [29, 13, 1, 14, 5, 6]. Faehndrich et al. [29] introduced a partial online cycle elimination while processing complex constraints (e.g., Load/Store) and demonstrated that cycle detection is critical for scaling inclusion-based pointer analysis. Heintze and Tardieu [13] proposed a new field-based Andersen's analysis that can analyze large-scale programs with one million lines of code. Compared to field-sensitive analysis, field-based analysis imprecisely treats all instances of a field as one. For example, $o_1.f$ and $o_2.f$ are treated as one variable $f$, even if $o_1$ and $o_2$ are two different base objects allocated from different allocation sites.

To reduce the overhead of repeatedly finding cycles on the constraint graph during points-to resolution, *Lazy Cycle Detection* [5] triggers an SCC detection only when a visited Copy edge whose source and destination node have the same point-to information during points-to propagation. In addition to the online cycle elimination techniques, a number of preprocessing techniques, such as *Offline Variable Substitution* [30] and HVN [7], have also been proposed. The techniques explore pointer and location equivalence to reduce the size of the constraint graph for subsequent pointer analysis without losing any precision. *Hybrid Cycle Detection* [5] presented a hybrid cycle elimination algorithm by combing linear-time offline preprocessing with online cycle detection to further accelerate constraint resolution. Pereira et al. [6] proposed *Wave Propagation* by separating the constraint resolution of Andersen's analysis into three stages, i.e., collapsing of cycles, points-to propagation and insertion of new edges. The three phases are repeated until no more changes are detected in the constraint graph. The approach differentiates the existing (old) and new points-to information of a

pointer to reduce set union overhead on an acyclic constraint graph in topological order during points-to propagation.

Field-sensitive analysis distinguishes fields of a struct object improving its field-insensitive counterpart [31–34, 10, 11]. The challenges of field-sensitivity in for C/C++ is that the address of a field can be taken, stored to some pointer and later read at an arbitrary load. To tackle this challenge, Pearce et al. [10] proposes PKH, a representative field-sensitive analysis by employing a field-index-based abstraction modeling in which the fields of an object are distinguished using unique indices. The Andersen's constraint graph is extended by adding a new Field constraint to model address-of-field instructions for deriving fields during constraint resolution. Miné [34] presented a field- and array-sensitive analysis that translates field and array accesses to pointer arithmetic in the abstract interpretation framework. LPA [35] presented a loop-oriented pointer analysis for automatic SIMD vectorization. DSA [31] supports field-sensitivity using byte offsets object modeling, however, the approach is based on Steensgarrds unification-based analysis, using a coarser abstract object/points-to than Andersen's analysis.

CCLYZER [11] presents a precision enhancement approach to Pearce's field-sensitive analysis (PKH) by lazily inferring the types of heap objects by leveraging the type casting information to filter out spurious field derivations. CCLYZER improves the precision of PKH in the presence of factory methods and heap allocation wrappers in a program, achieving the heap cloning results without explicit context-sensitivity, but at the expense of more analysis time since an order of magnitude more type-augmented objects are introduced into the analysis. Rather than sacrificing performance to enhance analysis precision, DEA maintains the same precision as PKH, but significantly reduces its analysis overhead by fast and precise handling of positive weight cycles, a key challenge in field-insensitive pointer analysis. Our approach is also complementary to other cycle elimination resolution algorithms and fits well into existing constraint resolution frameworks for Andersen's analysis.

## 6 Conclusion

This paper presents a fast and precise handling of positive weight cycles to significantly boost the existing field-sensitive Andersen's analysis by capturing derivation equivalence. A new stride-based field abstraction is proposed to represent a sequence of derivation equivalent fields when resolving $PWC$s. DEA has been implemented in LLVM-7.0.0 and evaluated using 11 real-world large C/C++ programs. The evaluation results show that DEA on average is 7.1X faster than Pearce et al.'s field-sensitive analysis with the best speedup of 11.0X.

## 7 Acknowledge

We would like to thank the anonymous reviewers for their helpful comments. This research is supported by Australian Research Grant DE170101081.

## References


1. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03*, volume 38, pages 103–114. ACM, 2003.
2. Atanas Rountev, Ana Milanova, and Barbara G Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01*, volume 36, pages 43–55. ACM, 2001.
3. Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *TOSEM '02*, 27(4):1–11, 2002.
4. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09*, volume 44, pages 243–262. ACM, 2009.
5. Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07*, volume 42, pages 290–299. ACM, 2007.
6. Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135. IEEE, 2009.
7. Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS '07*, pages 265–280. Springer, 2007.
8. Sam Blackshear, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of Andersens analysis in practice. In *SAS '11*, pages 60–76. Springer, 2011.
9. Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for C. In *OOPSLA '14*, volume 49, pages 829–845. ACM, 2014.
10. David J Pearce, Paul HJ Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *TOPLAS*, 30(1):4, 2007.
11. George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *SAS '16*, pages 84–104. Springer, 2016.
12. Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.
13. Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *PLDI '01*, volume 36, pages 254–263. ACM, 2001.
14. David J Pearce, Paul HJ Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 3–12. IEEE, 2003.
15. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16, 2011.
16. 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.
17. Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In *SAS '14*, pages 319–336. 2014.
18. Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *FSE '16*, pages 460–473. ACM, 2016.
19. ISO90. ISO/IEC. international standard ISO/IEC 9899, programming languages - C. 1990.
20. Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.
21. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

22. Yulei Sui and Jingling Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266, 2016.
23. Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *TSE '14*, 40(2):107–122, 2014.
24. Implementing next generation points-to in open64. `www.affinic.com/documents/open64workshop/2010/slides/8_Ravindran.ppt`.
25. Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, volume 44, pages 226–238. ACM, 2009.
26. Side-effects of external apis. `https://github.com/SVF-tools/SVF/blob/master/lib/Util/ExtAPI.cpp`.
27. Clang-7.0.0. `releases.llvm.org/7.0.0/cfe-7.0.0.src.tar.xz`.
28. Whole-program llvm. `github.com/travitch/whole-program-llvm`.
29. Manuel Fähndrich, Jeffrey S Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98*, volume 33, pages 85–96. ACM, 1998.
30. Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI '00*, volume 35, pages 47–56. ACM, 2000.
31. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI'07*, volume 42, pages 278–289. ACM, 2007.
32. Dzintars Avots, Michael Dalton, V Benjamin Livshits, and Monica S Lam. Improving software security with a C pointer analysis. In *ICSE '05*, pages 332–341. ACM, 2005.
33. Erik M Nystrom, Hong-Seok Kim, and Wen-mei W Hwu. Importance of heap specialization in pointer analysis. In *PASTE '04*, pages 43–48. ACM, 2004.
34. Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES '06*, volume 41, pages 54–63. ACM, 2006.
35. Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue. Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization. In *LCTES '16*, pages 41–51. ACM, 2016.