Correlated Aliasing Strong Updates

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

Strong updates are to enable flow sensitive analysis to kill old pointer information when a variable is assigned new information. Strong update for top-level pointers in C are trivial in SSA form. For indirect strong updates, a.k.a, strong update for address-taken variables, previous approaches consider singleton, which means information can only be killed when points-to set of the dereferenced pointer containing at most one memory object. In this paper, we propose a novel approach to explore correlated aliasing on *memory regions* for strong updates. (**NEEDS A REWRITE**)

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords keyword1, keyword2

1. Introduction

Pointer analysis is to statically approximate runtime value of a pointer in memory. It is a key analysis for both compiler optimizations [5, 11, 14] and memory-related software error detection [7, 13, 17, 23]. Analysis with high precision can give more opportunities for aggressive optimizations and accurate bug detection. As clients increasingly require high precision, there has been quite a few recent advanced techniques for more precise pointer analysis [6, 9, 10, 12, 15, 24, 25]. Strong updates are the most important feature in flow sensitive pointer analysis for precision improvement [9, 10, 15, 25]. It can also be used to incorporate with context sensitive analysis to achieve further accuracy [6, 7]. Strong updates enable the analysis to kill the old points-to information of a memory location when this location is assigned a new value. In contrast, weak updates keep the old information unchanged while adding new values.

An abstract location v at a program statement can be strongly updated if and only if v refers to at most one concrete memory location. It is trivial to enable strong updates for *top-level* pointers on SSA (Static Single Assignment) form [9, 10, 24, 25]. Because of the nature that each variable can only have one definition on SSA, *top-level* pointers whose address are not taken can be directly renamed just like scalars. As a result, a pointer defined at different program points can be easily distinguished with unique names. However, the difficulty lies at analyzing *address-taken* pointers, given an address-taken pointer v pointed by x, accordingly, v can be indirectly accessed by x, strong updates for variable v at a *store* statement $*x=y$ can not be fully processed until the points-to relation of x is computed. Previous algorithms [8–10, 15, 22, 24, 25] handle *indirect* strong updates at *store* statements in terms of *singleton*, provided that the points-to set of a dereferenced pointer contains at most one concrete memory location. More specifically, the old value of v can be killed, and updated to the value of y after the statement if and only if $*x$ refers to the unique location v in any execution paths.

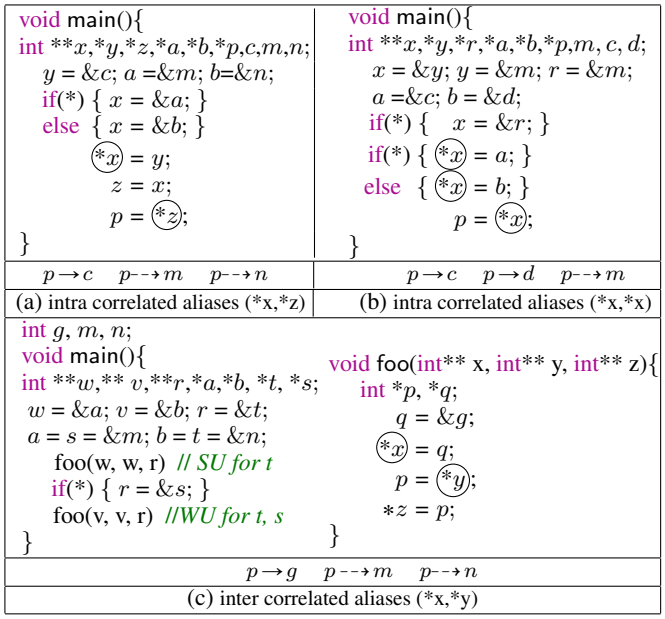

Figure 1. Strong updates with correlated aliases. Spurious points-to relations are labeled with dashed arrows (--→) according to *Singleton* based strong updates algorithm.

Singleton algorithm performs weak updates whenever a dereferenced pointer contains more than one concrete object. An example is shown in Figure 1(a), x points to two objects a and b from different branches at the entry of statement $*x=y$. Thus, a and b can only be weakly updated after analyzing the fourth statement, the old points-to target m is kept while adding the new value c into the points-to set of a. Likewise, both n and c are pointed by b. *Singleton* algorithm considers indirect strong updates at a *store* statement independently, while overlooks its correlations with another pointer dereference (e.g. dereference at *load* statements), which leads to weakly updated information at a *store* to be propagated unconditionally to any aliased expression. For example, Figure 1(a) shows that weakly updated points-to value of a and b at the fourth statement are assigned to pointer pointer p when handling *load* operation at statement $p=*z$. However, in real runtime execution scenar-

ISMM '13 June 20-21, Seattle Washington

ios, two aliases *x and *z always refer to exactly the same memory location in any program path. Therefore, strong updates occurs when considering the value flows between these two aliased expressions. Consequently, m and n in the points-to set of a and b are blocked for propagation from *x to *z, and only points-to target c of y can flow through this pair of aliases to p. (Similarly for Figure 1(b)(c))

Comparing to *Singleton* based algorithms which focus only on *store* statements for *indirect* strong updates, our approach performs strong updates beyond single memory location limitation by looking at a pair of aliased expressions (e.g. $\langle x, *y\rangle, \langle *x, *y\rangle, \langle *x, *x\rangle$). The key insight behind our approach is to consider points-to propagation between a *correlated aliases* pair instead of counting the number of memory locations at a pointer dereference alone. Strongly updated information can flow from a definition into its uses between *correlated aliases*. More strong updates can be obtained comparing with conventional *Singleton* approach.

Capturing correlations between a pair of aliases is challenging. First, points-to and aliasing analysis are cyclic dependent. On the one hand, for the sake of computing points-to relations by strong updates, correlated aliasing needs to be figured out. On the other hand, in order to decide whether a pair of aliased expressions are correlated (e.g.$\langle x, *y\rangle$), the points-to value of a dereferenced pointer y requires to be computed first. The naive approach is to use a cheap/imprecise analysis [1, 20] to distinguish aliases before strong updates. However, it might be too conservative to generate overwhelming imprecise aliases for correlation analysis. Ideally, correlated aliasing strong updates should be done at the same time when performing points-to resolution. Our way is to use a level-wised approach to partition pointer-related expressions into different levels, higher levels are analyzed prior to lower levels. With well-define processing order, earlier analysis can be used to bootstrap later phase analysis to generate precise results.

In addition, the precision of aliasing analysis is impacted by diverse factors (e.g. flow, context sensitivity), another challenge is how to effectively identify various correlation patterns when taking these multidimensional factors into account. By making use of flow sensitive analysis, the correlation connection between *x and *z is captured by assignment at statement z=x in Figure 1(a). However, correlations between two aliases are not limited to flow sensitivity. In practice, many aliases are correlated when considering their calling contexts and control-flow paths. Essentially, efficient and accurate algorithms which can easily leverage recent advanced analysis are crucial and attractive.

The following are the key contributions in this paper:

- A novel strong updates approach by considering the correlation between a pair of aliases for generating more precise strong updates than *Singleton* based algorithms
- Correlated aliasing analysis by considering flow, context and program paths information.
- A new level-wised memory region-based sparse analysis.
- We evaluate our *correlated aliasing* strong updates by comparing with *singleton* based algorithms and various correlation factors (e.g. context-, path-) with XXX benchmarks and XXX open source programs, our approach can have XXX more strong updates while keep the compatible analysis time XXX with state-of-art FSCS points-to analysis algorithms.

2. Language And Preliminaries

A simple C-like language abstract syntax is given in Figure 2 for our analysis. In the canonical form, a statement in the CFG of an input program is one of the following: (1) an assignment of the form, $p = \&q$ (`address`), $p = q$ (`copy`), $*p = q$ (`store`) or $p = *q$ (`load`), where p and q are local, global or heap variables, (2) a call $v = \mathsf{call}_k(p, \ldots, q)$ at callsite k, where $v, p, \ldots, q$ are all local variables, and (3) a return statement, $\mathsf{return}_t(v)$ at return site t, where v is a local variable. (4) a two-way branch (i.e., an if-statement), which is missing in Figure 2 will be considered after translating the program into SSA representation in Figure 3.

Local	l		
Global	g		
Heap	o		
Location	v, p, q	::=	$l \mid g \mid o$
Stmt	s	::=	$p = \&q \mid p = q \mid p = *q \mid p = q$
			$\mid v = \mathsf{call}_k(p, \ldots q) \mid \mathsf{return}_t(v)$
Procedure	f	::=	$f \cup \{s\} \mid \varnothing$

Figure 2. Abstract syntax of programs.

Generally, pointer analysis with strong updates applies iterative data-flow approach for resolution. Each statement s on the control flow graph (CFG) is analyzed with input points-to relations propagated from predecessors of s and outgoing points-to value are generated after analyzing s. Accordingly, the output points-to relations are then propagated to successors of s along the CFG for further computation. The propagation and resolution terminate until a fixed point is reached. However, iterative data-flow analysis is costly, as it propagates pointer information blindly from each node in the CFG to its successors without knowing if the information will be used there or not, which incurs huge analysis overhead and memory consumption especially when analyzing large programs [10]. On the contrary, a sparse analysis [9, 10, 15, 19, 21, 24] propagates the pointer information from defs of the variables directly to their uses along their def-use chains, but, unfortunately, the def-use information can only be computed using the pointer information. To break the cycle, a sparse pointer analysis typically proceeds in stages: def-use chains are initially approximated based on some fast pointer analysis and then refined in a sparse manner.

Sparse points-to analysis on SSA form can significantly speed up analysis for large programs [9, 10, 15, 21, 24]. However, it can not simply be adopted for our correlated aliasing strong updates. Several challenges need to be overcome: different from eariler sparse points-to analysis [9, 10, 24] which only calculate points-to result of a variable, our approach needs to analyze correlations between two aliased expressions simultaneously when doing pointer resolution. Therefore, both pointers (e.g. p) and dereference expressions (e.g. *p) are required to be sparsely represented on SSA. During our full-sparse SSA construction, a memory region R is introduced (Figure 3) to stands for an expression (e.g. $p_i, *p_i$), which represents a set of memory locations. For brevity, given a single variable expression p, $R(p)$ and p are used interchangeably to stand for the region. Statements are transformed into region-based statements as shown in Figure 3. Unlike previous sparse SSA form [9, 24], we put regions instead of individual variables on SSA for renaming. Each region is defined exactly once in the program, distinct definitions of a region are distinctly versioned.

There are many ways for creating a memory region, in one extreme, every expression like *p in the program can be treated as a region. However, p might be redefined to different versions like p_i and p_j at distinct program points on SSA, thereby referring *p_i and *p_j to the same memory region is imprecise. In another extreme, in order to create an exact region for *p, pointer p needs to be SSA renamed and fully solved first before region creation of *p. We arrange a well-defined processing order for analyzing different expressions so as to create precise regions for incrementally building full-sparse SSA form (Section 3).

During construction of SSA for all regions, three new types of operators Φ, μ, and χ are introduced following [3]. At a join point

Region	$R, R', R^\bullet$	$::= p \mid *p \mid \&p$
Region	s	$::= \langle R \overset{s}{\leftarrow} R^\bullet \rangle_\delta$
Based Stmt		$\mid \langle R \overset{s}{\leftarrow} \Phi(R,R) \rangle$
		$\mid \langle R \overset{s}{\leftarrow} \mathsf{call}_k(R', \ldots, R^\bullet) \rangle^\theta_\delta$
		$\mid \langle \mathsf{return}_t(R) \rangle^\theta$
Condition	C, P, M	$::= true \mid false \mid \neg C$
		$\mid C_1 \vee C_2 \mid C_1 \wedge C_2$
May-Use	θ	$::= \{\mu(R, C, P, M)\} \mid \theta \cup \theta \mid \varnothing$
May-Def	δ	$::= \{R = \chi(R, C, P, M)\} \mid \delta \cup \delta \mid \varnothing$

Figure 3. Extended abstract syntax of memory region based representation on full-spasre SSA form (a subscript associated with each region which stands for SSA version is ignored for brevity here).

in the control-flow graph (CFG) of the program, all versions of the same region reaching the point are combined using a Φ function, producing a new version for the region. Different from [3], address, copy, stores and loads (Figure 2) are all unified into a simple *region copy* statement $\langle R \overset{s}{\leftarrow} R^\bullet \rangle_\delta$ (Figure 3). Only χ functions instead of μ is annotated for each *region copy* statement s. May-Def list δ, which is a list of χ functions, such that every $R' = \chi(R')$ denotes a region R' is aliased with R and may be potentially defined at s. χ function is treated as a definition, suppose $R' = \chi(R')$ becomes $R'_m = \chi(R'_n)$ after SSA conversion. R'_m can be strongly updated if R' *must alias* with R, In this case, R'_m receives whatever $R^\bullet$ points to and the information in R'_n is ignored. Otherwise, R'_m must incorporate the pointer information from both R'_n and $R^\bullet$. May-Use list θ, which is a list of μ functions annotated only at a callsite or return statement s, such that every $\mu(R)$ denotes a region R that may be potentially read at s. Every callsite is annotated with a May-Use list θ and a May-Def list δ. Every return statement is annotated with a May-Use list θ to represent the list of regions that may indirectly return to a caller.

Each μ/χ are associated with condition C, P and M to specify under which condition the indirect read/modification holds. C stands for *context condition* represented as Boolean expressions over the set of points-to relations to be introduced in Section 3.2. P is a set of intra-procedure *path condition* represented as Boolean expressions encoding the branches in a CFG to be introduced in Section 3.1. In order to perform strong updates, M is a special *must condition* to represent under which a region in the μ/χ must be used/modified (*must alias*). It uses the calling contexts encoded similarly as C when performing inter-procedure analysis (Section 3.2). For intra-procedure analysis, its value is either true or false (Section 3.1).

Once indirect defs and uses at loads, stores and callsites are exposed by adding μ and χ functions and then the conversion to SSA can take place using a standard SSA algorithm [3, 4].

3. Correlated Aliasing Strong Updates

Two aliased regions are correlated whenever they can be represented by the same memory object along identical program path during runtime. Aliasing and points-to analysis are cyclic dependent. In order to break the cycle, a pre-analysis is used by Steensgaard algorithm [20] to bootstrap later phase analysis. Given an unification-based points-to graph with all nodes represented by regions, all predecessors of each node are merged, and finally, make the points-to graph acyclic by collapsing SCCs. The points-to level of a region is its longest length over $\{0, \ldots, \mathrm{L}\}$ to a sink node. Regions at higher levels are analyzed first so as to provide points-to information for analyzing regions at lower levels.

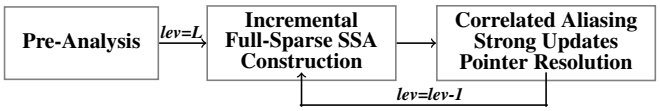

Figure 4. Overview of the Algorithms.

DEFINITION 1. *Processing Order: given two regions R' and R on SCC-unification points-to graph, $R' \sqsupset R$ denotes R' is analyzed before R, $R' \triangleq R$ stands for R' is analyzed together R.*

According to *Processing Order*, our analysis is partitioned into two parts as shown in Figure 4 : incremental SSA construction by inserting μ/χ functions to identify correlations and pointer resolution for strong updates with correlated aliasing constraints. These two parts are processed iteratively by analyzing regions from highest to lowest level.

DEFINITION 2. *Points-to Set and Aliasing Set: For a region R, $(v, C, P, M) \in \mathsf{Ptr}(R)$ contains a set of tuples representing that R points to location v under context C along path P, the points-to must holds under condition M. Similarly, $(v, C, P, M) \in \mathsf{AS}(R)$ denotes location v is represented by R under context C along path P, v is definitely represented by R if and only if M holds.*

Conceptually, aliasing set and points-to set has the same tuples, but they are used in different stages, points-to set is used during pointer resolution, While aliasing set is used to assist inserting μ/χ function during full-sparse SSA construction to mark correlations. If a region R stands for address expression like $\&p$, its $\mathsf{AS}(R) is \varnothing$ in default, if a region R stands for a single concrete variable expression like p, it is understandable that $\mathsf{AS}(R)$ is the memory location itself with all conditions true $\mathsf{AS}(R) = \{(p, \mathsf{true}, \mathsf{true}, \mathsf{true})\}$. Aliasing and points-to sets are mutually dependent. On the one hand, aliasing set of a region should be calculated before SSA conversion and pointer resolution. On the other hand, aliasing set of a region at level lev depends on points-to set of regions at the levels no less than lev. For example, when analzying a region $R(*p)$ at lev at a statement s, its aliasing set depends on points-to set of $lev + 1$ region p after p is fully solved and renamed to a SSA variable p_i at s, thus, $AS(R(*p_i)) = \mathsf{Ptr}(p_i)$. Likewise, SSA conversion and points-to resolution of $R(*p_i)$ is carried out later to generate the input for analyzing regions at $lev - 1$.

Correlation between two aliases is affected by analysis in several different dimensions. In the following sections, we first describe the intra-procedural correlated aliasing strong updates algorithms by considering flow sensitivity and program path information (Section 3.1). Then we progressively move to capture the inter-procedural correlations when taking calling context into account (Section 3.2).

3.1 Intra-procedural Analysis

In this section, we introduce how to perform correlated aliasing strong updates within a procedure by following the stages in Figure 4. As we concern about intra-procedure analysis in this section, only conditions P and M are introduced in μ/χ and points-to set, the computation of condition C is demonstrated in Section 3.2. Our path condition P is encoded into BDDs like [21, 23]. All branch nodes on CFG have two successors in our IR. For each branch node, a new guard P is allocated to express program paths. The edges and blocks in a CFG (with cycles collapsed) are associated with paths as follows. The path for the incoming edge of the entry block is initialized to be true (representing the set of all paths). Let BB be a block with n incoming edges associated with paths $P_1, \ldots, P_n$. The path for BB is $P_1 \vee \cdots \vee P_n$. If BB is a branch

1. $y = \&c$; $a = \&m$; $b = \&n$;		1. $y_0 = \&c$; $a_0 = \&m$; $b_0 = \&n$;	
2. if(*) $x_0 = \&a$; // path P_1		2. if(*) $x_0 = \&a$; // path P_1	$\mathsf{Ptr}(y_0) = \{(c,\text{true},\text{true})\}$
3. else $x_1 = \&b$; // path $\neg P_1$		3. else $x_1 = \&b$; // path $\neg P_1$	$\mathsf{Ptr}(a_0) = \{(m,\text{true},\text{true})\}$
4. $x_2 = \Phi(x_0, x_1)$	$\mathsf{Ptr}(x_0) = \{(a,P_1,\text{false})\}$	4. $x_2 = \Phi(x_0, x_1)$	$\mathsf{Ptr}(b_0) = \{(n,\text{true},\text{true})\}$
5. $*x_2 = y$;	$\mathsf{Ptr}(x_1) = \{(b,\neg P_1,\text{false})\}$	5. $R(*x_2)_0 = y_0$;	$\mathsf{Ptr}(a_1) = \{(m,\neg P_1,\text{false}),(c,P_1,\text{false})\}$
6.	$\mathsf{Ptr}(z_0) = \{(a,P_1,\text{false}),(b,\neg P_1,\text{false})\}$	6. $a_1 = \chi(a_0, P_1, \text{false})$	$\mathsf{Ptr}(b_1) = \{(n,P_1,\text{false}),(c,\neg P_1,\text{false})\}$
7.	$\mathsf{Ptr}(x_2) = \{(a,P_1,\text{false}),(b,\neg P_1,\text{false})\}$	7. $b_1 = \chi(b_0, \neg P_1, \text{false})$	$\mathsf{Ptr}(R(*x_2)_0) = \{(c,\text{true},\text{true})\}$
8.		8. $R(*z_0)_0 = \chi(R(*z_0)_0, \text{true}, \text{true})$	$\mathsf{Ptr}(R(*z_0)_0) = \{(c,\text{true},\text{true})\}$
9. $z_0 = x_2$;		9. $z_0 = x_3$;	$\mathsf{Ptr}(p_0) = \{(c,\text{true},\text{true})\}$
10. $p = *z_1$;		10. $p_0 = R(*z_0)_0$;	
(a) Full-sparse SSA	(b) Pointer Resolution	(c) Full-sparse SSA	(d) Pointer Resolution
Processing 2nd Level regions : x, z, R(&a),R(&b)		Processing 1st Level regions : y, p, a, b, R(*x), R(*z)	

Figure 5. An intra-procedure strong updates by identifying *correlated aliases*, (region $R(*x_3)$ and $R(*z_1)$). To avoid cluttering, points-to results in Figure 5(b) are also used to generate points-to when processing 1st level regions, but not shown in Figure 5(d).

$$\delta = \bigcup_{R' \in \Omega} (\chi(R', P_\chi, M_\chi)) \quad \text{if } \mathsf{ComPair}(R, R') \neq \varnothing \tag{1}$$

$$P_\chi = \bigvee_{\substack{((v,P_i,M_i),(v,P_j,M_j)) \\ \in \mathsf{ComPair}(R,R')}} (P_i \wedge P_j) \tag{2}$$

$$M_\chi = \begin{cases} \text{true} & \text{if } \mathsf{AS}(R) \equiv \mathsf{AS}(R') \\ \bigvee_{\substack{((v,P_i,M_i),(v,P_j,M_j)) \\ \in \mathsf{ComPair}(R,R')}} (M_i \wedge M_j) & \text{if } \mathsf{AS}(R) \not\equiv \mathsf{AS}(R') \end{cases} \tag{3}$$

Figure 6. Rules for inserting χ functions into δ list for a region assignment $\langle R \overset{s}{\hookleftarrow} R^\bullet \rangle_\delta$, ($P_s$ is the path condition at s, Ω stands for all regions in the procedure).

node encoded with P_b, the paths for its two outgoing edges are $(P_1 \vee \cdots \vee P_n) \wedge P_b$ and $(P_1 \vee \cdots \vee P_n) \wedge \neg P_b$, respectively. Otherwise, its unique outgoing edge is $P_1 \vee \cdots \vee P_n$.

DEFINITION 3. *Common Aliasing pairs: Given two regions R and R', $\mathsf{ComPair}(R,R')$ contains a set of pairs. Each pair has two elements $[(v, C_i, P_i, M_i) \in \mathsf{AS}(R), (v, C_j, P_j, M_j) \in \mathsf{AS}(R)]$ sharing the common memory location v.*

DEFINITION 4. *Equivalent Aliasing Sets: If two aliasing sets are equivalent $\mathsf{AS}(R) \equiv \mathsf{AS}(R')$, all elements in $\mathsf{AS}(R)$ and $\mathsf{AS}(R')$ are in $\mathsf{ComPair}(R,R')$, for each pair $[(v, C_i, P_i, M_i), (v, C_j, P_j, M_j)]$, satisfy $C_i = C_j, P_i = P_j, M_i = M_j \neq \text{false}$.*

If two aliasing sets $\mathsf{AS}(R)$ and $\mathsf{AS}(R')$ are equivalent, then R is always must aliased with R' during any program execution path. Given a context and path condition, both R and R' can be represented by the same memory location. Strongly updated information on one region is used directly in another one.

EXAMPLE 1. *Let us revisit the example in Figure 1(a) to go through our intra-procedural algorithms (Figure 6 and Figure 7) for correlated aliasing strong updates in Figure 5. Context condition C is assumed to be* true *and ignored in μ/χ and points-to set in the following paragraphs.*

After pre-analysis, the regions are partitioned into three levels: $\{x, z, R(\&a), R(\&b)\}$ are at level 2, and $\{y, p, a, b, R(*x), R(*z)\}$ are at level 1, and $\{m, n, c\}$ are at level 0.

Then, we start processing the regions at highest level by following the two stages in Figure 4. Regions at second level are all single variables which are not address-taken, their values can not be indirectly modified. Therefore, no μ/χ functions annotated at the statements as they do not alias with any other regions. SSA conversion is directly performed as shown in Figure 5. Next, pointer resolution starts by following the rules in Figure 7. Regions $R(\&a), R(\&b)$ are first initialized (`[INI]`). and then their values are passed into x_1 and x_2 through assignments at line 2 and 3 along path P_1 and $\neg P_1$ following rule `[CPY]`. δ list annotated at the assignments are empty, and there is no side-effects on them. The strong updates for x_0,x_1,z_0 are performed straightforward with points-to results shown in Figure 5(b). x_2 has both the value of the previous two versions x_0 and x_1 through a Φ function standing at the join points after two branches at line 4 (`[PHI]`) as shown in figure 5(b).

After values of regions at second level are solved, we start a new round for analyzing first level regions. Firstly, we need to insert χ functions for building SSA. Given the points-to sets of x_2 and z_0

$$\texttt{[INI]}\ \frac{R_i \text{ represents } \&q}{\mathsf{Ptr}(R_i) = \{(q, \text{true}, \text{true})\}}$$

$$\texttt{[CPY]}\ \frac{\langle R_i \overset{s}{\hookleftarrow} R_j^\bullet \rangle_\varnothing \rhd \mathsf{Ptr}\lfloor \underline{s} \rfloor}{\mathsf{Ptr}(R_i) = \mathsf{Ptr}(R_j^\bullet) \times P_s}$$

$$\texttt{[AWU]}\ \frac{\begin{array}{c}\langle R_i \overset{s}{\hookleftarrow} R_j^\bullet \rangle_\delta \rhd \mathsf{Ptr}\lfloor \underline{s} \rfloor \\ (R_t' = \chi(R_k', P_\chi, M_\chi)) \in \delta \qquad M_\chi \neq \text{true} \\ (v_j, P_j, M_j) \in \mathsf{Ptr}(R_j^\bullet) \qquad (v_k, P_k, M_k) \in \mathsf{Ptr}(R_k')\end{array}}{\begin{array}{l}\mathsf{Ptr}(R_i) = \mathsf{Ptr}(R_j^\bullet) \times P_s \\ \mathsf{Ptr}(R_t') \cup= \{(v_j, P_\chi \wedge P_j \wedge P_s, \text{false})\} \\ \mathsf{Ptr}(R_t') \cup= \{(v_k, \neg P_\chi \wedge P_k \wedge P_s, \text{false})\}\end{array}}$$

$$\texttt{[ASU]}\ \frac{\begin{array}{c}\langle R_i \overset{s}{\hookleftarrow} R_j^\bullet \rangle_\delta \rhd \mathsf{Ptr}\lfloor \underline{s} \rfloor \\ (R_t' = \chi(R_k', P_\chi, M_\chi)) \in \delta \qquad M_\chi = \text{true}\end{array}}{\mathsf{Ptr}(R_i) = \mathsf{Ptr}(R_j^\bullet) \times P_s \quad \mathsf{Ptr}(R_t') = \mathsf{Ptr}(R_j^\bullet) \times P_s}$$

$$\texttt{[PHI]}\ \frac{\begin{array}{c}\langle R_i \overset{s}{\hookleftarrow} \Phi(R_j, R_k) \rangle \rhd \mathsf{Ptr}\lfloor \underline{s} \rfloor \\ (v_j, P_j, M_j) \in \mathsf{Ptr}(R_j) \qquad (v_k, P_k, M_k) \in \mathsf{Ptr}(R_k)\end{array}}{\begin{array}{l}\mathsf{Ptr}(R_i) \cup= \{(v_j, P_j \wedge P_s, \text{false})\} \\ \mathsf{Ptr}(R_i) \cup= \{(v_k, P_k \wedge P_s, \text{false})\}\end{array}}$$

Figure 7. Rules for pointer resolution. ($\langle R_i \overset{s}{\hookleftarrow} R_j^\bullet \rangle_\delta \rhd \mathsf{Ptr}\lfloor \underline{s} \rfloor$ produces points-to relations after statement s. $\mathsf{Ptr}(R_i^\bullet) \times P_s = \{(v, P_j \wedge P_s, M_i) \mid (v, P_j, M_i) \in \mathsf{Ptr}(R_j^\bullet)\}$ denotes guarded points-to set assignment with path condition P_s at s)

already computed, two aliasing sets $\mathsf{AS}(R(*x_2)) = \mathsf{Ptr}(x_2)$ and $\mathsf{AS}(R(*z_0)) = \mathsf{Ptr}(z_0)$ reckons equivalent according to Definition 4. As $\mathsf{AS}(R(*x_2))$ have common aliases with $\mathsf{AS}(R(*z_0))$, $\mathsf{AS}(a)$ and $\mathsf{AS}(b)$, therefore three χ functions are annotated from line 6 to 8 for this $store$ statements according to Equation 4 in Figure 6. Although $\mathsf{AS}(R(*x_2)) \not\equiv \mathsf{AS}(a)$, they have one *common aliasing pair* $[(a, P, \mathsf{false}), (a, \mathsf{true}, \mathsf{true})]$, The path conditions $P_\chi = P_1$ and $M_\chi = \mathsf{false}$ for χ functions are easily obtained at line 6 and 7 in Figure 5(c) by applying Equation 2 and 3. As $\mathsf{AS}(R(*z_0)) \equiv \mathsf{AS}(R(*x_2)) = \{(a, P, \mathsf{false}), (b, \neg P, \mathsf{false})\}$, are equivalent aliasing sets, therefore true is added as *must* condition into χ functions at line 8 to signify region $R(*z_0)$ must be strongly updated here (Equation 3). And *path* condition is calculated to be $P_1 \vee \neg P_1 = \mathsf{true}$ based on Equation 2.

With the full-sparse SSA form, pointer resolution is carried out for first level regions by following the rules in Figure 6. According to rule [INI] and [CPY], points-to sets of a_0 and b_0 are easily achieved as shown in Figure 5(d). For statement at line 5, regions a and b in χ functions at line 6 and 7 are weakly updated because *must* condition generated in earlier round is false, therefore, a_1 has the points-to results coming from both its old value a_0 and y_0 from right hand side of the assignment. Following rule [AWU], a_1 points to m along path $\neg P_1$, and c along path P_1. Similarly, b_1 points to n along path $\neg P_1$ and points to c along P_1. As indicated by *must* condition, region $R(*z_0)$ is strongly updated to have the same value as region $R(*x_1)$, and later when analyzing statement at line 10, following rule [CPY], p_0 definitely points to c only.

3.2 Inter-procedural Analysis

Our inter-procedure analysis are performed in a two-pass manner by traversing on SCC Call graph (with recursion cycled collapsed) for each level of regions. In order to summary and apply the side-effects of a callee, full-sparse SSA construction and pointer resolution (Figure 4) are carried out during bottom-up traverse of the SCC call graph (Figure 11 and 13). Then an addtional top-down pass is performed right after bottom-up traverse so as to propagate points-to information downwards from callers to callees, further resolution is needed if any new points-to relation is discovered.

The principle behind our inter-procedure analysis is similar as many summary-based points-to analysis [2, 7, 16, 18, 22]. The variation is that we consider correlations among different calling contexts. In order to analyze program across function boundaries, two things need to be modeled: points-to information flows from a callsite at a caller to its callees; on the opposite side, the side-effect points-to information also flows from a callee back to callsites at callers. In order to propagate information among procedures, we have to handle inter-procedural mapping for regions. Explicitly, an actual parameter passes its value from callers to formal parameters at its callees, a return parameter returns its value back to a receive parameter at callsites. However, a region at caller can be used/modified indirectly at its callees via formal parameters. As a result, an analysis is needed for connecting these regions across procedures. We use a light-weight Mod-Ref analysis to determine regions which may represent locations that beyond the scope of current procedure (Figure 8), the Mod-Ref analysis is used before bottom-up and top-down analysis at the beginning of each round, so as to provide information for assisting μ/χ functions insertion at callsites. And the results of Mod-Ref analysis is used for mapping the regions for information propagation in Figure 9.

Given a function f, the Mod-Ref analysis determines the set $\mathcal{U}$ ($\mathcal{D}$) of memory locations that are not declared in f but that may be indirectly read (modified) in f, denoted as $\vdash f : \mathcal{U}, \mathcal{D}$. After SSA conversion, every region in $\mathcal{U}$ is version 0 while every region in $\mathcal{D}$ is maximum version of that region. During the analysis, the function body of f is naturally unfolded into a list (or set) of statements

$$\texttt{[I-PRO]}\ \frac{f : \mathcal{U}; \mathcal{D} \quad \widehat{s} = \mathcal{E}(f) \quad \vdash \widehat{s} : \widehat{\mathcal{U}}; \widehat{\mathcal{D}}}{\mathcal{U} = \bigcup_{\widehat{\mathcal{U}}} \quad \mathcal{D} = \bigcup_{\widehat{\mathcal{D}}}}$$

$$\texttt{[I-CAL]}\ \frac{\mathsf{call}_k : \mathcal{U}; \mathcal{D} \quad \widehat{f} = \Gamma(\mathsf{call}_k) \quad \vdash \widehat{f} : \widehat{\mathcal{U}}; \widehat{\mathcal{D}}}{\mathcal{U} = (\bigcup_{\widehat{\mathcal{U}}}) \cap \mathcal{G}_{\mathsf{call}_k} \ \mathcal{D} = (\bigcup_{\widehat{\mathcal{D}}}) \cap \mathcal{G}_{\mathsf{call}_k}}$$

$$\texttt{[I-ASN]}\ \frac{\begin{array}{l} R \overset{s}{\hookleftarrow} R' : \mathcal{U}; \mathcal{D} \\ (v, C_i, M_i) \in \mathsf{AS}(R) \quad v \text{ is nonlocal} \\ (v', C_j, M_j) \in \mathsf{AS}(R') \quad v' \text{ is nonlocal} \end{array}}{\mathcal{U} = \{R'\} \qquad \mathcal{D} = \{R\}}$$

$$\texttt{[I-RET]}\ \frac{\mathsf{return}_j(R) : \mathcal{U}; \mathcal{D}}{\mathcal{U} = \varnothing \qquad \mathcal{D} = \{R\}}$$

Figure 8. Rules used for Mod-Ref analysis.

$\widehat{s} = \mathcal{E}(f)$. Then the analysis is broken down for each statement, denoted as $s : \mathcal{U}, \mathcal{D}$. The rules used are given in Figure 8.

The root causes for the interprocedural side-effects in a function f are loads and stores. For a load $p = *q$, $\mathsf{Ptr}(q)$ may contain nonlocal locations read in f ([I-LD]). Similarly, [I-ST] collects nonlocal locations in $\mathsf{Ptr}(q)$ that may be modified at a store. In contrast, address statements and assignments do not contribute any side effects according to [I-ADD] and [I-ASN].

For a callsite call_k, the most conservative Mod-Ref analysis is to assume that the set of all variables passed into this callsite ($\mathcal{G}_{\mathsf{call}_k}$) may be read and modified by its callees invoked directly/indirectly. The notation $\Gamma(\mathsf{call}_k)$ stands for a mapping from call_k to a list of callees, $\widehat{f}$. This naive approach is inaccurate due to an overwhelmingly large number of unrealizable def-use chains created across the functions. Therefore, we perform a further analysis for the callees invoked at call_k to filter out spurious Mod-Ref information. In the presence of recursion, [I-CAL] and [I-PRO] are recursively applied until a fixed point is reached.

$$\mathcal{M}_f^{\mathsf{call_k}}(R) = \begin{cases} \bigcup_{\forall R' \in \mathcal{U}', \mathsf{ComPair}(R, R') \neq \varnothing} R' & \text{if } R \in \mathcal{U} \\ \bigcup_{\forall R' \in \mathcal{D}', \mathsf{ComPair}(R, R') \neq \varnothing} R' & \text{if } R \in \mathcal{D} \\ FP(f, i) \quad \text{if } R = AP(call_k, i) \quad i > 0 \\ RET(f) \quad \text{if } R = AP(call_k, i) \quad i = 0 \end{cases} \tag{4}$$

Figure 9. Inter-procedural mapping. For a callsite $\mathsf{call_k} : \mathcal{U}; \mathcal{D}$, $\mathcal{M}_f^{\mathsf{call_k}}(R)$ maps region R at call_k to a set of regions in one of its callees $f : \mathcal{U}'; \mathcal{D}'$. $AP(\mathsf{call_k}, i)$ stands for i th actual parameter at $call_k$, and $FP(f, i)$ represents i th formal parameter of f, when i = 0, it returns the receive parameter at $\mathsf{call_k}$.

$$\theta = \bigcup_{R \in \mathcal{U}} (\mu(R, C_\mu, M_\mu)) \qquad \text{if } \mathsf{Eval}(C_{\mathsf{call_k}}, C_i) = \mathsf{true} \tag{5}$$

$$\delta = \bigcup_{R \in \mathcal{D}} (R = \chi(R, C_\chi, M_\chi)) \quad \text{if } \mathsf{Eval}(C_{\mathsf{call_k}}, C_i) = \mathsf{true} \tag{6}$$

$$C_{\mu/\chi} = C_{\mathsf{call_k}} \tag{7}$$

$$M_{\mu/\chi} = \begin{cases} \mathsf{true} & \text{if } \mathsf{Eval}(C_{\mathsf{call_k}}, M_i) = \mathsf{true} \\ \mathsf{false} & \text{if } \mathsf{Eval}(C_{\mathsf{call_k}}, M_i) \neq \mathsf{true} \end{cases} \tag{8}$$

$$\text{Given } R' \in \mathcal{M}_f^{\mathsf{call_k}}(R) \quad (v, C_i, M_i) \in \mathsf{Ptr}(R')$$

Figure 11. Rules for inserting χ functions into μ/δ list at callsite $\langle \mathsf{call_k} \rangle_\delta^\theta$, given Mod-Ref $\mathsf{call_k} : \mathcal{U}; \mathcal{D}$ ($C_{\mathsf{call_k}}$ is context condition encoded by points-to relations at the callsite).

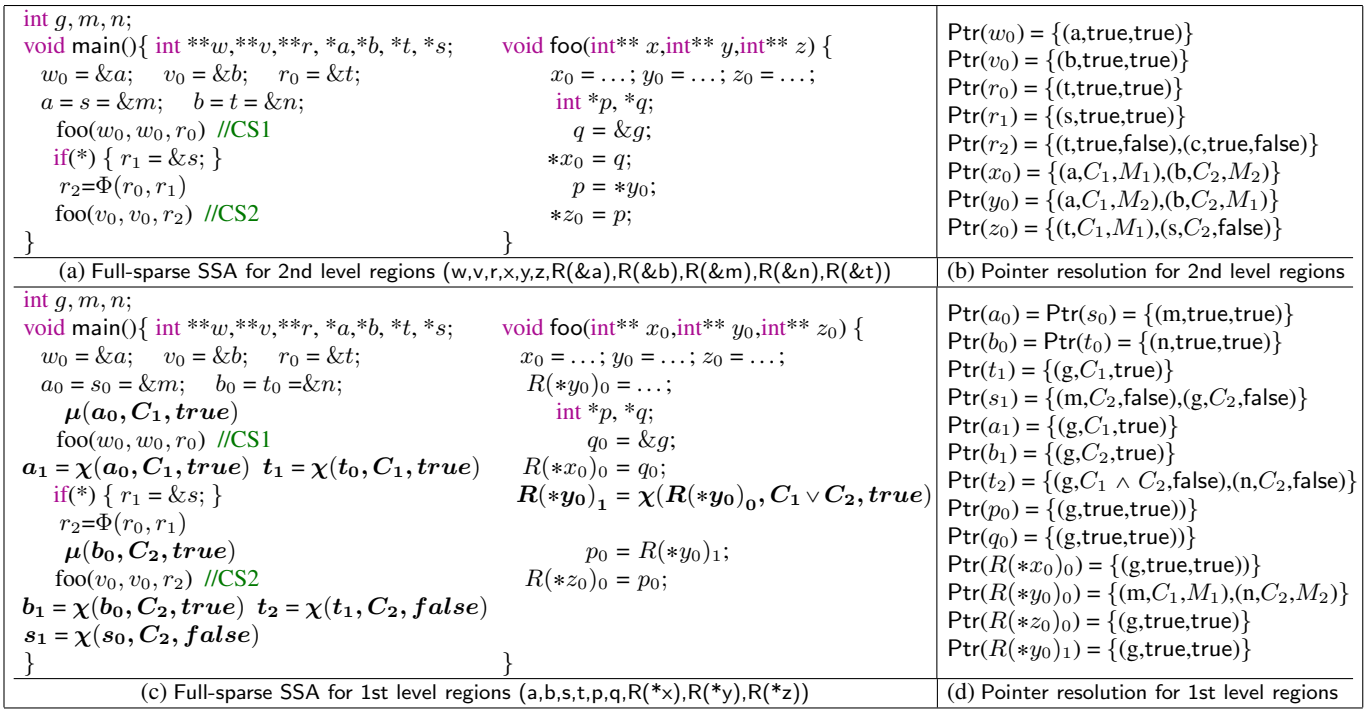

Figure 10. An inter-procedure strong updates example. To avoid cluttering, points-to results in Figure 10(b) are also used to generate points-to when processing 1st level regions, but not shown in Figure 10(d).

EXAMPLE 2. *Let us revisit our examples in Figure 1(c) to demonstrate our intra-procedure analysis in Figure 10.*

After pre-analysis, regions are partitioned in three levels and the lowest level regions are not pointers thus ignored for analysis, only 2nd and 1st level regions are presented for strong updates in Figure 10. Mod-Ref analysis for first level regions are unnecessary as they do not have side-effects at callsites. Then, we start building SSA (Figure 10(a)) and pointer resolution (Figure 10(b)) by bottom-up traverse of call graph from foo to main for top level regions. A top-down pass is continued after finishing resolution in main, the points-to information is propagated downwards via inter-procedure mapping (Figure 9). To be noted, given a procedure $f : \mathcal{U}; \mathcal{D}$, its formal parameters and regions in its Ref set $\mathcal{U}$ have a initial assignments at the procedure entry to stand for receiving values from callers. For example, $x_0 = \ldots$ denotes that formal parameter x_0 values from the two callsites in main. a is added into points-to set of x_0 and y_0 from callsites CS1 in main, while b is added into points-to set of x_0 and y_0 from callsites CS2. The points-to sets are shown in Figure 10(b), C_1 and C_2 here are used as context conditions encoded by points-to relations at callsites CS1 and CS2. M_1 has the same value of C_1 representing the fact that w_0 must point to a at the callsite CS1. M_2 has the similar meaning as M_1. Must condition M for a points-to target v is true at the procedure entry if and only if v definitely be pointed by at all callsites. t is added into points-to set z_0 under context condition C_1, and must hold this points-to at callsite CS1. s added into points-to set z_0 under C_1, but r_2 does not definitely point to s at CS2).

Next, a new round begins for analyzing 1st level regions. SSA construction (Figure 10(c)) and pointer resolution (Figure 10(d)) are performed from procedure foo to main. During SSA construction of foo, a χ function is inserted at statements $*x$=q. According to Definition 4, $R(*x_0) \equiv R(*y_0) = \{(a, C_1, M_1), (b, C_2, M_2)\}$,

$$\texttt{[PROP-1]}\ \frac{\langle \mathsf{call_k} \rangle^{\theta}_{\delta} \quad R \in AP(call_k) \quad f \in \Gamma(\mathsf{call_k}) \quad R' \in \mathcal{M}^{\mathsf{call_k}}_{f}(R) \quad (v, C_i, M_i) \in \mathsf{Ptr}(R)}{\mathsf{Ptr}(R') \cup= \{(v, C_i \wedge C_{call_k}, M_i \wedge M_{call_k})\}}$$

$$\texttt{[PROP-2]}\ \frac{\langle \mathsf{call_k} \rangle^{\theta}_{\delta} \quad \mu(R_i, C_{call_k}, M_{call_k}) \in \theta \quad f \in \Gamma(\mathsf{call_k}) \quad R' \in \mathcal{M}^{\mathsf{call_k}}_{f}(R) \quad (v, C_i, M_i) \in \mathsf{Ptr}(R)}{\mathsf{Ptr}(R') \cup= \{(v, C_i \wedge C_{call_k}, M_i \wedge M_{call_k})\}}$$

Figure 12. Rules for propagating points-to information from a callsite to callees ($C_{\mathsf{call_k}}$ is context condition at the callsite).

$$\texttt{[CALL-W]}\ \frac{\langle \mathsf{call}_k \rangle^{\theta}_{\delta} \rhd \mathsf{Ptr}[\underline{\mathsf{call}_k}] \quad (R_t = \chi(R_s, C_\chi, M_\chi)) \in \delta \quad R' \in \mathcal{M}^{\mathsf{call_k}}_{f}(R_t) \quad (v_t, C_t, M_t) \in \mathsf{Ptr}(R') \quad \mathsf{Eval}(C_\chi, C_t) = \mathsf{true} \quad M_\chi \neq \mathsf{true}}{\mathsf{Ptr}(R_t) \cup= \{(v_t, C_\chi, M_\chi)\} \quad \mathsf{Ptr}(R_t) \cup= \mathsf{Ptr}(R_s)}$$

$$\texttt{[CALL-S]}\ \frac{\langle \mathsf{call}_k \rangle^{\theta}_{\delta} \rhd \mathsf{Ptr}[\underline{\mathsf{call}_k}] \quad (R_t = \chi(R_s, C_i, M)) \in \delta \quad R' \in \mathcal{M}^{\mathsf{call_k}}_{f}(R_t) \quad (v_t, C_t, M_t) \in \mathsf{Ptr}(R') \quad \mathsf{Eval}(C_\chi, C_t) = \mathsf{true} \quad M_\chi = \mathsf{true}}{\mathsf{Ptr}(R_t) = \mathsf{Ptr}(R_t)}$$

Figure 13. Rules for pointer resolution at callsites ($C_{\mathsf{call_k}}$ is context condition at the callsite).

therefore, region $R(*y_0)$ is must aliased with $R(*x_0)$ here under context condition $C_1 \vee C_2$ following Figure 6 (C_χ is computed in the same way as P_χ). The pointer resolution is same as rules shown in Figure 7, $R(*y_0)_1$ is strongly updated and has the same points-

to as $R(*x_0)_0$ (Figure 10(d)). Then we move to procedure main, callsite μ/χ are inserted before SSA conversion by following rules in Figure 11. At the first callsite, a and t are inserted into χ to indicate MAY-DEF at the callsite with context C_1. region a and t are mapped to region $R(*x_0)_1$ and $R(*z_0)_1$ (Figure 9), and M_χ of the two χ functions are evaluated to be true to indicate strong updates (Figure 11). Similarly, for inserting χ functions at the second callsite, the only difference is that t and s are weakly updated here. Rules for points-to resolution at the callsites is similar as that for region assignments. The difference is that context conditions are needed to be evaluated in order for context sensitive analysis. The final points-to results are shown in Figure 10(d).

It is understood that each pointed-to location appears only once in $\mathsf{Ptr}(p)$. If (v, C_i, M_i) and (v, C_j, M_j) are added to it together, then both are merged into $(v, C_i \vee C_j, M_i \vee M_j)$.

4. Related Works

References

[1] L.O. Andersen. Program analysis and specialization for the C programming language. 1994.

[2] R. Chatterjee, B.G. Ryder, and W.A. Landi. Relevant context inference. In *POPL '99*, pages 133–146.

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

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

[5] M. Das, B. Liblit, M. F
"ahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. *Static Analysis*, pages 260–278, 2001.

[6] Arnab De and Deepak D'Souza. Scalable flow-sensitive pointer analysis for java with strong updates. In *ECOOP '12*, pages 665–687.

[7] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI '11*, pages 567–577.

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

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

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

[11] M. Hind and A. Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, 25(5):123, 2000.

[12] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08*, pages 249–259.

[13] W. Le and M.L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE '08*, pages 272–282.

[14] O. lhoták. Context-sensitive points-to analysis: is it worth it? In *CC '06*, pages 47–64.

[15] 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.

[16] D. Liang and M. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. *SAS '01*, pages 279–298.

[17] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *FSE'03*, pages 317–326, 2003.

[18] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

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

[20] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM New York, NY, USA, 1996.

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

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

[23] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS '07*, 29(3):16.

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

[25] Ondrej Lhota kKwok Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL '11*.