Object Versioning for Flow-Sensitive Pointer Analysis

Mohamad Barbar
University of Technology Sydney/CSIRO's Data61
Australia

Yulei Sui
University of Technology Sydney
Australia

Shiping Chen
CSIRO's Data61
Australia

***Abstract*—Flow-sensitive points-to analysis provides better precision than its flow-insensitive counterpart. Traditionally performed on the control-flow graph, it incurs heavy analysis overhead. For performance, staged flow-sensitive analysis (SFS) is conducted on a pre-computed def-use (value-flow) graph where points-to sets of variables are propagated across def-use chains sparsely rather than across control-flow in the control-flow graph. SFS makes the propagation of different objects' points-to sets sparse (multiple-object sparsity), however, it suffers from redundant propagation between instructions of the same object's points-to sets (single-object sparsity). The points-to set of an object is often duplicated, resulting in redundant propagation and storage, especially in real-world heap-intensive programs.**

We notice that a simple graph prelabelling extension can identify much of this redundancy in a pre-analysis. With this pre-analysis, multiple nodes (instructions) in the value-flow graph can share an individual memory object's points-to set rather than each node maintaining its own points-to set for that single object. We present object versioning for flow-sensitive points-to analysis, a finer single-object sparsity technique which maintains the same precision while allowing us to avoid much of the redundancy present in propagating and storing points-to sets. Our experiments conducted on 15 open-source programs, when compared with SFS, show that our approach runs up to 26.22× faster (5.31× on average), and reduces memory usage by up to 5.46× (2.11× on average).

***Index Terms*—points-to analysis, flow-sensitivity, sparse value-flow graph**

I. Introduction

Points-to analysis is a fundamental static analysis used to support many other program analyses, such as compiler optimisation [6], [15], vulnerability detection [17], [28], program verification [8], and program slicing [27]. It is a dataflow analysis, with a set union meet operator, which aims to statically determine the possible objects which pointers may point to at runtime. By virtue of being a static analysis, a *sound* points-to analysis can only give an over-approximation of the possible runtime points-to relations. This over-approximation can be made more precise through various precision dimensions, usually at the expense of performance. One such dimension is flow-sensitivity, where control-flow is taken into account.

Unlike flow-insensitive points-to analysis which treats instructions as unordered, flow-sensitive points-to analysis needs to perform the analysis on a data structure which encodes program control-flow. For this, the program's control-flow graph can be used. Since the points-to sets of pointers may change from instruction to instruction with respect to program execution order, we need to maintain a different points-to set for every variable at each instruction. However, in partial SSA form [14], upon which many modern points-to analyses for C/C++ are based on [3], [11], [21], [23], so called top-level variables only have a single definition so a global points-to set for such variables can be used. Unfortunately, address-taken variables, which are those modified indirectly through load and store instructions, still need separate points-to sets maintained at each program point.

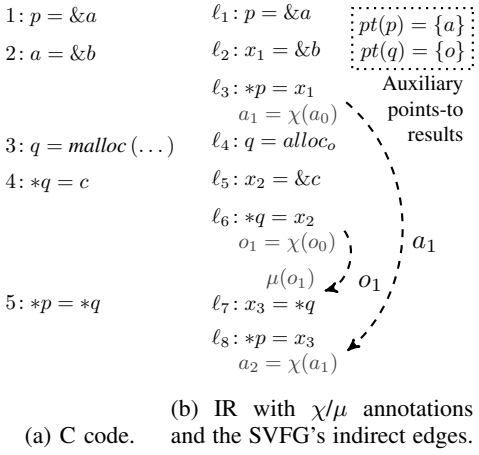
Staged flow-sensitive analysis (SFS) [11] does away with the notion of maintaining a points-to set for every variable at every program point. Rather, by using a sound and imprecise (hence cheaper) points-to analysis, it constructs an over-approximate def-use graph (or *sparse value-flow graph* (SVFG)), on which the flow-sensitive analysis is conducted. Rather than propagating all points-to sets to all program points on the control-flow graph, only a subset of object points-to sets need to be maintained and propagated on the SVFG as determined by the over-approximate def-use relations.

SFS aims to make the propagation of different objects' points-to sets sparse (multiple-object sparsity), however it suffers from excessive propagation of the same object's points-to sets between instructions (single-object sparsity). We notice that multiple instructions often use the same points-to set of an object, but SFS always assumes that such instructions are each computing and maintaining a separate points-to set, thus wasting time and space propagating and storing points-to sets redundantly. Simply put, the points-to set of an object o at an instruction can be reused from another instruction if that points-to set has not been changed (similar to copy-on-write [2]). We need to determine at which instructions the points-to sets of an object o are the same. So, our goal is to achieve finer single-object sparsity and complement existing multiple-object sparsity by soundly "versioning" object o at each instruction which may use o such that instructions which share o's version can safely share that object's points-to set.

We introduce *meld labelling*, a prelabelling extension for directed graphs, to version objects. Meld labelling extends a prelabelling by propagating labels such that each node's label is a "melding" of the labels of its incoming neighbour nodes. This melding process continues until a fixed-point is reached. At each store instruction which may write to an object o, we assign a distinct label (i.e., a distinct version) for o. Then, we perform meld labelling on top of this prelabelling (though we

TABLE I: Analysis domains and the LLVM-like instruction set.

Analysis domains			LLVM-like instruction set	
ℓ	$\in \mathcal{L}$	instruction labels	ALLOC	$p = alloc_{\hat{o}}$
x, y, z	$\in \mathcal{S}$	stack variables	PHI	$p = \phi(q, r)$
g	$\in \mathcal{G}$	global variables	MEMPHI	$o = \phi(a, b)$
p, q, r	$\in \mathcal{P} = \mathcal{S} \cup \mathcal{G}$	top-level variables	CAST	$p = (t)\, q$
$\hat{o}$	$\in \mathcal{O}$	base abstract objects	FIELD	$p = \&q \rightarrow f_k$
$\hat{o}.f_k$	$\in \mathcal{F}$	abstract field objects	LOAD	$p = *q$
o, a, b	$\in \mathcal{A} = \mathcal{O} \cup \mathcal{F}$	address-taken objects	STORE	$*p = q$
v	$\in \mathcal{V} = \mathcal{P} \cup \mathcal{A}$	variables	CALL	$p = q(r_1, \ldots, r_n)$
$\mathcal{SN}$	$\subset \mathcal{A}$	singleton objects	FUNENTRY	$fun(r_1, \ldots, r_n)$
κ, ε	$\in \mathcal{K}$	labels/versions	FUNEXIT	$ret_{fun}p$



(a) C code. (b) IR with χ/μ annotations and the SVFG's indirect edges.

Fig. 1: C code and its IR and SVFG.

ensure prelabelled nodes never change). The result is that all nodes which have the same label (version) for o rely on the same set of modifications (through stores) to o's points-to set. If two instructions rely on the same set of modifications to o, o's points-to set must be the same at those two instructions and can be shared. With the objects now versioned, we can perform flow-sensitive points-to analysis with finer single-object sparsity, which is more efficient than SFS and while maintaining precision.

To summarise, this paper describes:

- A new object versioned approach to flow-sensitive points-to analysis with finer single-object sparsity, improving efficiency while maintaining the same precision as SFS.
- Meld labelling, a fast and simple prelabelling extension for directed graphs, which we use to determine equivalent points-to sets for the same object at different instructions.
- An evaluation using 15 open-source programs. Compared with SFS, our experiments show that our approach runs up to 26.22× faster (5.31× on average), and reduces memory usage by up to 5.46× (2.11× on average).

II. PROGRAM REPRESENTATION

This section introduces the analysis domain, an LLVM-like instruction set, and the SVFG used in our points-to analysis.

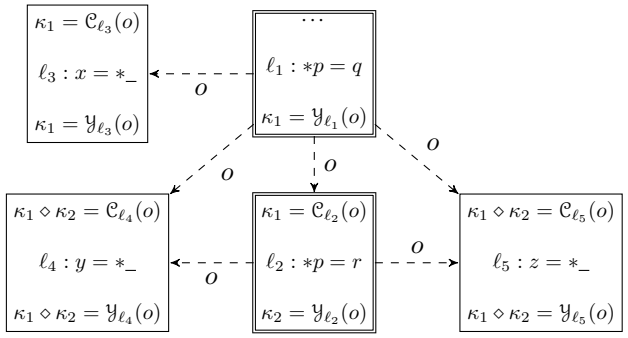
A. Domain and intermediate representation

Following previous points-to analyses for C and C++ [3], [10], [11], [13], [16], we perform our analysis on an LLVM-like [14] instruction set without loss of generality. Table I describes the target of our analysis. The set of all variables $\mathcal{V}$ is made up of two sets, $\mathcal{A} = \mathcal{O} \cup \mathcal{F}$, which represents all possible abstract objects and their fields, i.e. *address-taken variables*, and $\mathcal{P} = \mathcal{S} \cup \mathcal{G}$, which represents all stack and global pointers, i.e. *top-level pointers*. Top-level pointers are explicit unlike address-taken variables which are implicit and accessed indirectly at STORE and LOAD instructions through top-level pointers. All instructions are labeled with a label ℓ from $\mathcal{L}$. $\mathcal{SN}$ is the set of all abstract objects which represent exactly one real object. We call such objects *singletons* [16]. $\mathcal{K}$ contains the set of all labels or versions which we perform meld labelling upon and ε is the identity in $\mathcal{K}$. Meld labelling is discussed in Section IV.

Following conversion to partial SSA form [24], target programs are made up of 10 types of instructions. Of these, 8 make up functions bodies: ALLOC ($p = alloc_o$, allocates an object on the stack, globally, or on the heap), PHI ($p = \phi(q, r)$, selects the value of a top-level pointer at a join point in the control-flow), MEMPHI ($o = \phi(a, b)$, selects the value of an address-taken object at a join point in the control-flow), CAST ($p = (t)\, q$, casts a pointer to another type), FIELD ($p = \&q \rightarrow f_k$, retrieves a pointer pointing to a field of an aggregate object), LOAD ($p = *q$, reads the value of an object), STORE ($*p = q$, writes the value of an object), and CALL ($p = q(r_1, \ldots, r_n)$, calls a function with the specified arguments). The remaining 2 instructions connect calls and returns to their target. Each function has a single FUNENTRY instruction ($fun(r_1, \ldots, r_n)$) which contains the parameters of a function and a FUNEXIT instruction ($ret_{fun}p$). LLVM's `UnifyFunctionExitNodes` ensures all functions have a single FUNEXIT instruction. Figure 1 shows an example of intraprocedural C code and its corresponding IR. Some temporary variables (x_*) are required for the translation to split C statements into simpler instructions. The arrows and gray annotations are discussed in the next section.

B. Value-flow graph

An auxiliary analysis is required to construct the SVFG upon which the flow-sensitive analysis will be performed. For this purpose, SFS [11] uses a flow-insensitive inclusion-based points-to analysis (also known as Andersen's style points-to analysis [1]). Andersen's analysis is well studied, relatively performant, and precise enough to produce an acceptable SVFG. The nodes of the SVFG are trivial to determine since they are simply the program's instructions. The edges are split into two types: direct edges which do not require the auxiliary analysis to infer and indirect edges which require the pre-computed points-to results. Direct edges represent the value-flow of top-level pointers ($\mathcal{P}$) and can also be determined trivially because top-level pointers in partial SSA form can only be accessed directly by name, not indirectly through



(a) SVFG fragment from GNU Coreutil's `true`. We assume $pt(p) = \{o\}$, $pt(q) = \{a\}$, and $pt(r) = \{b\}$ during flow-sensitive solving.

	SFS	Our approach
Object points-to sets	$pt_{\ell_1\|}(o) = \{a\}$ $pt_{\|\ell_2}(o) = \{a\}$ $pt_{\|\ell_3}(o) = \{a\}$	$pt_{\kappa_1}(o) = \{a\}$
	$pt_{\ell_2\|}(o) = \{a, b\}$	$pt_{\kappa_2}(o) = \{a, b\}$
	$pt_{\|\ell_4}(o) = \{a, b\}$ $pt_{\|\ell_5}(o) = \{a, b\}$	$pt_{\kappa_1 \diamond \kappa_2}(o) = \{a, b\}$
Generated constraints	$pt_{\ell_1\|}(o) \subseteq pt_{\|\ell_2}(o)$	
	$pt_{\ell_1\|}(o) \subseteq pt_{\|\ell_3}(o)$	
	$pt_{\ell_1\|}(o) \subseteq pt_{\|\ell_4}(o)$ $pt_{\ell_2\|}(o) \subseteq pt_{\|\ell_4}(o)$	$pt_{\kappa_1}(o) \subseteq pt_{\kappa_1 \diamond \kappa_2}(o)$ $pt_{\kappa_2}(o) \subseteq pt_{\kappa_1 \diamond \kappa_2}(o)$
	$pt_{\ell_1\|}(o) \subseteq pt_{\|\ell_5}(o)$ $pt_{\ell_2\|}(o) \subseteq pt_{\|\ell_5}(o)$	

(b) Points-to sets stored and propagated.

Fig. 2: A motivating example.

another pointer. Indirect edges represent the value-flow of address-taken objects ($\mathcal{A}$) and rely on the auxiliary analysis because they are defined and used through STORE and LOAD instructions (by dereferencing a top-level pointer) and it is otherwise unknown which top-level pointer points to which address-taken object. Furthermore, the join points of address-taken objects (MEMPHI instructions) are unknown without a prior analysis. Thus, these indirect value-flows are determined through the memory SSA form [5].

To realise the memory SSA form, we annotate instructions with χ and μ functions using information from the auxiliary analysis. All STORE instructions which may store to an object o are annotated with a χ function as $o = \chi(o)$ indicating that o may be defined, potentially based on the previous definition of o. Similarly, all LOAD instructions which may load an object o are annotated with a μ function as $\mu(o)$. FUNENTRY and FUNEXIT instructions are annotated with χ and μ functions, respectively, to mimic parameter passing and returning of address-taken objects. Each CALL instruction is annotated with $\mu(o)$ or $o = \chi(o)$ if o may be used or modified, respectively, in any callees. The address-taken objects can then be converted to SSA form, giving objects unique definitions. For example, $o = \chi(o)$ may become $o_2 = \chi(o_1)$. MEMPHI instructions are then introduced where the definition of an object needs to be selected (analogous to PHI instructions for top-level pointers).

In all, the final SVFG contains every instruction as its own node (including inserted MEMPHI instructions), direct value-flows which represents the definition and use of a top-level pointer, and indirect value-flows which represent a *possible* definition and use of an address-taken object at the source and destination respectively. Given the SVFG of a program, SFS is able to conduct a sparse analysis by propagating points-to information along the def-use/value-flow chains rather than propagating and maintaining *all* points-to information at *each* program point.

Figure 1b shows the SVFG on top of the IR presented in the previous section. We have omitted the trivial direct edges to avoid visual noise, and have included the χ/μ annotations (determined through the auxiliary points-to analyses results), and the indirect value-flow edges ($\ell_3 \xrightarrow{a_1} \ell_8$ and $\ell_6 \xrightarrow{o_1} \ell_7$).

III. MOTIVATING EXAMPLE

Figure 2 presents a motivating example to illustrate the key idea of our approach. It shows an SVFG fragment derived from a real program's SVFG (`true` in GNU Coreutils[1]) in Figure 2a, with some extraneous edges and nodes removed, and the required points-to sets and propagation constraints for flow-sensitive analysis in Figure 2b (SFS and our approach). Direct edges are omitted (for readability; they are irrelevant to our purposes) so all edges are indirect edges and they are labelled with an object o. The double-lined nodes are STORE nodes which may define objects (i.e., place other objects in their points-to sets) and the remaining nodes are LOAD nodes which may use objects (in this case, o). The ℓ labels represent instruction labels for ease of reference. For exemplary purposes, we assume $pt(p) = \{o\}$, $pt(q) = \{a\}$, and $pt(r) = \{b\}$ (according to the current state of the flow-sensitive analysis), so the two STORE nodes may define o. According to Figure 2b, o's resulting points-to set is $\{a\}$ immediately after ℓ_1, and immediately before ℓ_2 and ℓ_3 and it is $\{a, b\}$ immediate after ℓ_2 and immediately before ℓ_4 and ℓ_5. The $\mathcal{C}$ and $\mathcal{Y}$ functions and the usage of versions (κ) are introduced by our approach and described in Section III-c.

*a) **SFS**:* In SFS, every SVFG node maintains an IN set and an OUT set to represent the points-to sets of objects before and after each instruction, respectively. STORE instructions update the points-to sets of objects in the node's OUT set to propagate forward. When an indirect edge is labelled with an object o, its points-to set is propagated from the OUT set of the source node to the IN set of the destination node. Thus,

[1]GNU's `true` pulls in, and calls, functions common across the GNU Coreutils suite. So while simple, it requires the facilities of an interprocedural points-to analysis unlike more straightforward implementations of `true`.

the points-to set of an object in the IN set of a node is the union of all the points-to sets of that object in the OUT sets of its incoming neighbours. We denote the points-to set of o in the IN set of ℓ as $pt_{|\ell}(o)$ and the points-to set of o in the OUT set of ℓ as $pt_{\ell|}(o)$. Since top-level pointers are defined once in partial SSA form, only a global points-to set (like $pt(p)$) is required for each pointer.

*b) **Applying SFS**:* In the example, SFS needs to maintain a points-to set for o in the IN set of every node and in the OUT of the two STORE nodes. This necessitates redundant storage and propagation since some of these points-to sets may be equivalent and can be shared instead of repeatedly storing the same points-to set and repeatedly propagating to form the same points-to sets (Column 2 of Figure 2b shows the maintained points-to sets and required constraints during flow-sensitive resolution for SFS). For example, the points-to set of o in the IN sets of ℓ_2 and ℓ_3 are equivalent to the points-to set of o in the OUT set of ℓ_1 since their IN sets are formed by propagation of ℓ_1's OUT set only. Similarly, the IN sets of ℓ_4 and ℓ_5 are equivalent since they are both made up of the union of the OUT sets of ℓ_1 and ℓ_2 so $pt_{|\ell_4}(o) = pt_{|\ell_5}(o)$.

*c) **Our approach**:* Instead of retrieving the points-to set of an object o from an IN or OUT set stored at each node, we break o into different versions so that the points-to set of version κ of o ($pt_\kappa(o)$) is global and shared by multiple SVFG nodes. We say that an instruction ℓ "consumes" version $\mathcal{C}_\ell(o)$ of o and "yields" version $\mathcal{Y}_\ell(o)$ of o. The version of an object which an instruction consumes can be used to access the object's points-to set before the instruction and the version which it yields can be used to access the object's points-to set after the instruction. Thus, for example, a STORE instruction ℓ storing to o would operate on $pt_{\mathcal{Y}_\ell(o)}(o)$ and a LOAD instruction ℓ reading from o would read from $pt_{\mathcal{C}_\ell(o)}(o)$. The flow-sensitive analysis propagates points-to sets from $pt_{\mathcal{Y}_\ell(o)}(o)$ to $pt_{\mathcal{C}_{\ell'}(o)}$ if an edge $\ell \xrightarrow{o} \ell'$ exists rather than propagating from $pt_{\ell|}(o)$ to $pt_{|\ell'}(o)$. Versions are simply labels or IDs which matter in how they relate to each other. With a fast pre-analysis, we can ensure the following:

$$\mathcal{C}_\ell(o) = \mathcal{C}_{\ell'}(o) \Rightarrow pt_{|\ell}(o) = pt_{|\ell'}(o) \quad (1)$$

$$\mathcal{C}_\ell(o) = \mathcal{Y}_{\ell'}(o) \Rightarrow pt_{|\ell}(o) = pt_{\ell'|}(o) \quad (2)$$

$$\mathcal{Y}_\ell(o) = \mathcal{Y}_{\ell'}(o) \Rightarrow pt_{\ell|}(o) = pt_{\ell'|}(o) \quad (3)$$

The result is that for each object, nodes will have a consumed version and a yielded version which can be used to access points-to sets of objects instead of accessing them through IN and OUT sets (which we completely forego). Importantly, nodes accessing the same version of an object can share a points-to set for that object.

*d) **Applying our approach**:* The consumed and yielded versions of the nodes in our example can be seen in Figure 2a. The points-to set of o in the IN sets of ℓ_2 and ℓ_3 is generated through the propagation of the points-to set of o in the OUT set of ℓ_1 and are thus equivalent, so $\mathcal{C}_{\ell_2}(o) = \mathcal{C}_{\ell_3}(o) = \mathcal{Y}_{\ell_1}(o) = \kappa_1$, and ℓ_4 and ℓ_5 have a similar relationship ($\kappa_1 \diamond \kappa_2$ is another version separate to κ_1 and κ_2 and the $\diamond$ operator is described in Section IV-B). Since multiple nodes may share versions, we can store fewer points-to sets. We also generate fewer propagation constraints since there are fewer points-to sets which need to be formed. Improvements to the number of points-to sets required and the number of propagation constraints generated are shown in Figure 2b where they are listed in Column 3 for our approach. Instead of storing 6 points-to sets, we store 3, and we reduce the number of propagation constraints generated from 6 to 2. Hence, our approach saves both space, through storing fewer points-to sets, and time, through fewer propagations.

IV. APPROACH

This section first formulates the ideas of existing flow-sensitive points-to analysis and our analysis. We then detail the 3 main parts of our approach, i.e., meld labelling, using meld labelling to version objects, and performing finer-grained sparse analysis using versioned objects.

We utilise various notation, some introduced in this section, frequently. The following describes some of the more common notation for ease of reference,

$pt(p)$	The points-to set of p.
$pt_\kappa(o)$	The points-to set of version κ of o.
$pt_{\|\ell}(o)$	The points-to set of o immediately before instruction ℓ.
$pt_{\ell\|}(o)$	The points-to set of o immediately after instruction ℓ.
$pt^a(o)$	The points-to set of o according to the auxiliary analysis.
$\delta(\ell)$	Returns whether instruction ℓ is a δ node in the SVFG.
$\kappa_1 \diamond \kappa_2$	Melding of versions κ_1 and κ_2.
$\mathcal{C}_\ell(o)$	The version of o which instruction ℓ consumes.
$\mathcal{Y}_\ell(o)$	The version of o which instruction ℓ yields.

A. Flow-sensitive points-to analysis

Traditional data-flow-based flow-sensitive points-to analysis computes the maximal-fixed-point solution as an over approximation of the meet-over-all paths problem by solving an iterative data-flow problem. The analysis is performed on an interprocedural control-flow graph (ICFG) by maintaining and computing points-to sets of each variable v immediately before ($pt_{|\ell}(v)$) and after ($pt_{\ell|}(v)$) each statement ℓ. We write IN_ℓ and OUT_ℓ to denote all the points-to sets at $|\ell$ and $\ell|$.

For each node ℓ in the ICFG, flow-sensitive analysis iteratively computes the following until a fixed point is reached:

$$\text{IN}_\ell = \bigcup_{\ell' \in pred(\ell)} \text{OUT}_{\ell'} \quad (4)$$

$$\text{OUT}_\ell = Gen_\ell \cup (\text{IN}_\ell - Kill_\ell) \quad (5)$$

where $pred(\ell)$ denotes the predecessor nodes of ℓ in the ICFG and Gen_ℓ and $Kill_\ell$ represent the points-to relations generated and killed after analysing statement ℓ.

Staged flow-sensitive analysis (SFS) [11] is an optimised version of traditional data-flow-based analysis. Instead of resolving data-flow facts on the ICFG, SFS operates on top of a def-use graph or sparse value-flow graph (SVFG) [11], [23] which captures the program's def-use chains conservatively. In particular, an edge $\ell \xrightarrow{v} \ell'$, where $v \in \mathcal{V}$, from statement ℓ to statement ℓ' signifies a potential def-use chain for v with its definition at ℓ and use at ℓ'. This representation is sparse since the intermediate program points between ℓ and ℓ' are omitted. With the SVFG, SFS can perform points-to analysis using the following equations,

$$\text{IN}_\ell^v = \bigcup_{\ell' \in spred(\ell,v)} \text{OUT}_{\ell'}^v \tag{6}$$

$$\text{OUT}_\ell^v = Gen_\ell^v \cup (\text{IN}_\ell^v - Kill_\ell^v) \tag{7}$$

where $spred(\ell, v)$ denotes the set of instructions with a value-flow of variable $v \in \mathcal{V}$ to ℓ in the SVFG. Every data-flow set (e.g., IN_ℓ^v and OUT_ℓ^v) is qualified by a variable v at ℓ where v is defined or used. Compared with traditional flow-sensitive analysis on the ICFG, by computing and maintaining points-to sets on the SVFG, SFS can separate points-to set propagation to achieve multiple-object sparsity.

Our approach aims to reduce the number of IN^v and OUT^v for each v, such that only one points-to solution for each version κ of v is computed and maintained globally (note that here we only care about the version for abstract memory objects rather than top-level pointers which are only defined once). For example, $pt_\kappa(o)$, the points-to set of object o with version κ, is stored globally rather than at each program point. This approach avoids redundant points-to propagation within a single object (single-object sparsity) through meld-labelling-based object versioning.

We thus have,

$$\text{IN}_\ell^{v:\kappa} = \bigcup_{\ell' \in spred(\ell,v)} \text{OUT}_{\ell'}^{v:\kappa'} \tag{8}$$

$$\text{OUT}_\ell^{v:\kappa'} = Gen_\ell^v \cup (\text{IN}_\ell^{v:\kappa} - Kill_\ell^v) \tag{9}$$

where $\text{OUT}_\ell^{v:\kappa}$ represents the retrieval of the globally maintained points-to set of object v with version $\kappa = \Upsilon_\ell(v)$, and similarly for $\text{IN}_\ell^{v:\kappa}$ and $\kappa = \mathcal{C}_\ell(v)$.

B. Meld labelling

A meld labelling is a prelabelling extension on directed graphs where each node is labelled with a "melding" of the labels found at the source ends of its incoming edges. Given that the label domain is $\mathcal{K}$, to achieve meld labelling, we define the *meld operator* $\diamond : \mathcal{K}^2 \mapsto \mathcal{K}$. The meld operator can be any operation that is commutative, associative, idempotent, and has an identity element in relation to $\mathcal{K}$. In other words, given that $\kappa_1, \kappa_2, \kappa_3, \varepsilon \in \mathcal{K}$ and ε is the identity,

$$\kappa_1 \diamond \kappa_2 = \kappa_2 \diamond \kappa_1 \qquad \text{(Commutativity)}$$
$$\kappa_1 \diamond (\kappa_2 \diamond \kappa_3) = (\kappa_1 \diamond \kappa_2) \diamond \kappa_3 \qquad \text{(Associativity)}$$
$$\kappa_1 \diamond \kappa_1 = \kappa_1 \qquad \text{(Idempotence)}$$
$$\kappa_1 \diamond \varepsilon = \kappa_1 \qquad \text{(Identity)}$$

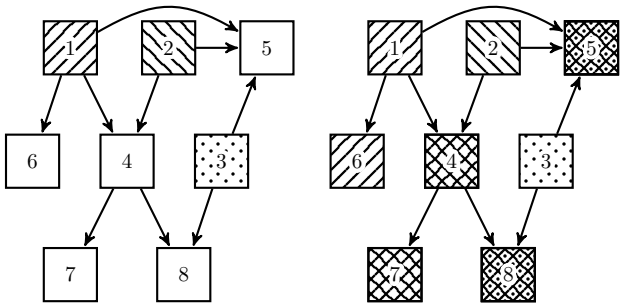
The set union operator, $\cup$, and the bitwise-or operator found in many programming languages, are examples of suitable meld operators when labels are represented as sets or bit sets.

The initial graph is prelabelled with labels in $\mathcal{K}$, except ε, according to some condition chosen according to the meld labelling's purpose. Nodes which are not part of that prelabelled subset are labelled with the identity ε. The meld labelling process is simple: meld the label of each node with its incoming neighbours' labels repeatedly until all nodes which would be labelled are labelled (i.e., that a fixed-point is reached). This is exemplified by the $[\texttt{MELD}]^{\texttt{M}}$ rule in Figure 3 where n and n' are nodes, and κ_n and $\kappa_{n'}$ are the labels of n and n'.

$$[\texttt{MELD}]^{\texttt{M}} \frac{n' \to n \quad \kappa_{n'} \neq \varepsilon}{\kappa_n = \kappa_{n'} \diamond \kappa_n}$$

Fig. 3: Meld labelling process. κ_n is the label of node n.

Prelabelled nodes and nodes reachable by any prelabelled node will be labelled with some non-ε label by the end of the meld labelling process. All other nodes will finish labelled with ε. The final result is that nodes have been split into equivalence classes according to the melding of prelabels which transitively reach them. Those which finish with ε are in their own class: nodes unreachable by any prelabelled node.



(a) Prelabelled graph. (b) Result of meld labelling.

Fig. 4: An example of meld labelling. Patterns are labels and $\diamond$ combines them. The blank pattern is the identity.

Figure 4 shows an example of a prelabelled graph and its state after meld labelling. In this instance, the label domain, $\mathcal{K}$, is made up of patterns, specifically $\mathcal{K} = \{$ □, ▨, ▧, ⁘, ▩, ▧, ▨, ▩$\}$ where □ is the identity. Nodes are prelabelled with ▧, ▨, and ⁘, and remaining nodes are labelled with the identity □. The meld operator combines the patterns. With the following subset of cases for the meld operator (though other subsets would be sufficient too), knowledge that $\diamond$ is commutative, associative, and idempotent, and that □ is the identity, all cases can be derived,

▧ $\diamond$ ▨ = ▩ ▧ $\diamond$ ⁘ = ▧ ▨ $\diamond$ ⁘ = ▨
▩ $\diamond$ ⁘ = ▩

In Figure 4, despite nodes 5 and 8 (and similarly nodes 4 and 7) having different incoming neighbours, they finish with the same label because the melding of their incoming neighbours' labels is the same. Thus, equivalence is of labels at nodes is not a result of sharing incoming neighbours but by sharing the set of labels (from prelabelling) which reach them.

1) Complexity: In the worst case, meld labelling takes $\mathcal{O}(|E|P)$ time where P is the number of nodes prelabelled with other than the identity and E is the set of edges. This is because each label already on the graph may need to be propagated along each edge (whether as a part of a melding or on its own). In space, it would always take $\mathcal{O}(|N|)$ space where N is the set of nodes since a label would need to be stored at each node.

C. Versioning objects using meld labelling

SFS propagates points-to sets across instructions such that at each instruction there is a points-to set (or two) for every object it may use or define. However, two instructions which rely on the exact same modifications to an object's points-to set can share the points-to set they use. In such a case, we say that those two instructions (i.e., SVFG nodes) **consume** the same **version** of an object. Complementary to consuming a version of an object, we say that the version an instruction **yields** is the version of an object it defines (and if it does not define an object, it yields what it consumes).

A version of an object represents a state of that object's points-to set such that any change in an objects points-to set warrants a version. There are two ways the points-to set of an object o at a program point can change: (1) through a STORE instruction $*p = q$ when p points to o, and (2) through the merging of the points-to sets of o at different program points *before* a specific instruction (i.e., when the points-to set of an object in an IN set is the union of that in *multiple* OUT sets). All instructions only consume a single version of an object and yield a single version. Determining the versions of an object, which version each instruction may consume, and which version each instruction may yield requires points-to information, and since flow-sensitive points-to information is obviously unavailable before the flow-sensitive analysis is performed, we use the pointer information of the auxiliary analysis (Andersen's analysis, in our case). This may give us more versions than necessary whereby two versions may be collapsible into a single version (both versions have equivalent points-to sets per the flow-sensitive analysis) if versioning was done using more precise points-to information, but the over-approximation is sound and is still performant as evaluated in Section V.

We give each instruction ℓ a $\mathcal{C}$ (for consume) function, defined as,

Definition 1: $\mathcal{C} : \mathcal{A} \mapsto \mathcal{K}$ where $\mathcal{C}_\ell(o)$ is the version of o which ℓ consumes.

And a $\mathcal{Y}$ (for yield) function, defined as,

Definition 2: $\mathcal{Y} : \mathcal{A} \mapsto \mathcal{K}$ where $\mathcal{Y}_\ell(o)$ is the version of o which ℓ yields.

Overall, versioning objects allows two or more instructions to access the same points-to set of o if those instructions rely on the same modifications to o (through stores and control-flow merges). By foregoing IN and OUT sets we can save time and space propagating and storing points-to sets, since many instruction may consume/yield the same version of an object (which is represented by a single points-to set).

Meld labelling discussed in the previous section encodes reliances between nodes according to the prelabelling which it extends such that if nodes share the same label, they rely on the same prelabelled nodes. With a prelabelling of nodes which modify objects' points-to sets (with some caveats), we can use meld labelling to version objects. Since the set union operator which is used by inclusion-based points-to analysis fulfils the requirements of the meld operator, meld labelling can be seen as a light-weight simulation of the real points-to analysis's propagation using labels/versions to represent points-to sets and relying on imprecise points-to information from the auxiliary analysis rather than flow-sensitive information. The first step is to appropriately describe the graph which we wish to meld label and prelabel it. In the context of meld labelling for versioned staged flow-sensitive points-to analysis (VSFS), our analysis, the terms label and version are synonymous.

*1) **Prelabelling**:* At any given node, multiple objects may be used or defined. Thus, we are not aiming for a single version at each node, but rather a version per object (of interest) at each node. Furthermore, we want two versions per object at each node: one to consume and one to yield, because some nodes may not propagate (yield) the same version they use (consume). Since the only points-to information available to us is from the imprecise auxiliary analysis results, we are aiming for a versioning that signifies the worst case of the flow-sensitive analysis (unrealistically being no more precise than the auxiliary analysis).

In the course of execution, assuming a STORE instruction may point to o, it may propagate forward a different points-to set of o than the one propagated to it because it may modify o's points-to set. Whether a STORE instruction $*p = q$ modifies an object's points-to set relies on whether p points to o and whether the points-to set of q contains elements not found in the points-to set of o. This information is unavailable to us before the analysis so we soundly assume that STORE instructions yield a different version to that which they consume in case they modify the points-to sets of the objects which they point to during flow-sensitive solving. Having spurious versions is sound, and as it stands, SFS can be thought of as having a unique consumed and yielded version for each object in the IN and OUT, respectively, at each node. Since the version a STORE yields is a new version, not necessarily reliant on any other version, prelabelling should occur at STORE nodes. For each STORE instruction ℓ, we need to provide ℓ's yielded version as a prelabel (i.e., set $\mathcal{Y}_\ell(o)$).

Example 1: The state of the motivating example from Figure 2 after prelabelling is shown in Figure 5. The STORE nodes are given prelabels (κ_1 and κ_2) to yield for o, and all other consumed or yielded versions are set to the identity.

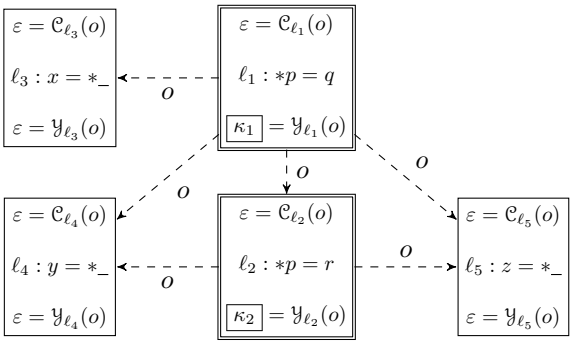


Fig. 5: SVFG from the motivating example after the prelabelling phase. Versions introduced in this phase are boxed.

If we use the results from the (Andersen's) auxiliary analysis to perform call graph resolution [11], this is sufficient. However, we perform on-the-fly call graph resolution using results from the flow-sensitive points-to analysis itself, which is more precise and performant, so some nodes are missing incoming edges which may affect versioning. In other words, some reliances between versions (e.g. that $\mathcal{C}_\ell(o)$ is a melding of itself and some $\mathcal{Y}_{\ell'}(o)$) are not determined until we perform the flow-sensitive analysis. To remedy this, any such node, referred to as a δ node, consumes a unique version (from prelabelling) for each object it may propagate forward. As determined by the auxiliary analysis, these nodes are any FUNENTRY instruction that may be the target of an indirect call and any CALL instruction which makes an indirect call (i.e., the return target of an indirect call). This is a sound over-approximation since in actuality we either need to introduce a new version, which we have done, or reuse a version, which enhances performance and would only be possible if we had flow-sensitive points-to information available.

For convenience, we define a δ function to encode this as,

Definition 3: $\delta : \mathcal{L} \mapsto \mathbb{B}$ such that

$$\delta(\ell) = \textit{true} \Leftrightarrow \exists \ell' \in \mathcal{L}.\ \exists o \in \mathcal{O}.\ \ell' \not\xrightarrow{o} \ell \wedge P(\ell' \xrightarrow{o} \ell)$$

where $P(\ell' \xrightarrow{o} \ell)$ indicates the possibility of an edge $\ell' \xrightarrow{o} \ell$ being created during the flow-sensitive analysis due to on-the-fly call graph resolution. For all objects o, every other node's consumed and yielded versions are set to the identity ε.

Prelabelling is fast to the point where time taken is inconsequential as it only performs a linear scan on the SVFG and sets $\mathcal{C}$ or $\mathcal{Y}$ to new versions for a relatively small number of nodes. The inference rules in Figure 6 show this performed on a graph with all $\mathcal{C}$ and $\mathcal{Y}$ having been already set as the identity ε. The $[\text{STORE}]^{\text{P}}$ rule ensures that STORE instructions yield a new version for each object they may define, as determined by the auxiliary analysis, and the $[\text{OTF-CG}]^{\text{P}}$ rule ensures that δ nodes similarly consume a new version for each object which they may eventually yield in case such is necessary.

*2) **Versioning with meld labelling**:* At this point, we have an SVFG with the versions consumed and yielded set at a small portion of nodes (prelabels). In our meld labelling of the prelabelled SVFG we need to account for the fact that

$$[\text{STORE}]^{\text{P}}\ \frac{\ell : *p = q \quad o \in pt^a(p) \quad \varepsilon = \mathcal{Y}_\ell(o)}{\mathcal{Y}_\ell(o) = nv(o)}$$

$$[\text{OTF-CG}]^{\text{P}}\ \frac{\delta(\ell) \quad \ell \xrightarrow{o} \ell' \quad \varepsilon = \mathcal{C}_\ell(o)}{\mathcal{C}_\ell(o) = nv(o)}$$

Fig. 6: Prelabelling inference rules. $nv(o)$ returns a new version for o and $pt^a(p)$ is the points-to set of p according to the auxiliary analysis.

edges are labelled with address-taken objects. We only need to propagate versions for objects along edges labelled with that object because if there does not exist an edge $\ell \xrightarrow{o} \ell'$ then ℓ' is unaffected by any definition of o at ℓ.

Example 2: In Figure 7, $\kappa_1 = \mathcal{Y}_{\ell_1}(a)$ is only propagated along the edge to ℓ_2 labelled with a. This occurs similarly for ℓ_3 and b.

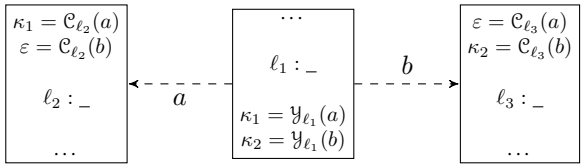


Fig. 7: An example SVFG involving two objects, a and b.

That each node may have two versions per object (a consumed and a yielded version) also needs to be considered. We thus perform meld labelling propagation by introducing propagation "internal" to, and "external" to, nodes. Internal propagation occurs when a node yields what it consumes, which are all non-STORE nodes since no other type of node can ever propagate a different points-to set for an object o than the one propagated to it. At such a node ℓ, $\mathcal{Y}_\ell(o) = \mathcal{C}_\ell(o)$ for all $o \in \mathcal{A}$. When an edge $\ell \xrightarrow{o} \ell'$ exists, we perform external propagation. In such a case, we meld $\mathcal{Y}_\ell(o)$ into $\mathcal{C}_{\ell'}(o)$ (i.e, ℓ' consumes what ℓ yields, and it potentially consumes other versions too) except when $\delta(\ell')$ because $\mathcal{C}_{\ell'}(o)$ would be a prelabel and they are not changed. We explicitly avoid changing what was set in the prelabelling phase as unique versions were specifically chosen for those positions, and we want to maintain that. More formally, the inference rules in Figure 8 will, for each node, provide the consumed and yielded version of objects used at that node.

$$[\text{EXTERNAL}]^{\text{V}}\ \frac{\ell \xrightarrow{o} \ell' \quad \neg\delta(\ell')}{\mathcal{C}_{\ell'}(o) = \mathcal{C}_{\ell'}(o) \diamond \mathcal{Y}_\ell(o)}$$

$$[\text{INTERNAL}]^{\text{V}}\ \frac{\neg \ell : *p = \_}{\mathcal{Y}_\ell(o) = \mathcal{C}_\ell(o)}$$

Fig. 8: Meld labelling inference rules for versioning.

The $[\text{EXTERNAL}]^{\text{V}}$ rule propagates a yielded version from the incoming neighbours of a node and melds that with the consumed version of that node. This rule is similar to $[\text{MELD}]^{\text{M}}$ in the original definition of meld labelling in

Section IV-B. It excludes δ nodes because they have had their relevant consumed versions set in the prelabelling phase. The [INTERNAL]$^{\text{V}}$ rule ensures that any node which yields what it consumes (i.e., non-STORE nodes) has its yielded version set to its consumed version.

Example 3: In Figure 9, we revisit our motivating example from Figure 2 again after being prelabelled in Example 2. o's version is propagated externally from $\mathcal{Y}_{\ell_1}(o)$ to $\mathcal{C}_{\ell_2}(o)$, $\mathcal{C}_{\ell_3}(o)$, $\mathcal{C}_{\ell_4}(o)$ and $\mathcal{C}_{\ell_5}(o)$, and from $\mathcal{Y}_{\ell_2}(o)$ to $\mathcal{C}_{\ell_4}(o)$ and $\mathcal{C}_{\ell_5}(o)$. When more than one version is propagated to another node, melding occurs, as can be seen in the consumed version of o at nodes ℓ_4 and ℓ_5 ($\kappa_1 \diamond \kappa_2$). Finally, internal propagation occurs at ℓ_3, ℓ_4, and ℓ_5 since they are not STORE nodes and yield what they consume. For example, $\mathcal{Y}_{\ell_3}(o) = \mathcal{C}_{\ell_3}(o)$.

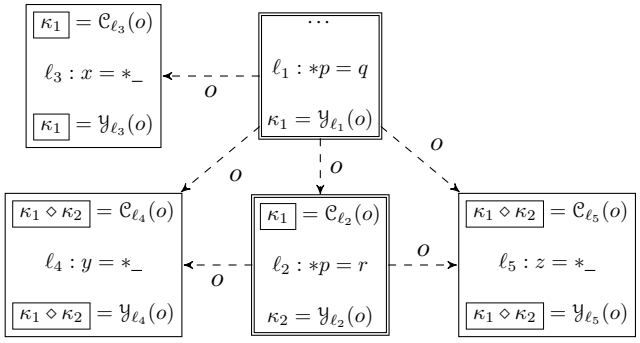


Fig. 9: The SVFG from the motivating example after being versioned. Consumed/yielded versions changed during meld labelling are boxed.

D. Flow-sensitive points-to analysis using object versioning

At this point, every instruction which may access the points-to set of an address-taken object is labeled with two versions for that object: the version it consumes and the version it yields. We use these versions to choose which points-to set to access for each object of interest instead of accessing objects' points-to sets from IN/OUT sets in order to avoid maintaining redundant points-to sets at each program point. The inference rules in Figure 10 modify SFS [11] to use versions instead of IN/OUT sets. We use the notation $pt_\kappa(o)$ to refer to the points-to set of o version κ and we refer to our new formulation of a flow-sensitive points-to analysis as versioned staged flow-sensitive points-to analysis (VSFS).

The [LOAD]$^{\text{F}}$ and [STORE]$^{\text{F}}$ rules work exactly as their original counterparts except they, instead of accessing the points-to set of o from an IN or OUT set, use the consumed and yielded versions of o's points-to sets at ℓ, respectively. For LOAD instruction $p = *q$, the [LOAD]$^{\text{F}}$ rule adds q's pointees' points-to sets (consumed versions) into p's points-to set, and for STORE instruction $*p = q$, the [STORE]$^{\text{F}}$ rule adds q's points-to set into p's pointees' points-to sets (yielded versions). The [SU/WU]$^{\text{F}}$ rule propagates from the points-to sets of the consumed versions of objects at ℓ to the points-to sets of yielded versions of objects at the same instruction instead of propagating from the IN set to the OUT set of ℓ. It performs strong updates by interacting with the *kill* function in the same way as the standard approach; if an object is a singleton, the consumed version of the object's points-to set is killed and not propagated to the yielded version of the object's points-to set. Finally the [$\mathcal{A}$-PROP]$^{\text{F}}$ rule propagates points-to sets between nodes. Given $\ell' \xrightarrow{o} \ell$, instead of propagating from the IN set or OUT set of ℓ' (depending on what type of instruction ℓ' is) to the IN set of ℓ, for o, it includes the points-to set of the yielded version of o at ℓ' in the points-to set of the consumed version of o at ℓ. Since many nodes may consume and yield the same versions, propagation occurs far less often than in SFS and we save space by storing fewer points-to sets. The [CALL]$^{\text{F}}$ and [RET]$^{\text{F}}$ rules copy the value of actual arguments to formal arguments and return values to pointers, respectively. Using a pointer q as the called function allows for on-the-fly call graph resolution. As shown in gray, if the instructions are annotated with χ/μ, they produce new edges if they did not already exist.

The remainder of the analysis works in the exact same way as SFS. The [ADDR]$^{\text{F}}$ rule inserts a newly allocated object in the left-hand side pointer's points-to set. The [ϕ]$^{\text{F}}$ and [CAST]$^{\text{F}}$ rules add the right-hand side pointer's or pointers' points-to set(s) to the left-hand side pointer's points-to set. The [FIELD$*$]$^{\text{F}}$ rules give the analysis field-sensitivity by inserting a field object (offset into another object) in the left-hand side pointer's points-to set. We use the [FIELD-ADD]$^{\text{F}}$ rule to avoid creating field objects from field objects as it is easier to reason about and implement the analysis with objects like $\hat{o}.f_{i+j}$ rather than $\hat{o}.f_i.f_j$.

E. Correctness

This section discusses how our approach produces the same results as SFS. Intuitively, we aim to have global points-to sets for each o numbering less than the points-to sets of o in IN/OUT sets (or numbering the same, in the unrealistic theoretical worst case). Our approach treats multiple points-to sets as one when it can soundly determine that in they would be equivalent in SFS.

The number of versions for an object introduced in the prelabelling phase is the *minimum* number of points-to sets we will store for that object, thus we try to minimise the number introduced. We introduce a new version for each STORE to yield because we cannot determine whether the consumed version of an object at a STORE would have the same points-to set as the yielded version without the flow-sensitive analysis. In other words, we do not know what the STORE instruction will actually do. We also introduce a new version for δ nodes to consume to soundly handle on-the-fly call graph construction. This is also because we do not know if δ nodes will have any new incoming indirect edges (and from where) until we perform the flow-sensitive analysis.

Then in the meld labelling phase, we propagate versions across and within nodes. The meld operator is specifically designed to mimic the set union operator which is used in inclusion-based points-to analysis (a constraint $pt(p) \subseteq pt(q)$ translates to $pt(q) = pt(q) \cup pt(p)$). Like the meld operator,

$$\texttt{[ADDR]}^{\texttt{F}}\ \frac{\ell : p = alloc_{\hat{o}}}{\hat{o} \in pt(p)} \qquad \texttt{[}\phi\texttt{]}^{\texttt{F}}\ \frac{\ell : p = \phi(q, r)}{p(q) \cup p(r) \subseteq p(p)} \qquad \texttt{[CAST]}^{\texttt{F}}\ \frac{\ell : p = (t)\, q}{p(q) \subseteq p(p)}$$

$$\texttt{[LOAD]}^{\texttt{F}}\ \frac{\ell : p = *q \quad o \in pt(q) \quad c = c_\ell(o)}{pt_c(o) \subseteq pt(p)} \qquad \texttt{[STORE]}^{\texttt{F}}\ \frac{\ell : *p = q \quad o \in pt(p) \quad y = y_\ell(o)}{pt(q) \subseteq pt_y(o)}$$

$$\texttt{[SU/WU]}^{\texttt{F}}\ \frac{\ell : *p = \_ \quad o \in \mathcal{A} \setminus \mathit{kill}(\ell) \quad y = y_\ell(o) \quad c = c_\ell(o)}{pt_c(o) \subseteq pt_y(o)} \qquad \texttt{[}\mathcal{A}\texttt{-PROP]}^{\texttt{F}}\ \frac{\ell' \xrightarrow{o} \ell \quad \kappa_s = \mathcal{Y}_{\ell'}(o) \quad \kappa_d = \mathcal{C}_\ell(o)}{pt_{\kappa_s}(o) \subseteq pt_{\kappa_d}(o)}$$

$$\texttt{[FIELD]}^{\texttt{F}}\ \frac{\ell : p = \&q \rightarrow f_i \quad \hat{o} \in pt(q)}{\hat{o}.f_i \in pt(p)} \qquad \texttt{[FIELD-ADD]}^{\texttt{F}}\ \frac{\ell : p = \&q \rightarrow f_j \quad \hat{o}.f_i \in pt(q)}{\hat{o}.f_{i+j} \in pt(p)}$$

$$\texttt{[CALL]}^{\texttt{F}}\ \frac{\begin{array}{c}\ell : \_ = q(\ldots, r, \ldots)\ \mu(o) \quad o_{\mathit{fun}} \in pt(q) \\ \ell' : \mathit{fun}(\ldots, r', \ldots)\ o = \chi(o)\end{array}}{pt(r) \subseteq pt(r') \quad \ell \xrightarrow{o} \ell'} \qquad \texttt{[RET]}^{\texttt{F}}\ \frac{\begin{array}{c}\ell : p = q(\ldots)\ o = \chi(o) \\ o_{\mathit{fun}} \in pt(q) \quad \ell' : ret_{\mathit{fun}} p'\ \mu(o)\end{array}}{pt(p') \subseteq pt(p) \quad \ell' \xrightarrow{o} \ell}$$

$$\mathit{kill}(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \in \mathcal{SN} \\ \mathcal{A} & \text{if } pt(p) \equiv \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

Fig. 10: Inference rules for the new main-phase flow-sensitive analysis.

$\cup$ is commutative ($\{o_1\} \cup \{o_2\} = \{o_2\} \cup \{o_1\} = \{o_1, o_2\}$), associative ($(\{o_1\} \cup \{o_2\}) \cup \{o_3\} = \{o_1\} \cup (\{o_2\} \cup \{o_3\}) = \{o_1, o_2, o_3\}$), idempotent ($\{o\} \cup \{o\} = \{o\}$), and has an identity, the empty set $\varnothing$ ($\{o\} \cup \varnothing = \{o\}$). So when we perform meld labelling, we are performing an extremely lightweight version of the IN/OUT set propagation SFS performs using versions to represent points-to sets and relying on information from the auxiliary analysis. So, instead of propagating the points-to sets of objects from OUT sets to IN sets, and from IN sets to OUT sets internally, we propagate yielded versions to the versions to be consumed, and vice versa internally. Similar to in the prelabelling phase, we may introduce excess versions. For example, if ℓ consumed version $\kappa_1 \diamond \kappa_2$, $pt_{|\ell}(o)$ would actually be made up of the union of at least two points-to sets. However, κ_1 may be sufficient for ℓ to consume if the points-to set represented by κ_2 is a subset of the points-to set represented by κ_1. We would not know until the flow-sensitive analysis, so we assume the worst to maintain soundness.

Since our pre-analysis assumes the worst when it requires flow-sensitive points-to analysis results (the pessimistic prelabelling and the excess versions), it soundly ensures our analysis only treats multiple points-to sets as one when they would be the same in SFS (Equations 1, 2, and 3) and so produces the same results as SFS.

V. Evaluation

This section describes our experiments comparing the performance and memory usage of our approach (VSFS) against SFS. Open source pointer analysis framework SVF [23] contains an implementation of SFS as described in Section V of the original paper [11], and we implement our new versioned analysis alongside it for comparison. The auxiliary analysis used to build the SVFG is Andersen's analysis boosted by wave propagation [19]. We use LLVM's `SparseBitVector` data structure and the bitwise operator defined upon it during the versioning phase for the labels/versions and the meld operator, respectively. During the versioning phase, as an optimisation, analogous to SFS, if a node consume what it yields, we store the version once and do not explicitly perform the $\texttt{[INTERNAL]}^{\texttt{V}}$ rule. To make the analysis cheaper, after the versioning phase, we use unsigned integers to represent versions, converting equivalent `SparseBitVector`s to the same integer, and we explicitly store all consumed and yielded versions for simplicity's sake (though this incurs a slight time and memory penalty).

We use 15 open-source programs to benchmark and compare our analysis as listed in Table II. Of these benchmarks, ninja, astyle, and hyriseConsole are written in C++, and the remainder are written in C. We compile the benchmarks using Clang 10 (`O3` flag) through Whole Program LLVM (WLLVM)[2] to generate bitcode files. For each benchmark, the number of lines of code (LOC) listed in Table II is obtained by counting the number of lines of code in every source file listed in the LLVM debug information (excluding system headers). Finally, we strip debug information to get a better view of the bitcode size.

We run Andersen's analysis (also provided by SVF), SFS, and VSFS 5 times on each benchmark and record the average running times in seconds and memory usage (maximum resident set size) in gigabytes in Table III. We give each run a maximum of 12 hours to complete (no benchmark exhausted this) and 120 GB of memory (lynx could not be analysed by SFS within this limit). For SFS and VSFS, we omit the running time of the required auxiliary analysis, memory SSA construction, and SVFG construction, and only show the time taken by the main phase of the analysis (and versioning time for VSFS). Memory usage statistics do not make this

[2]https://github.com/travitch/whole-program-llvm

TABLE II: Benchmarks used to evaluate VSFS. For each benchmark, we list the number of lines of code (**LOC**), bitcode size (**KiB**) after optimisation (`O3`) and debug information is stripped, the number of nodes, and direct (**D. Edge**) and indirect (**I. Edge**) edges in the SVFG, the number of top-level and address-taken variables, and a brief description.

Bench.	LOC	Size	SVFG			# Variables		Description
		(KiB)	# Nodes	# D. Edges	# I. Edges	Top-Level	Address-Taken	
du	27704	376	27192	16229	367638	35314	1705	Disk usage (GNU)
ninja	8702	576	46776	31594	392071	46075	2681	Build system
bake	20548	580	78426	34038	1833144	40668	3097	Build system
dpkg	21934	612	77793	32117	379719	38843	2884	Package manager
nano	27564	828	84637	42463	1911821	72663	2637	Text editor
i3	22895	1016	100009	54532	268501	67716	3237	Window manager
psql	47444	1120	92421	62659	495139	67310	3513	PostgreSQL frontend
janet	56500	1172	128838	54625	2491254	99753	3154	Janet compiler
astyle	16715	1560	156322	86563	8616092	125992	8712	Code formatter
tmux	48205	1656	154683	101518	8302942	123889	6009	Terminal multiplexer
mruby	50807	1732	118992	101613	1145376	149346	2381	Ruby interpreter
mutt	64046	1820	297971	106211	11044405	147712	6939	Terminal email client
bash	102319	2200	309236	100660	29697360	187307	6217	UNIX shell
lynx	138182	3800	554940	172969	35091128	232829	7802	Terminal web browser
hyriseConsole	37300	9524	627834	437230	6377348	607274	37339	Hyrise DB frontend

distinction and include everything. Time is measured with C's `clock` function and memory is measured with GNU's `time` program. The average in the bottom row is calculated using the geometric mean, and non-existent data (running time for SFS for lynx, specifically) is ignored. All experiments were conducted on a machine running 64-bit Ubuntu 18.04.2 LTS with an Intel Xeon Gold 6132 processor at 2.60 GHz and 128 GB of memory.

Overall, we see that since VSFS sees improvement in both time and memory, and VSFS targets the excessive propagation and storage of IN and OUT sets, a considerable amount of SFS overhead is spent on the propagation and storage of them.

A. Time

We find that our analysis always performs better than SFS. The versioning process is always cheap. For benchmarks which are easy for VSFS to analyse, it can be a large percentage of total running time (see mruby and bake, for example), but in our tests this never caused the overall analysis to be slower than SFS. As benchmarks take longer to analyse, versioning time becomes more and more negligible. For example, VSFS's main phase takes almost 3 and a half hours to analyse lynx, but only takes less than a minute to perform versioning.

Benchmarks like bake (the most extreme case with a 26.22× improvement), astyle, hyriseConsole, ninja, and janet benefit greatly from the reduced propagation required and the versioning time is negligible. Benchmarks dpkg and bash see little speedup, 1.74× and 1.46×, respectively. Programs like dpkg are not really targets of our analysis being that they can be analysed by SFS easily, and bash still sees some improvement and no regression. The remaining benchmarks benefit with speedups in the range of 2.43× to 6.27×. The geometric mean of speedups is 5.31× which we believe is an indication of success, especially considering this includes benchmarks which did not show a drastic performance improvement (dpkg and bash). If we exclude all benchmarks which take less than 30 seconds for SFS to analyse (which are not the target of our work), du, dpkg, i3, psql, and mruby, we obtain a geometric mean speedup of 6.74×. The time for lynx is missing because it ran out of memory (in what appears to be) early in the analysis.

B. Memory usage

We find that our analysis improves memory usage compared to SFS for all benchmarks except dpkg, i3, and mruby, where it used about the same amount of memory as SFS. In these instances, the analysis is small enough that the cheap memory overhead from versioning accounts for a significant percentage of overall memory usage. On the other hand, SFS runs out of memory analysing lynx meaning that it required more than 120 GB while VSFS used less than 22 GB. At best, for our benchmarks, our approach reduced memory by 5.46× or more, and offered significant improvements in analysing lynx, bash, and astyle. Overall our analysis sees a (geometric) mean improvement of at least 2.11× meaning VSFS uses half the memory SFS does on average whilst improving performance.

The memory overhead present by versioning grows much slower than the main phase analysis, similar to the time overhead. We believe that overhead could perhaps be further reduced by designing a data structure specifically catered to versioning rather than using one off-the-shelf (LLVM's `SparseBitVector`) which perhaps may use a completely different meld operator.

VI. Related work

To improve performance, flow-sensitive points-to analysis has seen its sparsity improved over time. Earlier approaches [4], [12], [16] focused on reducing the ICFG by removing irrelevant (to the analysis) nodes, rather than introducing a new data structure to perform the analysis on. Hardekopf and Lin [10] introduce a semi-sparse analysis which makes use of the partial SSA form [24] to perform the

TABLE III: Time, in seconds, and memory usage (maximum resident set size), in gigabytes, of Andersen's analysis, SFS (main phase time only), and VSFS. Time statistics are split into 3 for VSFS: time to version objects, time to perform the analysis using versions, and the sum of those two times which is then used for comparison. The last two columns show how many times faster (or slower) our approach is compared to SFS and the reduction (or increase) in memory usage of our approach compared to SFS. OOM means a benchmark was unable to complete because it exhausted memory resources.

Benchmark	Andersen's		SFS		VSFS				Time diff.	Mem. diff.
	Time	Mem.	Time	Mem.	Versioning	Main phase	Total time	Mem.		
du	1.15	0.19	21.43	2.24	0.27	3.94	4.20	1.52	5.10×	1.48×
ninja	1.15	0.22	60.97	2.37	0.24	4.01	4.25	1.47	14.35×	1.61×
bake	1.06	0.19	60.90	3.12	0.92	1.40	2.32	1.69	26.22×	1.84×
dpkg	0.60	0.16	3.34	1.41	0.37	1.55	1.92	1.41	1.74×	1.00×
nano	2.91	0.41	74.90	5.59	1.57	16.69	18.26	2.16	4.10×	2.58×
i3	1.14	0.27	3.28	1.55	0.32	0.81	1.13	1.52	2.90×	1.02×
psql	0.99	0.28	8.04	1.82	0.38	0.90	1.28	1.56	6.27×	1.17×
janet	2.93	0.43	116.76	7.09	1.35	8.32	9.67	2.34	12.07×	3.03×
astyle	20.18	1.14	12437.38	85.41	7.76	1199.55	1207.31	19.59	10.30×	4.36×
tmux	22.78	1.19	483.14	12.75	14.50	167.25	181.75	7.79	2.66×	1.64×
mruby	7.65	0.64	16.78	2.75	1.97	3.45	5.41	2.62	3.10×	1.05×
mutt	13.74	1.05	981.53	26.59	15.84	388.72	404.56	8.13	2.43×	3.27×
bash	19.54	1.48	2160.97	77.70	24.50	1458.23	1482.73	15.44	1.46×	5.03×
lynx	57.18	1.86	OOM	OOM	58.80	11947.68	12006.48	21.96	–	≥5.46×
hyriseConsole	18.18	2.76	701.93	18.42	4.21	39.95	44.16	6.85	15.89×	2.69×
Average									5.31×	≥2.11×

analysis sparsely on top-level pointers, whilst performing the analysis in the same way as before for address-taken objects.

Several recent works use the idea of a staged analysis [7], [18], [26], for example to perform a sparse def-use analysis for top-level variables, or to perform a sparse analysis upon all variables [11] by building the memory SSA form [5], [25]. This is a form of multiple-object sparsity in that all object points-to sets are no longer propagated together based on control-flow but propagated on an object-to-object basis based on the sparse value-flows. We build upon this work to improve single-object sparsity, or sparsity within the propagation of a single object's points-to sets to different program points. Our approach efficiently and effectively determines duplicate points-to sets at different program points for a single object and treats them as one. Hardekopf and Lin also introduce "points-to graph equivalence" in their work on semi-sparse analysis [10] however, their approach, while similar to ours, is imprecise in that they conservatively determine entire IN and OUT which cannot realise the same opportunities to collapse objects' points-to sets as in our single object sparsity. Their pre-analysis is performed on the sparse evaluation graph, whereas we perform our pre-analysis on the SVFG. We have not compared to this optimisation since we do not know of a modern implementation of it.

In Section 3.4 of their work [16], Lhoták and Chung sparsely allocate instruction labels on the ICFG such that propagation is skipped where they can determine that address-taken objects' points-to sets will not change during the analysis (non-STORE instructions and non-merge points of control-flow). Our work differs in that they perform their analysis on the ICFG and their sparse allocation is not on an object-to-object basis but upon all objects for each label allocation. Their approach also *always* allocates separate labels for separate merge points whereas our approach can sometimes determine when merge points produce a points-to set that is being merged and can be reused. Their label allocation is faster than our versioning, but our versioning is more effective (yet still performant), and this is crucial for larger programs since constraint solving grows much faster than either label allocation or versioning.

Our analysis is an instance of offline variable substitution [20] in that we collapse multiple equivalent variables (in our case, variable/location pairs) before the main phase points-to analysis. Variable substitution and similar techniques have been applied successfully before [9], [10], [16], [22].

VII. CONCLUSION

This paper presents an object versioned flow-sensitive points-to analysis, an improvement over the state-of-the-art staged flow-sensitive points-to analysis (SFS) in both time and space. We use a prelabelling extension, meld labelling, to version objects such that SVFG nodes can in many instances share the same points-to sets for an object. We achieve finer grained single-object sparsity than SFS giving us an average speedup of 5.31× (up to 26.22×) and an average memory reduction of over 2.11× (up to 5.46×).

ACKNOWLEDGEMENT

This research is supported by Australian Research Grant DP210101348. Additionally, the first author is supported by a PhD scholarship funded by CSIRO's Data61.

APPENDIX
ARTIFACT

A. Abstract

This artifact provides a Docker image containing our implementation of VSFS. VSFS is built upon SVF and LLVM

10. The image also includes SVF's implementation of SFS. A script is included to easily compare the running time and memory usage of VSFS and SFS on a variety of open source benchmark programs and reproduce Table II and Table III.

B. Artifact Checklist

- **Algorithm:** Flow-sensitive points-to analysis utilising versioned objects.
- **Program:** Analysis implemented in SVF. Benchmarks are a variety of open source programs.
- **Binary:** SVF and benchmarks are built and included in the Docker image. They can be rebuilt within the image.
- **Runtime environment:** Docker.
- **Hardware:** System with at least 120 GB of available memory to run all 15 tests, with at least 32 GB of available memory to run 11/15 tests, or with at least 8 GB of available memory to run 8/15 tests.
- **Metrics:** Analysis time and memory usage.
- **Output:** Tables (text) matching Table II and Table III.
- **Experiments:** Running a script and comparing output to Table II and Table III. The "Time diff." and "Mem. diff." columns should be similar.
- **How much disk space required?** The compressed Docker image is 1.9 GB. It is 6.2 GB once loaded.
- **How much time is needed to prepare workflow?** Very little time, depending on network connection.
- **How much time is needed to complete experiments?** One run for all benchmarks takes approximately 10 hours on a Xeon Gold 6132 CPU.
- **Publicly available?:** Yes.
- **Code licenses?** The GPL.
- **Archived (DOI)?** https://doi.org/10.6084/m9.figshare.13269662.v1

C. Description

1) Distribution: The Docker image can be downloaded from https://doi.org/10.6084/m9.figshare.13269662.v1. For information on running VSFS outside the benchmarking environment, visit https://github.com/SVF-tools/SVF/wiki/VSFS.

2) Hardware dependencies: The analyses are all single-threaded so multiple cores will not have any effect, however they can still be computationally intensive. In terms of memory, to run the analyses on the full set of benchmarks (15/15), a machine with 120 GB of available memory is required. A subset of benchmarks can be evaluated on machines with less memory. To run the analyses on 11/15 of the benchmarks, a machine with at least 32 GB of available memory is required. To run the analyses on 8/15 of the benchmarks, a machine with at least 8 GB of available memory is required.

3) Software dependencies: Docker. We have tested the image on Docker version 19.03.6 on an Ubuntu 18.04 machine.

D. Installation

Download the Docker image (`vsfs.tar.gz`) from the URL in Appendix C1. Then, load the image,

```
$ docker load -i vsfs.tar.gz
```

Finally, run the image,

```
$ docker run -it vsfs bash
```

You are now in a full Linux environment.

E. Experiment workflow

First, change into the benchmark directory,

```
$ cd $HOME/bench
```

Here, the `bench.sh` script runs the analyses and tables them automatically using the `table.awk` script. The `bench.sh` script takes 4 arguments followed by at least 1 more:

1) The SVF binary (in the image it is `wpa` and already in the path).
2) The number of runs for each analysis.
3) The maximum time limit (in seconds).
4) The maximum memory limit (in bytes).
5) The bitcode files to analyse.

For the bitcode file arguments we have created 3 environment variables: `W_8GB`, `W_32GB`, and `W_120GB`. These variables contain the names of the bitcode files (separated by a space) which can be analysed by a system with 8 GB, 32 GB, and 120 GB, respectively. For example, `$W_120GB` expands to `du.bc ninja.bc bake.bc dpkg.bc ...`, i.e. the full list of benchmarks.

The results in the paper were produced by,

```
$ ./bench.sh wpa 5 43200 120000000000 \
  $W_120GB
```

When running on a system with only 8 GB of free memory, for example, the following is more suitable to produce a portion (about half) of the table,

```
$ ./bench.sh wpa 1 43200 8000000000 $W_8GB
```

We recommend performing only a single run for evaluation purposes (by setting the second argument to `1`) to save time. Progress is printed as the benchmark currently being analysed and with which analysis (`ander` is Andersen's analysis, `vfspta` is VSFS, and `fspta` is SFS).

The benchmarks can be rebuilt (note: this may require up to 15 GB more disk space). First, change into the bitcode directory,

```
$ cd $HOME/bench/bitcode
```

Remove the existing source and bitcode files,

```
$ ./clean.sh
```

Run the build script (note: source files are downloaded from the network),

```
$ ./build.sh
```

Finally, move the produced bitcode and LLVM IR files to `$HOME/bench`,

```
$ mv *.bc *.ll $HOME/bench
```

F. Evaluation and expected result

The aforementioned `bench.sh` will reproduce Tables II and III. Most importantly, the "Time diff." and "Mem. diff." columns should be similar with a 0.5× tolerance for benchmarks with little difference, and 1.5× for benchmarks with greater difference (more than 10×).

G. Experiment customisation

Other programs can be used in place of the provided benchmarks. To do so, build the program with WLLVM (`CC=wllvm CXX=wllvm++ build-command`) in `$HOME/bench/bitcode`, extract the bitcode file (`extract-bc program`), copy the bitcode file to `$HOME/bench`, and use the resulting bitcode file as an argument to `bench.sh`.

To also count lines of code, the program must be compiled with the debug flag (`-g`). The resulting bitcode file (`program.bc`) must be dissassembled to LLVM IR (`llvm-dis program.bc`) and also copied along with the bitcode file to `$HOME/bench` in the form `program.dbg.ll`. It may also be a good idea to optimise the bitcode, and to strip the bitcode file of debugging information to get a better reading of the size (`opt -O3 --strip-debug program.bc`).

For example, if we were to build web log analyser GoAccess for analysis,

```
$ cd $HOME/bench/bitcode
$ wget https://tar.goaccess.io/\
  goaccess-1.4.3.tar.gz
$ tar -xf goaccess-1.4.3.tar.gz
$ cd goaccess-1.4.3
$ CC=wllvm CFLAGS="-g -O0" ./configure
$ make
$ extract-bc -o goaccess.dbg.bc goaccess
$ llvm-dis goaccess.dbg.bc
$ opt -O3 --strip-debug goaccess.dbg.bc \
  -o goaccess.bc
$ mv goaccess.bc goaccess.dbg.ll \
  $HOME/bench
```

Finally, the `bench.sh` script can be run as described in Appendix E with `goaccess.bc` as the fifth argument.

H. Source code

SVF's source code is available in `$HOME/svf`. Most of our VSFS code resides in two files (relative to `$HOME/svf`):

```
include/WPA/VersionedFlowSensitive.h
lib/WPA/VersionedFlowSensitive.cpp
```

SVF can also be recompiled if modified. First, change into the build directory,

```
$ cd $HOME/svf/Release-build
```

Then, run the build command.

```
$ make
```

VSFS's source code has been merged into mainline SVF and is actively maintained. SVF can be found at https://github.com/SVF-tools/SVF.

References

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

[2] M. J. Bach. *The design of the UNIX operating system*. Prentice-Hall International, USA, 1986.

[3] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *International Static Analysis Symposium*, SAS '16, pages 84–104, Germany, 2016. Springer.

[4] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 55–66, USA, 1991. ACM.

[5] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 253–267, Germany, 1996. Springer.

[6] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 106–117, USA, 1998. ACM.

[7] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2), May 2008.

[8] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Computer Aided Verification*, pages 343–361, Switzerland, 2015. Springer.

[9] B. Hardekopf and C. Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *International Static Analysis Symposium*, SAS '07, pages 265–280, Germany, 2007. Springer.

[10] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 226–238, USA, 2009. ACM.

[11] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 289–298, USA, 2011. IEEE Computer Society.

[12] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *International Static Analysis Symposium*, SAS '07, pages 57–81, Germany, 1998. Springer.

[13] J. Kuderski, J. A. Navas, and A. Gurfinkel. Unification-based pointer analysis without oversharing. In *2019 Formal Methods in Computer Aided Design*, FMCAD '19, pages 37–45, USA, 2019. IEEE Computer Society.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[15] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 278–289, USA, 2007. ACM.

[16] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, USA, 2011. ACM.

[17] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 1007–1024, USA, 2017. USENIX Association.

[18] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 229—238, USA, 2012. ACM.

[19] F. M. Q. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 126–135, USA, 2009. IEEE Computer Society.

[20] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 47—-56, USA, 2000. ACM.

[21] P. D. Schubert, B. Hermann, and E. Bodden. PhASAR: an inter-procedural static analysis framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '19, pages 393–410, Germany, 2019. Springer.

[22] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *Proceedings of the 28th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, pages 253—-270, USA, 2013. ACM.

[23] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, USA, 2016. ACM.

[24] Y. Sui, H. Yan, Z. Zheng, Y. Zhang, and J. Xue. Parallel construction of interprocedural memory ssa form. *Journal of Systems and Software*, 146:186–195, 2018.

[25] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 254––264, USA, 2012. ACM.

[26] A. Tavares, B. Boissinot, F. Pereira, and F. Rastello. Parameterized construction of program representations for sparse dataflow analyses. In *Proceedings of the 23rd International Conference on Compiler Construction*, pages 18–39, Germany, 2014. Springer.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[28] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 327–337, USA, 2018. ACM.