University of Technology Sydney
School of Computer Science
Faculty of Engineering and Information Technology

Dissertation submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

under principal supervisor **Dr. Yulei Sui** and co-supervisor **Dr. Shiping Chen**

Data Structures for Points-To Analysis

Mohamad Barbar

October 2022

Certificate of Original Authorship

I, Mohamad Barbar, declare that this thesis, is submitted in fulfilment of the requirements for the award of Doctor of Philosophy, in the School of Computer Science, Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: 06/10/2022

Preface

With the end of my candidature, more than 4 years later, it is time to collect my work in one place. I hope it is of use to some.

Firstly, major thanks is due to my family who were patient with me during my candidature. 4 years is not a short period of time.

I also thank my principal supervisor, Dr. Yulei Sui, for the many long hours of discussion on both the big picture and the smallest of details. Certainly, working with Yulei got me started with static analysis, and it is an area I would like to continue hacking on. The opportunity to work on SVF has also been great as my first prolonged exposure to a large codebase where performance really matters. I also thank my co-supervisor, Dr. Shiping Chen, and CSIRO's Data61 as a whole, for supporting my research with the Data61 scholarship.

My thanks also goes to my peers at UTS who kept me company, particularly before moving to work-from-home at the start of the pandemic; Omar, Mingshan, Ayman, Joakim (who also inspired some of the work in Chapter 4), Ibraheem, Yahya, and Akram. I also thank the Program Analysis Group, especially Yanxin and Yuxiang with whom I started around the same time, though it was unfortunate meetings had to move online. I must also say that all my interactions with administration at UTS has been pleasant, and express my thanks in that direction. At CSIRO, I also thank Ejaz for organising seminars allowing for some nice exchange of ideas. Unfortunately, that too was cut short by the pandemic.

Finally, most of all, I give thanks to the All-Wise, the All-Knowing and hope for His forgiveness for my many shortcoming during this candidature.

Mohamad Barbar
Sydney, Australia
October 2022

Abstract

In this dissertation, we present improvements to data structures, and the algorithms upon them, for points-to analysis. Our focus is mainly on flow-sensitive analysis but our techniques can either be applied to other analyses or used in analyses which combine flow-sensitivity with other sensitivities. For staged flow-sensitive analysis (SFS), we introduce a pre-analysis (meld versioning) where we determine when it is possible to reuse the points-to sets of individual address-taken variables at different program points, then perform the main analysis using this information. Meld versioning is also amenable to parallelisation with minimal effort. For points-to sets, we introduce an improved bit-vector stripping both leading and trailing zero-words, then use that to aid in improving the object-to-identifier mapping required to use bit-vectors as points-to sets. We frame this as an integer programming problem, yielding an optimal solution but with impractical performance, and so we develop a more approximate (yet extremely fast) method based on hierarchical clustering. We also describe hash consing and memoisation, along with some optimisations which would otherwise be impractical, for points-to sets. We have implemented our techniques in open source points-to analysis framework SVF, and upon evaluating with 12 open source programs, we find, on average, a speedup of almost $6\times$ and a reduction in memory usage of more than $3.97\times$ when compared with a baseline SFS.

Publications

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-Sensitive Type-Based Heap Cloning. In 34th European Conference on Object-Oriented Programming (ECOOP '18, Vol. 166). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 24:1–24:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.24

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21). IEEE Computer Society, USA, 222–235. https://doi.org/10.1109/CGO51591.2021.9370334

Mohamad Barbar and Yulei Sui. 2021. Hash Consed Points-To Sets. In International Static Analysis Symposium (SAS '21). Springer, Germany, 25–48. https://doi.org/10.1007/978-3-030-88806-0_2

Mohamad Barbar and Yulei Sui. 2021. Compacting Points-to Sets through Object Clustering. Proceedings of the ACM on Programming Languages 5, OOPSLA, Article 159 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485547

Colophon

This dissertation was created using XeLaTeX. The Maggi Memoir Thesis template, originally by Federico Maggi and later modified by Vel from `LaTeXTemplates.com`, is used as a base after some slight modifications. The body is set 10pt with Linux Libertine O primarily. Linux Biolinum O, Latin Modern Math, and Latin Modern Mono make occasional appearances. Graphs, in the graph theoretical sense, are drawn with TikZ/PGF, tables use the `booktabs` package, and the single line graph in this dissertation is drawn with Matplotlib (DejaVu Sans on the axes).

Contents

List of Figures

List of Tables

List of Acronyms

BDD binary decision diagram 47

DUG def-use graph 15

GLLVM Whole Program LLVM in Go 18

ICFG interprocedural control-flow graph 14

IP integer programming 57

IR intermediate representation 5

ISA instruction set architecture 47

OOM out of memory 20

PAG pointer assignment graph 18

SFS staged flow-sensitive analysis xvi

SSA static single assignment 6

STL Standard Template Library 18

SVFG sparse value-flow graph xvi

VSFS versioned staged flow-sensitive analysis 3

Introduction 1

Increasingly, due to its cost, flexibility, and availability, software has been replacing or complementing systems traditionally developed or performed by others means such as electronics, mechanical tools, and humans. With such an ever growing reliance, software quality (along dimensions such as performance, reliability, and security) is non-negotiable. Unfortunately, as programs grow and become more complex, it becomes increasingly difficult for programmers to write software that performs well and is free of defects. Program analysis promises to help mitigate both these difficulties and has thus become more attractive.

Program analysis may be static or dynamic, that which does not run the program under analysis and that which does. Static analysis may be further divided into a myriad of analyses, of them, pointer analysis. Even further, pointer analysis may be divided into alias analysis that answers queries such as "May/must pointers p and q point to the same memory location at runtime?" and points-to analysis that answers queries such as "Which memory locations may pointer p point to at runtime?". In this dissertation, we focus on whole program points-to analysis where we analyse the entire program at hand though some techniques can also be applied to other forms such as demand-driven analysis.

The result of points-to analysis is a points-to set containing abstract memory objects (representing a set of actual runtime memory locations) for each pointer. Ideally, an analysis is both sound, in that no objects are missing from

points-to sets, and precise, in that no spurious objects appear in points-to sets, but this is exceedingly difficult. Typically, we build off of a sound base, subject to some *soundiness* [Livshits et al., 2015], and gradually add precision as we discover new ways to scale the resulting more expensive analyses.

Various sensitivities form the main dimensions of precision in points-to analysis. For example, an analysis could be field-sensitive, meaning fields of objects are treated separately to the objects they are derived from, context-sensitive, meaning calling context is considered, flow-sensitive, meaning the control-flow of the program is taken into account rather than ignored, and more. In this dissertation, we largely focus on flow-sensitive analysis but the same techniques can either be applied to analyses of other sensitivities or used when mixing flow-sensitivity with other sensitivities.

Points-to analysis can support a variety of clients including memory error detection [Livshits and Lam, 2003; Yan et al., 2018], concurrency bug detection [Chen et al., 2020; Pratikakis et al., 2006], typestate verification [Fink et al., 2008; Wang et al., 2020], control-flow integrity [Fan et al., 2017; Farkhani et al., 2018], symbolic execution [Trabish et al., 2018, 2020], code embedding [Sui et al., 2020; Cheng et al., 2021], and compiler optimisation [Le et al., 2005; Hirzel et al., 2007]. With such a wide ranging need for points-to analysis, performance in both time and space is paramount. Imprecise analyses are generally performant enough but these clients sometimes benefit from the precision afforded by less scalable, more precise analyses [Hind and Pioli, 1998; Ghiya et al., 2001; Guyer and Lin, 2005; Lhoták and Hendren, 2006; Fink et al., 2008; Chang et al., 2008].

Modelling aspects of program execution more precisely requires more elaborate data structures that may be costly to store and operate on (e.g., a flow-sensitive analysis may require a control-flow graph). This is in addition to the increased number of points-to sets introduced and previously unneeded bookkeeping now required. Thus, in this dissertation, we present new methods improving the data structures, and algorithms upon them, used to perform points-to analysis—with no loss of sound(i)ness or precision—with the aim of reducing both the time and space needed to perform points-to analysis.

First, after some background on points-to analysis, we discuss improvements to staged flow-sensitive analysis to reduce the number of points-to sets stored and points-to set propagations by performing a pre-analysis, meld versioning. Meld versioning versions address-taken objects allowing for multiple points-to sets of individual such objects to be effectively merged. We

also introduce parallelisation into the pre-analysis at almost no implementation complexity. Then, we turn our focus to more fundamental data structures: the bare points-to sets themselves. We discuss a bit-vector representation, the core bit-vector, which strips both trailing and leading zero-words and then optimise the object-to-identifier mapping required to use bit-vectors as points-to sets. First, we formulate this problem as one solved by integer programming, and then, since it is not performant enough, apply a more approximate solution that is still effective, but extremely fast, built upon hierarchical clustering. Before finally going over future work and concluding, we discuss the application of hash consing and memoisation to the points-to sets and their operations, including some optimisations which would otherwise be too expensive without hash consing.

1.1 Contributions

What follows are the contributions this dissertation makes.

In Chapter 3 [Barbar et al., 2021]:

- The versioning of address-taken objects to group points-to sets of individual such objects found to be equivalent through a pre-analysis (meld versioning) in *versioned staged flow-sensitive analysis* (VSFS), a new flow-sensitive analysis building on *staged flow-sensitive analysis* (SFS), which uses those versions.
- Optimisations to the versioning phase of VSFS, including simple to implement scalable parallelisation (previously unpublished).

In Chapter 4 [Barbar and Sui, 2021a]:

- The core bit-vector, a bit-vector representation that strips both leading and trailing zero-words.
- An integer programming formulation to produce an optimal object-to-identifier mapping for the auxiliary analysis of a staged analysis which in turn can be used for the main phase of such an analysis.
- A more approximate, yet far more efficient, method of achieving a good object-to-identifier mapping utilising hierarchical clustering.

In Chapter 5 [Barbar and Sui, 2021b]:

- A description of hash consing and memoisation for points-to sets and their operations, along with some optimisations which would otherwise be inefficient.

And overall:

- An implementation of VSFS and associated versioning optimisations, the core bit-vector, the clustering-driven approach to producing a good object-to-identifier mapping, and hash consing and memoisation of points-to sets (and associated optimisations), in open source points-to analysis framework SVF.
  Available upstream at `https://github.com/SVF-Tools/SVF`.
- An evaluation of all that which has been implemented on 12 open source programs from a variety of domains with an overall finding that, against SFS, we see an average speedup of 5.92× and an average reduction in memory usage of more than 3.97× (Table 6.1).

Background 2

2.1 Program Representation

To analyse programs, we must define what a program is. We are interested in analysing C and C++ programs and like many previous works on points-to analysis [Hardekopf and Lin, 2011; Sui and Xue, 2016a; Balatsouras and Smaragdakis, 2016], we build our analyses atop the LLVM *intermediate representation* (IR) [Lattner and Adve, 2004]. The LLVM IR is large and complicated, so for the sake of defining and describing our analyses, we work on a simpler set of domains and instructions as presented in Figures 2.1 and 2.2.

Figure 2.1 shows the domains upon which our instructions will operate on. The set of all variables $\mathcal{V}$ is separated into two subsets: $\mathcal{A} = \mathcal{O} \cup \mathcal{F}$ which

ℓ	$\in$	$\mathcal{L}$	instruction labels
i, j, k	$\in$	$\mathcal{C}$	constants
s	$\in$	$\mathcal{S}$	stack virtual registers
g	$\in$	$\mathcal{G}$	global variables
p, q, r, x, y, z	$\in$	$\mathcal{P} = \mathcal{S} \cup \mathcal{G}$	top-level variables
$\hat{o}$	$\in$	$\mathcal{O}$	abstract objects
$\hat{o}.f_k$	$\in$	$\mathcal{F}$	abstract field objects
o, a, b	$\in$	$\mathcal{A} = \mathcal{O} \cup \mathcal{F}$	address-taken variables
v	$\in$	$\mathcal{V} = \mathcal{P} \cup \mathcal{A}$	program variables
t	$\in$	$\mathcal{T}$	types

FIGURE 2.1: Domains our analysed programs operate on.

Alloc	$p = alloc_{\hat{o}}$
Phi	$p = \phi(q, r)$
MemPhi	$o = \phi(a, b)$
Cast	$p = (t)\, q$
Field	$p = \&q \rightarrow f_k$
Load	$p = *q$
Store	$*p = q$
Call	$p = q(r_1, \ldots, r_n)$
FunEntry	$fun(r_1, \ldots, r_n)$
FunExit	$ret_{fun}\, p$

Figure 2.2: Instructions we define our analyses upon.

contains all abstract memory objects and their fields (*address-taken variables*), and $\mathcal{P} = \mathcal{S} \cup \mathcal{G}$ which contains all *top-level variables*, namely stack virtual registers and named global variables (in LLVM, such variables would be prefixed by % and @, respectively). Top-level variables, or those in $\mathcal{P}$, are explicit and accessed directly by name, whereas address-taken variables, or those in $\mathcal{A}$, are implicit and are accessed indirectly through top-level variables at certain instructions.

For address-taken variables, we use o to represent all objects and $\hat{o}$ to represent non-field objects. This is to later simplify the handling of field objects. We also need constants for field objects, and these are sourced from $\mathcal{C}$. Though this dissertation does not define any type-based analyses, the Cast instruction below requires a type which would come from $\mathcal{T}$. Finally, as we move on to our instructions, $\mathcal{L}$ contains labels for instructions such that every instruction is given a label. This is important when we want to perform flow-sensitive analyses.

Following conversion of programs to partial *static single assignment* (SSA) form, our programs are made up of 10 instructions (the MemPhi instruction would not appear yet and is described in Section 2.3.2). Of these 10 instructions, 8 form function bodies:

- Alloc ($p = alloc_{\hat{o}}$) which allocates an object on the stack, globally, or on the heap,
- Phi ($p = \phi(q, r)$) which assigns a value to a top-level pointer at a merge point in the control flow (choosing one of two top-level pointers),
- Cast ($p = (t)\, q$) which casts a pointer to another pointer, essentially acting as a copy instruction,

- FIELD ($p = \&q \rightarrow f_k$) which assigns to a top-level pointer the k-th field of an aggregate object,
- LOAD ($p = *q$) which reads the value of a top-level pointer, assigning it to another,
- STORE ($*p = q$) which writes the value of a top-level pointer to the value of another top-level pointer (i.e., an abstract memory object), and
- CALL ($p = q(r_1, \ldots, r_n)$) which calls a function with the specified arguments (for simplicity we will represent each function call as a call through a function pointer – direct calls such as $p = \mathit{fun}(r_1, \ldots, r_n)$ would be represented as $p = q(r_1, \ldots, r_n)$ where q would point to a single object, that allocated for *fun*, for the entire analysis),

The remaining two instructions help connect calls and returns to their targets. Each function has a single entry instruction, FUNENTRY ($\mathit{fun}(r_1, \ldots, r_n)$), which contains the formal parameters of a function, and a single exit instruction, FUNEXIT ($ret_{\mathit{fun}}\ p$), which contains the return value of a function. While most non-trivial programs will contain multiple return statements in functions, in practice, we can merge them before analysis (the `mergereturn` transformation pass in LLVM, for example).

2.2 Inclusion-Based Points-To Analysis

Many points-to analyses are either *unification-based* [Steensgaard, 1996] or *inclusion-based* [Andersen, 1994]. Generally, fast and imprecise vs. slow and precise is the trade-off presented for the two. Inclusion-based analysis, however, can perform well, and has formed the larger focus of research over the years. We too, in this dissertation, focus exclusively on inclusion-based points-to analysis.

Andersen [1994] was the first to describe inclusion-based points-to analysis, where the *points-to sets* of variables are included within the points-to sets of other variables (rather than being merged/unified). A points-to set is a set associated with each variable (or some extension of each variable) containing abstract memory objects whose concrete or runtime equivalents are determined by the analysis to be possible points-to targets. In other words, if at the end of some analysis, the points-to set of variable v contains a single

abstract memory object o ($pt(v) = \{o\}$), then that analysis has concluded that v, at runtime, *may* point to a concrete object which o abstracts.

We say that a points-to analysis is sound if, for each pointer, every possible runtime points-to target (in the form of abstract memory objects) appears in that pointer's points-to set. In other words, the analysis is over-approximate or has not missed any possibility. Use cases such as compiler optimisation demand conservative results and so are usually more interested in sound analyses. Unfortunately, truly sound points-to analysis is difficult, and most analyses are actually *soundy* [Livshits et al., 2015]. A soundy analysis is one which is sound for the most part except when dealing with some particularly tricky language features such as reflection and integer-to-pointer casts. Depending on the target program and language, not only can it be expensive to handle such features soundly but analysis results may become too over-approximate to the point of uselessness. However, there has been some work on truly sound analyses [Smaragdakis and Kastrinis, 2018] finding that sizeable portions of programs can have truly sound and useful results.

On the other hand, precision indicates an analysis's ability to not include spurious points-to targets in points-to sets, i.e., those which will never have a runtime analogue. Precision is usually yielded by more closely modelling the program to be analysed. Some precision dimensions include:

- Field-sensitivity, which is to consider fields of composite or aggregate objects individually rather than treating them as the aggregate object itself,
- Array-sensitivity, which is akin to field-sensitivity but for arrays only, and is generally handled separately,
- Flow-sensitivity, which is to take into account the flow of control through the analysed program, rather than treat the instructions in a bag-of-words fashion, and
- Context-sensitivity, which is to take into account calling context when analysing a function, rather than analysing functions once (effectively merging calling contexts).

Generally, a more precise analysis is a more expensive one. Usually, we begin with a sound(y) imprecise analysis and gradually introduce precision as new techniques for performant precise analysis are found. There is certainly

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

$$\texttt{[LOAD]}\ \frac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)} \qquad \texttt{[STORE]}\ \frac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)}$$

$$\texttt{[FIELD]}\ \frac{p = \&q \rightarrow f_k \quad o \in pt(q)}{o \in pt(p)}$$

$$\texttt{[CALL]}\ \frac{\_ = q(\dots, r, \dots) \quad o_{\mathit{fun}} \in pt(q) \quad \mathit{fun}(\dots, r', \dots)}{pt(r) \subseteq pt(r')}$$

$$\texttt{[RET]}\ \frac{p = q(\dots) \quad o_{\mathit{fun}} \in pt(q) \quad ret_{\mathit{fun}} p'}{pt(p') \subseteq pt(p)}$$

FIGURE 2.3: Inference rules for a basic inclusion-based points-to analysis.

a trade-off between precision and performance, and some analyses seek precise results in the parts of the program which matter for the sake of performance [Lhoták and Chung, 2011; Oh et al., 2014]. All said, some use cases do not demand sound results, such as bug detection, and in fact may shun sound results for introducing too many false positives, and so can incorporate precision more liberally by foregoing the requirement of soundness.

2.2.1 Basic Definition

We define points-to analysis as a set of inference rules mainly generating constraints between points-to sets. These constraints are continually fulfilled until a fixed-point is reached. The rules in Figure 2.3 present a basic inclusion-based points-to analysis.

In Figure 2.3, each function body instruction is present in the premise of some rule. In the [ALLOC] rule, given that a memory object is allocated and then assigned to some pointer (the ALLOC instruction), the allocated abstract memory object is inserted into the points-to set of the assigned-to pointer. For the [CAST] rule, when a pointer q is cast to a pointer p, we simply include the contents of q's points-to set into the points-to set of p ($pt(q) \subseteq pt(q)$). We

ignore the type as we are uninterested in performing a type-based analysis, thus we process the [CAST] rule as a copy. Type-based analysis is difficult for weakly typed languages such as C and C++ but is possible [Balatsouras and Smaragdakis, 2016; Barbar et al., 2020]. The $[\phi]$ rule is similar to the [CAST] rule except it includes two points-to sets, those of the operands of the ϕ function. Points-to analysis is static and has no runtime information available, so choosing an operand is not possible.

In the [LOAD] rule which handles the instruction $p = *q$, we need to include the "points-to set" $*q$ in p's points-to set. Of course, there is no points-to set for $*q$, rather, q points-to some abstract memory objects, any one of which could be an abstraction of what q points to at run time (i.e., what $*q$ is). Each one of these abstract memory objects has a points-to set. Thus, for each $o \in pt(q)$, we include the points-to set of o in p. The [STORE] rule is the reverse of the [LOAD] rule, whereby we include the points-to set of a top-level pointer in zero or more points-to sets of abstract memory objects. The Field instruction, $p = \&q \rightarrow f_k$, retrieves the k-th field of some object (that which q points to). This analysis is field-insensitive, so we treat the base object from which a field is being accessed as the field object itself. Thus, in the [FIELD] rule, if q points to o, we insert o in the points-to set of p. We will consider field-sensitivity in the next section.

The [CALL] rule handles the copying of actual arguments to formal parameters. Through the $o_{fun} \in pt(q)$ premise, it also performs on-the-fly call graph construction (and handles that all function calls are through a pointer, as previously described), connecting the Call instruction to the FunEntry instruction. Then, the $pt(r) \subseteq pt(r')$ conclusion ensures the points-to set of the actual argument is included in the points-to set of the formal parameter. The [RET] rule does the reverse of the [CALL] rule, connecting the FunExit instruction to the Call instruction(s) which called *fun*.

2.2.2 Field-Sensitivity

Field-sensitivity is a common source of extra precision, and we build our work on field-sensitive analyses. In this dissertation, we are concerned with a simple model of field-sensitivity whereby access of the k-th field of an object results in the access of a completely separate object representing that field for the accessed object (except when $k = 0$, where we simply access the object from which the field was derived from). Our approach is array-insensitive

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

FIGURE 2.4: Modifications and additions to the inference rules in Figure 2.3 to allow for field-sensitivity.

and so we treat any access to index k of an array o as an access of o, and we take a similar approach for accesses to a non-constant field such as the n-th field where n is a variable integer (you will notice we do not have instructions to model these scenarios).

Despite its simplicity (which allows for easy description and implementation), precision is improved over a completely field-insensitive analysis. However, we are then ignoring precision-improving models that consider C and C++ types (we almost blindly derive field objects), which is difficult due to the weak type system [Barbar et al., 2020; Yong et al., 1999], or LLVM types [Balatsouras and Smaragdakis, 2016], consider individual bytes [Wilson and Lam, 1995], somewhat overcoming the type system, or can consider variable indices. All of these schemes present challenges in description and implementation due to complexity and in most cases would incur a larger runtime cost than our chosen scheme. There also exist techniques to speed up the approach we have chosen and others like it [Lei and Sui, 2019] though as well carry some complexity in description and implementation. Ultimately, field-sensitivity is mostly incidental to this dissertation and the chosen scheme gives us a suitable balance of simplicity (for presentation, especially), precision, and performance. We are confident that most or all field-sensitivity schemes can work alongside that which we present in the coming sections.

To achieve field-sensitivity, we modify the `[FIELD]` rule in Figure 2.3 and add the `[FIELD-ADD]` rule as shown in Figure 2.4. In the `[FIELD]` rule, with all non-field objects $\hat{o}$ pointed to by q, a field object extended from $\hat{o}$ is included in the points-to set of p. In practice, this can mean creating new objects during the analysis. The `[FIELD-ADD]` rule handles the case when a field object is in the points-to set of q. Instead of deriving an object such as $\hat{o}.f_i.f_j$, we operate on $\hat{o}.f_{i+j}$, which is simpler to reason about.

Unfortunately for us, so-called positive weight cycles can cause the infinite derivation of field objects (an unbounded k in $\hat{o}.f_k$). This could prevent the analysis from reaching a fixed-point. To solve this, we simply cap k in

$\hat{o}.f_k$ (in our experiments, at 10 000). More sophisticated approaches to positive weight cycles have been proposed to avoid this problem and improve performance [Lei and Sui, 2019], but this is outside of this dissertation's scope.

2.3 Flow-Sensitive Inclusion-Based Points-To Analysis

As the name suggests, flow-sensitive inclusion-based points-to analysis is points-to analysis which takes control-flow into account. The rules presented in the previous section ignore control-flow, losing out on some precision. To make the analysis flow-sensitive, we must associate points-to sets for variables at program points; it is no longer enough to simply associate a single points-to set per variable. Thus, we introduce the notation $pt_{|\ell}(v)$ and $pt_{\ell|}(v)$ to represent the points-to set of v just before ℓ and just after ℓ, respectively. Importantly however, top-level variables are defined only once in partial SSA form, and so a single points-to set suffices, unlike for their address-taken counterparts. This kind of sparsity, i.e., for top-level variables only, has been referred to as semi-sparse analysis [Hardekopf and Lin, 2009] since no real sparsity is afforded to dealing with address-taken variables. In a way, the enforcement of a single definition for top-level variables encodes some form of flow-sensitivity, so it may not be entirely untrue to say that the analysis presented in the previous section does contain some flow-sensitivity since it is also based on the partial SSA form.

2.3.1 Basic Definition

The rules in Figure 2.5 present a flow-sensitive analysis. From the flow-insensitive analysis, the `[ALLOC]`, `[`ϕ`]`, `[CAST]`, `[FIELD]`, and `[FIELD-ADD]` rules remain unchanged because they operate solely on the points-to sets of top-level pointers. The `[LOAD]` and `[STORE]` rules remain unchanged except in which points-to set of o accessed.

In the `[LOAD]` rule, the points-to set of o accessed is the one immediately before the Load instruction, so what o points to just before the instruction is analysed is what is included in the points-to set of p, more closely matching real execution. In the `[STORE]` rule, we include points-to targets in the points-to set of o just after the Store instruction, analogous to what happens in real execution where o is assigned to and the new state of o appears just after the instruction.

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

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

$$\texttt{[LOAD]}\ \frac{\ell : p = *q \quad o \in pt(q)}{pt_{|\ell}(o) \subseteq pt(p)} \qquad \texttt{[STORE]}\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt_{\ell|}(o)}$$

$$\texttt{[SU/WU]}\ \frac{\ell : \_ \quad o \in \mathcal{O} \setminus \mathit{kill}(\ell)}{pt_{|\ell}(o) \subseteq pt_{\ell|}(o)} \qquad \texttt{[CONTROL-FLOW]}\ \frac{\ell \rightarrow \ell'}{\forall o \in \mathcal{O}.\ pt_{\ell|}(o) \subseteq pt_{|\ell'}(o)}$$

$$\texttt{[CALL]}\ \frac{\ell : \_ = q(\dots, r, \dots) \quad o_{\mathit{fun}} \in pt(q) \quad \ell' : \mathit{fun}(\dots, r', \dots)}{pt(r) \subseteq pt(r') \quad \ell \rightarrow \ell'}$$

$$\texttt{[RET]}\ \frac{\ell : p = q(\dots) \quad o_{\mathit{fun}} \in pt(q) \quad \ell' : ret_{\mathit{fun}} p'}{pt(p') \subseteq pt(p) \quad \ell' \rightarrow \ell}$$

$$\mathit{kill}(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{kill}(\ell : \_) \triangleq \emptyset$$

FIGURE 2.5: Inference rules for a flow-sensitive field-sensitive inclusion-based points-to analysis.

Since we have separate points-to sets for each address-taken object at different program points, we need to propagate between these individual points-to sets, as in real control-flow. Firstly, the [SU/WU] rule propagates the points-to sets of all objects across a single instruction, i.e., from before an instruction to after an instruction. For non-STORE instructions, the *kill* function always returns the empty set, and can thus be ignored. For STORE instructions though, we have the option of a strong update [Lhoták and Chung, 2011; Sui and Xue, 2016a] which prevents some inclusion and thus improves precision. Given some STORE instruction $\ell : *p = \_$, when the points-to set of p contains a single object, and that object is a singleton (would represent at most

one concrete object at runtime), then the points-to set of o before ℓ does not need to be propagated to after ℓ because we know with certainty that at runtime, the value of that object (specifically, its runtime analogue) will *always* be updated at the Store instruction (handled by the `[STORE]` rule). When the points-to set of p is larger or not a singleton, we cannot make the same assertion as some such objects at runtime may be updated, and some may not. We also propagate nothing when p's points-to set is empty to ensure a strong update can occur if the conditions are later met in the analysis. In the absence of the ability to perform a strong update, a weak update occurs, just like with the remainder of instructions.

The other form of object points-to set propagation is across instructions or along control-flow as is encoded by the `[CONTROL-FLOW]` rule. Given a control-flow edge from ℓ to ℓ' ($\ell \rightarrow \ell'$) in the *interprocedural control-flow graph* (ICFG), for each object, we propagate that object's points-to set immediately after ℓ to the object's points-to set immediately before ℓ'.

Finally, the `[CALL]` and `[RET]` instructions are as before, except with (potentially) new control-flow added to the conclusions due to on-the-fly call graph construction. In the case of indirect calls, edges in the control-flow graph may be initially missing, but now the `[CONTROL-FLOW]` rule can then propagate object points-to sets in and out of indirect calls.

2.3.2 Staged Flow-Sensitive Analysis

The analysis present in Figure 2.5 is too expensive. As programs grow, it is unreasonable to maintain and propagate a points-to set at every program point for address-taken variables. To counter this, Hardekopf and Lin [2011] introduce *staged flow-sensitive analysis* (SFS) making propagation and storage of address-taken variables far sparser.

The difficulty is that address-taken variables are not in SSA form to be able to do what has been done for top-level variables. To construct an SSA form for such variables (memory SSA form [Chow et al., 1996]), we require points-to information since address-taken variables are used and defined through (top-level) pointers. This lends to a circular dependency if we are to use the memory SSA form in the points-to analysis. Hardekopf and Lin [2011] remind that not all points-to analyses are equal in their performance and precision and so it is possible to stage our expensive precise analysis (the main analysis) with a fast imprecise analysis (the auxiliary analysis) that over-approximates

the main analysis in order to build the memory SSA form. Ultimately, it is faster and more memory efficient to perform the auxiliary analysis (we will use the flow-insensitive field-sensitive analysis from Section 2.2), construct the memory SSA form [Chow et al., 1996; Sui et al., 2018], construct subsequent data structures, and finally perform the main analysis (a flow-sensitive field-sensitive points-to analysis).

The result of constructing the memory SSA is overlaid on top of the program we are analysing rather than replacing any aspect. At any Store that defines an object o according to the auxiliary analysis, we label the instruction (immediately after) with a χ function as $o = \chi(o)$. For uses of an object o (according to the auxiliary analysis) at any Load instruction ℓ, we label ℓ with a μ function (immediately before) as $\mu(o)$. We also need to handle calls, so we annotate Call instructions with $\mu(o)$ (immediately before) or $o = \chi(o)$ (immediately after) if any of its callees, as determined by the auxiliary analysis, may use or define o. Potentially, a Call instruction could be annotated by both χ and μ. Then, associated FunEntry instructions are annotated with $o = \chi(o)$ and (in the case of o being defined per the auxiliary analysis) FunExit instructions are annotated with $\mu(o)$ [Chow et al., 1996]. Finally, the variables are converted to SSA form by renaming objects such that each is only defined once. As in the case of top-level variables, an address-taken variable may be defined in branches, thus we introduce MemPhi instructions to the program to handle merge points. MemPhi instructions work exactly as Phi instructions except that they operate on address-taken variables.

The main data structure which allows for a performant flow-sensitive main analysis is the *def-use graph* (DUG), also know as the *sparse value-flow graph* (SVFG) (SVF's nomenclature), which is the term that we will use. The SVFG is a graph comprising of program instructions as nodes (including introduced MemPhi instructions) and so called value-flows as edges. There are two kinds of value-flow edges: direct edges which represent the value-flow of top-level variables and indirect edges which represent the value-flow of address-taken variables. Direct edges are trivial to determine because the programs we analyse are in partial SSA form and top-level variables can only be defined once. A direct edge labelled with p (i.e., an edge representing the value-flow of p) is added from the SVFG node containing instruction ℓ to that containing instruction ℓ' if there exists a definition of p at ℓ and a use of p at ℓ'. We represent this as $\ell \xrightarrow{p} \ell'$. Indirect edges are more difficult to determine

Table 2.1: Variable and SVFG statistics for our benchmark programs.

Program	Variables		SVFG		
	Top-level	Address-taken	Nodes	Direct edges	Indirect edges (condensed)
dhcpcd	23 038	2398	57 540	38 157	1 854 604 (48 715)
gawk	53 342	4007	308 955	87 472	15 702 019 (466 932)
bash	45 203	4294	260 891	83 161	17 166 478 (361 537)
mutt	70 233	5880	365 239	127 466	18 341 884 (492 475)
lynx	91 191	5829	574 167	175 455	32 991 927 (801 118)
sqlite	172 000	6570	576 368	271 205	47 895 338 (717 689)
xpdf	152 653	12 124	514 780	252 951	74 489 409 (525 743)
emacs	267 269	17 505	984 030	437 978	469 240 526 (1 206 373)
git	241 840	26 481	1 562 682	460 511	476 989 809 (2 105 184)
kakoune	221 119	29 413	770 629	384 877	119 393 118 (844 865)
squid	437 645	54 678	2 334 316	771 281	579 006 358 (2 804 418)
wireshark	557 460	59 912	1 134 303	888 573	66 188 531 (716 012)

and are why we need the auxiliary analysis and memory SSA form.

We add indirect edges the same way we added direct edges except according to the computed memory SSA rather than the partial SSA which is missing any SSA form for address-taken variables. Thus, at any instruction ℓ which defines an object o_1 derived from o (i.e., a MemPhi instruction of the form $o_1 = \phi(\_, \_)$ or an instruction labelled with a χ function of the form $o_1 = \chi(\_)$), we add an indirect edge labelled with o to every instruction ℓ' which uses o_1 (i.e., MemPhi instructions of the form $\_ = \phi(o_1, \_)$ or $\_ = \phi(\_, o_1)$ or instructions labelled with $\mu(o_1)$). We represent this as $\ell \xrightarrow{o} \ell'$, notably labelling the edge with o, not variables introduced by the memory SSA form as we define the analysis in terms of the original abstract memory objects.

At the end of this process, we have an SVFG that contains a node for every instruction, including the inserted MemPhi instructions, direct value-flow edges representing the definition and use of a top-level variable according to both our auxiliary and subsequent analysis, and indirect value-flow edges representing *potential* [1] definitions and uses of address-taken variables in the main analysis. For context, Table 2.1 contains statistics on the SVFG of our benchmark (which are introduced in Section 2.4). Of note, the number of indirect edges is incredibly large. As an implementation detail, SVF merges many indirect edges into a single indirect edge labelled with a set. For example, $\ell \xrightarrow{o_1} \ell'$ and $\ell \xrightarrow{o_2} \ell'$ may be condensed into $\ell \xrightarrow{\{o_1, o_2\}} \ell'$. This reduces

[1] Recall we built the memory SSA from the results of the imprecise auxiliary analysis.

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

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

$$\texttt{[LOAD]}\ \frac{\ell : p = *q \quad o \in pt(q)}{pt_{|\ell}(o) \subseteq pt(p)} \qquad \texttt{[STORE]}\ \frac{\ell : *p = q \quad o \in pt(p)}{pt(q) \subseteq pt_{\ell|}(o)}$$

$$\texttt{[SU/WU]}\ \frac{\ell : \_ \quad o \in \mathcal{O} \setminus \mathit{kill}(\ell)}{pt_{|\ell}(o) \subseteq pt_{\ell|}(o)} \qquad \texttt{[VALUE-FLOW]}\ \frac{\ell \xrightarrow{o} \ell'}{\forall o \in \mathcal{O}.\ pt_{\ell|}(o) \subseteq pt_{|\ell'}(o)}$$

$$\texttt{[CALL]}\ \frac{\mu(a_i)\ \ell : \_ = q(\dots, r, \dots) \quad o_{\mathit{fun}} \in pt(q) \quad \ell' : \mathbf{fun}(\dots, r', \dots)\ a_1 = \chi(a_0)}{pt(r) \subseteq pt(r') \quad \ell \xrightarrow{r} \ell' \quad \ell \xrightarrow{a} \ell'}$$

$$\texttt{[RET]}\ \frac{\ell : p = q(\dots)\ a_{i+1} = \chi(a_i) \quad o_{\mathit{fun}} \in pt(q) \quad \mu(a_j)\ \ell' : \mathit{ret}_{\mathit{fun}} p'}{pt(p') \subseteq pt(p) \quad \ell' \xrightarrow{p'} \ell \quad \ell' \xrightarrow{a} \ell}$$

$$\mathit{kill}(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{kill}(\ell : \_) \triangleq \emptyset$$

FIGURE 2.6: Inference rules for the main phase of SFS.

the number of edges significantly, as shown in the final column of Table 2.1, while retaining full semantics. Still, this number is not insignificant, and storing a set obviously costs more than storing a single variable. Another way of condensing edges is presented by Hardekopf and Lin [2011] through *access equivalence* (though this is not implemented by SVF).

We can now define a sparser flow-sensitive analysis atop this data structure. The new rules for the flow-sensitive analysis in Figure 2.6 are the same as those defined earlier in Figure 2.5 except for 3 rules. The `[CONTROL-FLOW]` rule is replaced by the `[VALUE-FLOW]` rule which only differs in that we perform points-to set propagation along value-flow edges in the SVFG rather

than control-flow edges in the ICFG. This is a major contributor to sparsity. With respect to terminology, we say that a points-to set $pt_{|\ell}(o)$ resides in the IN set of ℓ and that a points-to set $pt_{\ell|}(o)$ reside in the OUT set of ℓ. There is no rule with the MemPhi instruction explicitly in the premise as the MemPhi instruction's purpose is achieved by the `[VALUE-FLOW]` rule.

The `[CALL]` and `[RET]` rules are modified to add SVFG edges instead of ICFG edges. In the rules, direct SVFG edges do not appear in any premise so could be omitted but we include them for completeness. They are also useful in implementation. In grey, we have μ and χ annotations that *may* have been added earlier. When they do exist, we add indirect value-flow edges in and out of functions, from and to the appropriate callsite.

2.4 Benchmarks

Throughout this dissertation, we evaluate our various techniques by implementing them in open source points-to analysis framework SVF [Sui and Xue, 2016b] which analyses LLVM bitcode. SVF constructs its data structures such as the *pointer assignment graph* (PAG) and control-flow graph from the program's LLVM representation or analysis of it. It should be noted though that C++ programs encode more complicated semantics than C programs and thus produce more complicated bitcode. To better handle this, SVF attempts to rebuild the class hierarchy and treats virtual table pointers in a special manner to constrain the number of call targets which would otherwise have been determined for virtual calls. Still, C++ is more complicated than C in ways more than virtual calls, and work is underway for further improvements (e.g., a simplified *Standard Template Library* (STL) to produce less complicated bitcode for analysis's sake [Simplified-STL, 2022]).

To evaluate against, we use 12 open source programs built with the `O3` optimisation level through `crux-bitcode` [crux-bitcode, 2021] which uses *Whole Program LLVM in Go* (GLLVM) [WLLVM, 2021] and LLVM/Clang 12.0.0. These programs were chosen to cover a variety of domains and since they are programs that would benefit from points-to analysis. Certainly all of these programs would benefit from aggressive optimisation as they either regularly operate on large amounts of data, run constantly in the background, or face the user in such a way that any stutter would be noticeable. All of them also form important attack vectors (a concrete example would be bash through

Table 2.2: Information about our benchmark programs.

Program	Version	Size (in MB)	Instructions	Lines of code	Description
dhcpcd	9.3.4	1.20	55 546	62 387	Dynamic Host Configuration Protocol client
gawk	5.1.0	2.53	124 307	52 328	GNU AWK; text filtering language interpreter
bash	5.0.18	2.73	127 554	74 446	Bourne Again Shell; Unix shell interpreter
mutt	2.0.3	3.33	153 353	80 214	Text-based email client
lynx	2.8.9	5.36	178 002	134 322	Text-based web browser
sqlite	3.34.0	8.79	444 146	183 147	SQL database engine
xpdf	4.03	8.99	339 979	119 298	PDF viewer
emacs	27.2	11.89	568 065	236 465	Extensible text editor
git	2.29.2	12.55	499 529	205 102	Distributed version control system
kakoune	2020.08.04	17.73	478 033	30 496	Modal text editor
squid	4.13	22.42	772 825	274 456	Web proxy cache
wireshark	3.4.0	35.43	1 232 779	344 720	Network packet analyser

Shellshock [CVE, 2014]) or regularly deal with untrusted data so would benefit from accurate bug detection.

In Table 2.2, we describe our benchmarks. The size refers to the size of each of the bitcode files (without debug information), the number of instructions is obtained using LLVM `opt`'s `instcount` option, and the lines of code are counted by counting the number of lines of codes in the C and C++ header and source files which appear in the debug information of the bitcode files, excluding system headers (this is obtained through `crux-bitcode`). For the compiled and analysed parts of the programs, xpdf, kakoune, and squid are primarily written in C++ whilst the remaining benchmarks are primarily written in C.

One peculiarity is the low number of lines of code for kakoune but high number of instructions and large size. kakoune is written in C++ and template expansion would not factor into lines of code. It also makes use of classes in the C++ template library whose implementation would be included in the compiled result (not dynamically linked) but be excluded by `crux-bitcode` when counting lines of code since they are considered system headers.

All of our experiments are conducted on a 64-bit Debian 11 machine with an AMD Ryzen 5800X processor. SVF is compiled with Clang 11 using the `O3` optimisation level and the `march=native` option. Statistics are gathered by running analyses 3 times, which we have found to sufficiently handle variance, and taking an arithmetic mean for results (such as time taken and memory used). Any memory statistics refer to the maximum resident set size of the entirety of SVF's execution and is measured with GNU's `time`. Time statistics are measured within SVF and are wall clock times obtained through POSIX's `clock_gettime` function with the first argument set to `CLOCK_MONOTONIC`.

Memory is capped at 120 GB and time is capped at 12 hours for each individual analysis execution. No analysis ever reached the time limit but some ran *out of memory* (OOM).

Benchmarks, code, scripts, and instructions on reproducing our experiments are available at `https://mbarbar.net/papers/ds-pta-artifact.zip`.

Object Versioning 3

Though SFS significantly reduces the number of object points-to set propagations required, there does remain redundancy. There are still many instances where the points-to set of a single object remains unchanged across large parts of the SVFG. Thus, this chapter focuses on the points-to sets of individual objects at multiple program points and reducing storage and propagation within that.

The points-to set of an object o at an instruction can be reused at another instruction if it is unchanged. This would save time propagating points-to sets and space storing them. In essence, we need to determine where the points-to sets of an object o will be the same in the main analysis. The result is that we can *version* objects, and we say that two program points may refer to the same version of o if the points-to set of o at those two program points are guaranteed to be the same. Then, instead of accessing points-to sets of objects stored per program point, we would access points-to sets of versions of objects, with the aim that we would have far fewer versions per object than the number of program points it is relevant at. Concretely, we aim to access the various points-to sets of an object o as some $pt_{\kappa}(o)$, where $\kappa \in \mathcal{K}$ is a so-called version, instead of accessing them as some $pt_{|\ell}(o)$ or $pt_{\ell|}(o)$.

To achieve this, we use *meld labelling*, a prelabelling extension for directed graphs. Meld labelling extends a prelabelling (a straightforward node labelling applied to the graph which we will use to perform the main meld labelling procedure) such that the label of each node which has not been prela-

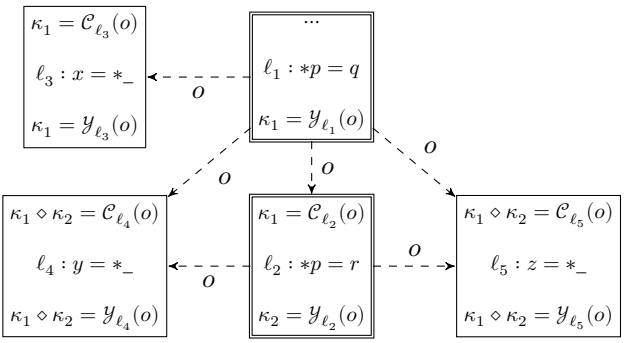
belled is a "melding" of the labels of its incoming neighbour nodes. We perform meld labelling on a per-object basis, so each node will have multiple labels. With a careful prelabelling, we can soundly determine when an object will have the same points-to set at two program points, allowing us to skip work during the main phase of the analysis. This occurs when two program points read or write the same label (i.e., the same version). The intuition is that this pre-analysis (object versioning) would save enough work from the main flow-sensitive analysis that it would be cheaper overall in both time and space. The pre-analysis also pulls some computation out of points-to analysis into, what is in essence, solving a transitive closure, allowing for optimisations independent of points-to analysis. We also find that this problem is extremely amenable to parallelisation.

In summary, this chapter describes meld labelling on the SVFG in order to version objects and how to do so inexpensively, presents rules and algorithms describing how to perform flow-sensitive analysis using versions, *versioned staged flow-sensitive analysis* (VSFS), and evaluates the effectiveness of this approach along the axes of time and memory. Most of the work in this chapter has been published at CGO [Barbar et al., 2021].

3.1 Motivating Example

Figure 3.1 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], though we choose the points-to sets and edge labels for simplicity) in Figure 3.1a, with some extraneous edges and nodes removed, and the required points-to sets and propagation constraints for flow-sensitive analysis in Figure 3.1b (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 only a single 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 various ℓ represent instruction labels for ease of reference. For exemplary purposes, we assume $pt(p) = \{o\}$, $pt(q) = \{a\}$, and $pt(r) = \{b\}$ and that all points-to sets of o are empty according to the

[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`.



(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 propagation constraints required.

FIGURE 3.1: An example motivating the effectiveness of object versioning.

current state of the flow-sensitive analysis, so the two STORE nodes may define o, which p points to. According to Figure 3.1b, 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 later in this section.

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. Column 2 of Figure 3.1b 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 ($pt_{|\ell_2}(o) = pt_{|\ell_3}(o) = pt_{\ell_1|}(o)$) since o's points-to set in their IN sets are formed by propagation of that in ℓ_1's OUT set only. Similarly, the points-to set of o in the IN sets of ℓ_4 and ℓ_5 are equivalent ($pt_{|\ell_4}(o) = pt_{|\ell_5}(o)$) since they are both made up of the union of the points-to sets of o in the OUT sets of ℓ_1 and ℓ_2. It stands to reason that we can detect much of this equivalence in a pre-analysis on the SVFG allowing points-to sets to be reused instead of repeatedly storing equivalent points-to sets and repeatedly propagating equivalent, or otherwise, points-to sets to form other sets of equivalent points-to sets.

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 which operate on the points-to set of o. 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 access $pt_{\mathcal{C}_\ell(o)}(o)$. The flow-sensitive analysis propagates points-to sets from $pt_{\mathcal{Y}_\ell(o)}(o)$ to $pt_{\mathcal{C}_{\ell'}(o)}(o)$ if an edge $\ell \xrightarrow{o} \ell'$ exists rather than propagating from $pt_{\ell|}(o)$ to $pt_{|\ell'}(o)$. Versions are simply labels or identifiers which only matter in how they relate to each other, not in their concrete values. In a simple pre-analysis, for each object, nodes are assigned a consumed version and a yielded version which can be used to access points-to sets of objects instead of accessing them from 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. The

pre-analysis maintains the properties that

$$\mathcal{C}_\ell(o) = \mathcal{C}_{\ell'}(o) \Rightarrow pt_{|\ell}(o) = pt_{|\ell'}(o), \tag{3.1}$$

$$\mathcal{C}_\ell(o) = \mathcal{Y}_{\ell'}(o) \Rightarrow pt_{|\ell}(o) = pt_{\ell'|}(o), \text{and} \tag{3.2}$$

$$\mathcal{Y}_\ell(o) = \mathcal{Y}_{\ell'}(o) \Rightarrow pt_{\ell|}(o) = pt_{\ell'|}(o), \tag{3.3}$$

which ensures that our versioning, when used to perform the main phase, produces precisely the same results as SFS.

Applying our Approach The consumed and yielded versions of the nodes in our example can be seen in Figure 3.1a. The points-to sets of o in the IN sets of ℓ_2 and ℓ_3 are generated through the propagation of the points-to set of o in the OUT set of ℓ_1 and are thus equivalent, so $\mathcal{C}_{\ell_2}(o) = \mathcal{C}_{\ell_3}(o) = \mathcal{Y}_{\ell_1}(o) = \kappa_1$. Similarly, the points-to sets of o in the IN sets of ℓ_4 and ℓ_5 are generated through the union of the points-to sets of o in the OUT sets of ℓ_1 and ℓ_2 ($\kappa_1 \diamond \kappa_2$ is another version separate to κ_1 and κ_2 and the $\diamond$ operator is described in Section 3.2). All this can be determined before any flow-sensitive analysis. Since multiple nodes may share versions, we can store fewer points-to sets. We also generate fewer propagation constraints since each of our new constraints represent more than one of those generated to perform SFS, and some are eliminated. Improvements to the number of points-to sets required and the number of propagation constraints generated are shown in Figure 3.1b 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 performing fewer points-to set propagations[2].

3.2 Meld Labelling

Meld labelling is a prelabelling extension on directed graphs where each non-prelabelled 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

[2]We note that points-to graph equivalence [Hardekopf and Lin, 2009] improves the situation too, and the example was chosen for simplicity. Points-to graph equivalence is discussed further in Section 3.7

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,

$$\begin{aligned} \kappa_1 \diamond \kappa_2 &= \kappa_2 \diamond \kappa_1 && \text{(Commutativity)} \\ \kappa_1 \diamond (\kappa_2 \diamond \kappa_3) &= (\kappa_1 \diamond \kappa_2) \diamond \kappa_3 && \text{(Associativity)} \\ \kappa_1 \diamond \kappa_1 &= \kappa_1 && \text{(Idempotence)} \\ \kappa_1 \diamond \varepsilon &= \kappa_1 && \text{(Identity)} \end{aligned}$$

The set union operator, $\cup$, and the bitwise-or operator found in many programming languages, are examples of suitable meld operators when $\mathcal{K}$ is the set of sets or bit-vectors.

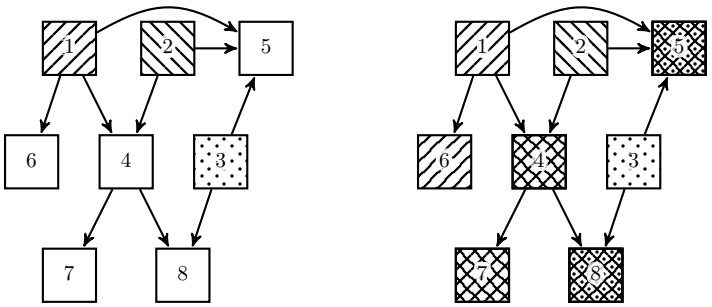
The graph is initially prelabelled with labels in $\mathcal{K}$, except ε, per some condition chosen according to the meld labelling's purpose. Nodes that are not part of that prelabelled subset, *Pre*, are labelled with the identity ε. The meld labelling process is simple: meld the label of each node not in *Pre* 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 `[MELD]` rule in Figure 3.2 where n and n' are nodes, and κ_n and $\kappa_{n'}$ are the labels of n and n'.

$$\texttt{[MELD]} \quad \frac{n' \to n \quad n \notin \mathit{Pre}}{\kappa_n = \kappa_{n'} \diamond \kappa_n}$$

Figure 3.2: 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 that finish with ε are in their own class: nodes unreachable by any prelabelled node. In essence, this problem is a subset of a transitive closure.

Figure 3.3 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 the remaining nodes are labelled with the identity □. The meld operator $\diamond$ melds or combines the patterns.



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

FIGURE 3.3: An example of meld labelling. Patterns are labels and the meld operator $\diamond$ combines them. The blank pattern is the identity.

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 $\square$ is the identity, all cases can be derived:

$\blacksquare \diamond \blacksquare = \blacksquare \qquad \blacksquare \diamond \square = \blacksquare \qquad \blacksquare \diamond \square = \blacksquare \qquad \blacksquare \diamond \square = \blacksquare.$

For example, we can determine that $\blacksquare \diamond \blacksquare = \blacksquare$ because $\blacksquare \diamond \blacksquare = (\blacksquare \diamond \square) \diamond \blacksquare = \blacksquare \diamond (\square \diamond \blacksquare) = \blacksquare \diamond (\blacksquare \diamond \square) = (\blacksquare \diamond \blacksquare) \diamond \square = \blacksquare \diamond \square = \blacksquare$.

Importantly, in Figure 3.3, 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 of labels at nodes is not a result of sharing incoming neighbours but by sharing the set of labels (from prelabelling) which reach them.

Complexity In the worst case, meld labelling takes $\mathcal{O}(|E||Pre|)$ time, where E is the set of edges, if we count the number of times a meld needs to occur (i.e., excluding the complexity of the meld operation). 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. This only differs slightly from a standard transitive closure worst case of $\mathcal{O}(|E||N|)$ where N is the set of nodes in the graph. 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. The choice of label is important, as a label taking $\mathcal{O}(|Pre|)$ space (as opposed to $\mathcal{O}(1)$) in the worst case may become too expensive and unwieldy when we apply this on a per object basis as we do in the next section.

3.3 Versioning Objects

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 (if we can determine this in a pre-analysis, at least), 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 may define. An instruction which may not define a particular object would yield the version of that object which it consumes.

A version of an object represents a state of that object's points-to set such that any change in an object's points-to set requires a new version. Since we perform versioning as a pre-analysis, any *potential* change in an object's points-to set warrants a version. The points-to set of an object o at a program point may change in two ways: (1) through a Store instruction $*p = q$ when p points to o, and (2) through the merging of points-to sets of o at different program points before an instruction (recall the `[VALUE-FLOW]` rule), that is, when o's points-to set in an IN set is the union of that in multiple OUT sets.

All instructions consume a single version per object and yield a single version per object. Determining the versions of an object and which versions of it each instruction may consume and 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 points-to information produced by the auxiliary analysis. This may give us more versions than necessary whereby two versions may be collapsible into a single version if versioning was done using more precise points-to information, but the over-approximation is sound and is still performant as we shall see in Section 3.6.

Where $\mathcal{K}$ is the set of all versions, we give each instruction ℓ a $\mathcal{C}$ (for consume) function, defined as,

Definition 3.3.1 *$\mathcal{C} : \mathcal{A} \mapsto \mathcal{K}$ where $\mathcal{C}_\ell(o)$ is the version of o which ℓ consumes,*

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

Definition 3.3.2 *$\mathcal{Y} : \mathcal{A} \mapsto \mathcal{K}$ where $\mathcal{Y}_\ell(o)$ is the version of o which ℓ yields.*

These versions are just labels with no meaning except to differentiate between them. If we were to use natural numbers as versions, for example, no impor-

tance is given to magnitude; there is no difference between versions say κ_1 and κ_2 and we are only concerned with whether they are equal or not. 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 value-flow merges.

Meld labelling 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 may modify objects' points-to sets, 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 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. In the context of meld labelling for *versioned staged flow-sensitive analysis* (VSFS), our new analysis, the terms label and version are synonymous, that is, we will use the result of an appropriately tweaked meld labelling to version objects.

3.3.1 Preversioning

At any given node, multiple objects may be used or defined. Thus, we are not aiming for a single version (label) at each node, but would need a version per object (of interest). In fact, 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 this section, we will focus on prelabelling, or *preversioning*, the SVFG, and modify the meld labelling process in the next section to account for this kind of prelabelling.

In the course of execution, assuming a STORE instruction operates on object 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 (i.e., define o). Whether a STORE instruction $*p = q$ modifies an object's points-to set relies on whether 1) p points to o, and 2) whether the points-to set of q contains elements not found in the points-to set of o. We have an imprecise form of the first piece of information through the sound auxiliary analysis. From the aux-

iliary analysis, we can soundly infer whether p *might* point to o in the final flow-sensitive analysis but it may not be possible to determine that the Store instruction may modify the points-to sets of o or not by only considering the results of the auxiliary analysis. Thus, when p points to o in the auxiliary analysis, we soundly assume Store instructions would always modify the points-to set of o during the main analysis, and thus yield a different version of o to that which they consume (regardless of whether this actually occurs). 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 sets, respectively, at each node. Since the version of an object which a Store instruction yields is a new version, and not reliant on any other version, preversioning should occur at Store nodes. Specifically, for each object o which may be defined at each Store instruction ℓ, we need to provide ℓ's yielded version as a preversion (i.e., set $Y_\ell(o)$).

Example 3.3.1 *The state of the SVFG in the motivating example (Figure 3.1) after preversioning is shown in Figure 3.4. The Store nodes are given preversions (κ_1 and κ_2) to yield for o and all other consumed and yielded versions are set to the identity.*

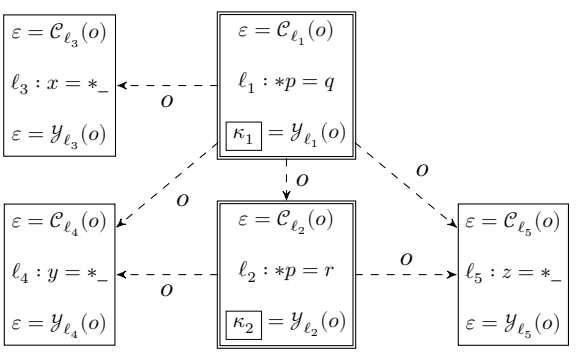


Figure 3.4: The SVFG from the motivating example after the preversioning phase. Versions introduced in this phase are boxed.

If we use the results from the imprecise auxiliary analysis to perform call graph resolution [Hardekopf and Lin, 2011], this is sufficient. However, we would prefer performing on-the-fly call graph resolution using results from the flow-sensitive points-to analysis itself (the main phase) as we would expect a smaller call graph. Thus, the SVFG may be missing edges which may

be added during the main phase and this can affect versioning. Specifically, 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, which we refer to as a δ-node, consumes a unique version for each object it may propagate forward. These unique versions are assigned during preversioning. 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). Obviously, direct calls are static. 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 improves performance but does not appear to be possible with the available information.

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

Definition 3.3.3 $\delta : \mathcal{L} \mapsto \mathbb{B}$ *such that*

$$\delta(\ell) = \mathit{true} \Leftrightarrow \exists \ell' \in \mathcal{L}.\ \exists o \in \mathcal{O}.\ \ell' \overset{o}{\twoheadrightarrow} \ell \wedge P(\ell' \overset{o}{\rightarrow} \ell)$$

where $P(\ell' \overset{o}{\rightarrow} \ell)$ indicates the *possibility* of indirect value-flow edge $\ell' \overset{o}{\rightarrow} \ell$ being created during the main analysis due to on-the-fly call graph resolution.

Preversioning 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 3.5 show this performed on an SVFG with all $\mathcal{C}$ and $\mathcal{Y}$ having been already set as the identity ε (for all objects o, where a node is not given a fresh version to consume/yield, we want it to consume/yield ε). The `[STORE]` rule ensures that Store instructions yield a new version for each object they may define, as determined by the auxiliary analysis, and the `[OTF-CG]` rule ensures that δ-nodes similarly consume a new version for each object which they may eventually yield in case that is necessary.

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

`[OTF-CG]`
$$\frac{\delta(\ell) \quad \ell \overset{o}{\rightarrow} \ell' \quad \varepsilon = \mathcal{C}_\ell(o)}{\mathcal{C}_\ell(o) = nv(o)}$$

Figure 3.5: Preversioning 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.

3.3.2 Meld Versioning

At this point, we have an SVFG with the versions consumed and yielded set at a small portion of nodes (preversions). In meld versioning of the preversioned SVFG, we need to consider that edges are labelled with address-taken objects. We only 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 not directly affected by the value of o at ℓ (indirect edges could be added during on-the-fly call graph resolution, but we handled this soundly during preversioning).

Example 3.3.2 *In Figure 3.6, $\kappa_1 = \mathcal{Y}_{\ell_1}(a)$ is only propagated along the edge to ℓ_2 labelled with a. This occurs similarly for $\kappa_2 = \mathcal{Y}_{\ell_1}(b)$ with ℓ_3 and b.*

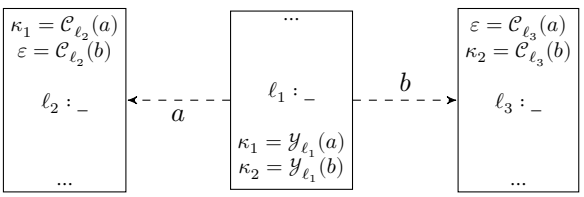


Figure 3.6: An example SVFG involving two objects, a and b.

That each node may have two versions per object (a consumed and a yielded version) also needs to be considered. We thus perform the meld versioning 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 $o \in \mathcal{A}$. An implementation would not need to store $\mathcal{Y}_\ell(o)$ and $\mathcal{C}_\ell(o)$ separately, but we do so here for simplicity. 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 potentially other versions too) except when $\delta(\ell')$ because $\mathcal{C}_{\ell'}(o)$ would be a preversion. We explicitly avoid changing what was set in the preversioning phase as unique versions were specifically chosen for those positions, and we want to maintain that. More formally, the inference rules in Figure 3.7 will, for each node, determine the consumed and yielded version of objects used at that node.

The `[EXTERNAL]` rule propagates a yielded version from the incoming neighbours of a non-δ-node and melds that with the consumed version of that node. This rule is similar to the `[MELD]` rule in the original definition of meld

$$\texttt{[EXTERNAL]}\quad \frac{\ell \xrightarrow{o} \ell' \quad \neg\delta(\ell')}{\mathcal{C}_{\ell'}(o) = \mathcal{C}_{\ell'}(o) \diamond \mathcal{Y}_{\ell}(o)} \qquad \texttt{[INTERNAL]}\quad \frac{\neg \ell : *\_ = \_}{\mathcal{Y}_{\ell}(o) = \mathcal{C}_{\ell}(o)}$$

FIGURE 3.7: Inference rules for meld versioning.

labelling in Section 3.2. It excludes δ-nodes because they have had their relevant consumed versions set in the preversioning phase. The [INTERNAL] rule ensures that any node which yields what it consumes—non-STORE nodes—has its yielded version set to its consumed version.

Example 3.3.3 *In Figure 3.8, we revisit our motivating example (Figure 3.1) again after preversioning in Example 3.3.1. o's version is propagated externally from ℓ_1 to ℓ_2, ℓ_3, ℓ_4, and ℓ_5 and from ℓ_2 to ℓ_4 and ℓ_5. When more than one version is propagated to another node, more interesting melding occurs, as is seen in the consumed version of o at nodes ℓ_4 and ℓ_5 ($\kappa_1 \diamond \kappa_2$). Internal propagation occurs at ℓ_3, ℓ_4, and ℓ_5 since they yield what they consume. For example, $\mathcal{Y}_{\ell_3}(o) = \mathcal{C}_{\ell_3}(o)$. Thus, from the process and result, it becomes clear why some consumed or yielded versions are equivalent.*

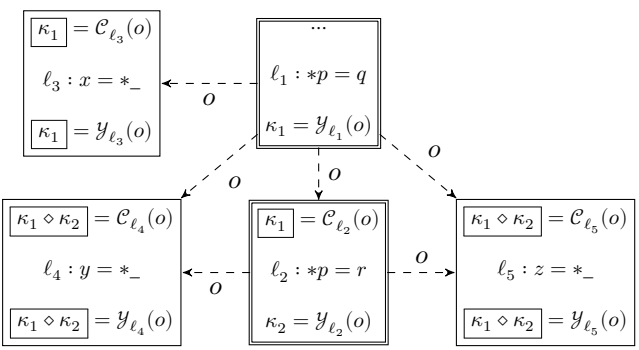


FIGURE 3.8: The SVFG from the motivating example after being versioned. Consumed/yielded versions changed during meld versioning are boxed.

3.4 Flow-Sensitive Points-To Analysis with Versioned Objects

At this point, every instruction which may access the points-to set of an address-taken object is given two versions for that object: the version of that

object it consumes and the version of that object 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. With many program points sharing versions, we can save on redundant points-to sets. The inference rules in Figure 3.9 modify SFS as described in Figure 2.5 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 analysis* (VSFS).

The [LOAD] and [STORE] 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 or yielded versions of o's points-to sets at the instruction, respectively. For Load instruction $p = *q$, the [LOAD] rule includes q's pointees' points-to sets (consumed versions) in p's points-to set, and for Store instruction $*p = q$, the [STORE] rule includes q's points-to set in p's pointees' points-to sets (yielded versions). The [SU/WU] 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 SFS but uses versions to choose object points-to sets. Finally the [VALUE-FLOW] rule propagates points-to sets between nodes. Now, given $\ell \xrightarrow{o} \ell'$, instead of propagating from the OUT set of ℓ (or the IN set as a simple optimisation, 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, many of the constraints generated are equivalent meaning propagation occurs far less often.

The remainder of the analysis works in the exact same way as SFS. The [ALLOC] rule inserts a newly allocated object in the left-hand side pointer's points-to set. The [ϕ] and [CAST] 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*] 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. The [CALL] and [RET] rules copy the value of actual arguments to formal arguments and return values to pointers, respectively. Again, using a pointer q as the called function allows for on-the-fly call graph resolution. As shown in grey, if the instructions are annotated with χ/μ, they produce new edges

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

$$\texttt{[LOAD]}\ \frac{\ell : p = *q \quad o \in pt(q) \quad \kappa_c = \mathcal{C}_\ell(o)}{pt_{\kappa_c}(o) \subseteq pt(p)}$$

$$\texttt{[STORE]}\ \frac{\ell : *p = q \quad o \in pt(p) \quad \kappa_y = \mathcal{Y}_\ell(o)}{pt(q) \subseteq pt_{\kappa_y}(o)}$$

$$\texttt{[SU/WU]}\ \frac{\ell : *p = \_ \quad o \in \mathcal{A} \setminus \mathit{kill}(\ell) \quad \kappa_c = \mathcal{C}_\ell(o) \quad \kappa_y = \mathcal{Y}_\ell(o)}{pt_{\kappa_c}(o) \subseteq pt_{\kappa_y}(o)}$$

$$\texttt{[VALUE-FLOW]}\ \frac{\ell \xrightarrow{o} \ell' \quad \kappa_s = \mathcal{Y}_\ell(o) \quad \kappa_d = \mathcal{C}_{\ell'}(o)}{pt_{\kappa_s}(o) \subseteq pt_{\kappa_d}(o)}$$

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

$$\texttt{[CALL]}\ \frac{\mu(a_i)\ \ell : \_ = q(\dots, r, \dots) \quad o_{\mathit{fun}} \in pt(q) \quad \ell' : \mathit{fun}(\dots, r', \dots)\ a_1 = \chi(a_0)}{pt(r) \subseteq pt(r') \quad \ell \xrightarrow{r} \ell' \quad \ell \xrightarrow{a} \ell'}$$

$$\texttt{[RET]}\ \frac{\ell : p = q(\dots)\ a_{i+1} = \chi(a_i) \quad o_{\mathit{fun}} \in pt(q) \quad \mu(a_j)\ \ell' : ret_{\mathit{fun}} p'}{pt(p') \subseteq pt(p) \quad \ell' \xrightarrow{p'} \ell \quad \ell' \xrightarrow{a} \ell}$$

$$\mathit{kill}(\ell : *p = \_) \triangleq \begin{cases} \{o\} & \text{if } pt(p) \equiv \{o\} \wedge o \text{ is singleton} \\ \mathcal{O} & \text{if } pt(p) \equiv \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathit{kill}(\ell : \_) \triangleq \emptyset$$

FIGURE 3.9: Inference rules for the new main phase flow-sensitive analysis in VSFS.

if they did not already exist. Recall that δ-nodes had their consumed version for relevant objects already set. This is what allows this call graph resolution to work seamlessly as we do not need to modify any versions when handling the [CALL] or [RET] rules for the analysis to move forward.

3.5 Efficient Versioning

To naively implement versioning as in the rules in Figure 3.7 would be expensive, especially in space. While it is possible to use integers as versions, the meld versioning phase requires versions to be represented as sets (or similar; e.g., bit-vectors) to be performed quickly and naturally. Associating two bit-vectors per relevant object per node, each containing up to $|Pre|$ elements, does not scale very well, particularly with respect to memory. Thus, we describe a few methods which help keep memory footprint low whilst also improving versioning time, ensuring VSFS is competitive.

3.5.1 Per-Object Versioning

Indirect edges in the SVFG are labelled with objects. One option for an implementation is to perform versioning for all objects at once, i.e., processing all the outgoing edges of an SVFG node regardless of the object they are labelled with. Performing this on a per-object basis however opens up new optimisations. With respect to the flow of points-to information of address-taken objects along the SVFG (recall the [VALUE-FLOW] and [SU/WU] rules), the SVFG can be thought of as multiple graphs, each of which is made up of the same nodes but with edges only labelled with a particular object. Then, it is easy to think about performing versioning on a per-object basis by considering these subgraphs, each of which we will refer to as an o-SVFG where all edges not labelled with o are removed.

An o-SVFG is not necessarily acyclic, but performing versioning on an acyclic graph is far more efficient than doing so on a cyclic graph as we can exploit the topological order of the graph for propagation (recall that versioning is similar to solving the transitive closure or determining reachability). By splitting up an o-SVFG into strongly connected components (with some caveats), we obtain an acyclic graph of these qualified strongly connected components. We want to split each o-SVFG such that every node in a given such strongly connected component has equivalent consumed versions of o

and equivalent yielded versions of o. For this, we modify Tarjan's strongly connected component algorithm [Tarjan, 1972]. We chose the original formulation for our implementation but later iterations of the algorithm can also be used [Nuutila and Soisalon-Soininen, 1994; Pearce, 2016].

In considering the neighbours of v, w, in the STRONGCONNECT procedure in the original algorithm [Tarjan, 1972, §4], we ignore any w which is a δ-node or STORE node. We "cut off" δ-nodes because they consume a preversion and thus cannot share their consumed version with any incoming neighbours. We cut off STORE nodes because they yield a preversion and thus we cannot guarantee that all of our qualified strongly connected components involving such a node will all yield a single version (consider that all other nodes yield what they consume and thus might yield a melding involving the version yielded in a STORE and a version yielded outside our qualified strongly connected component).

We can perform the meld versioning on our qualified strongly connected components rather than individual nodes, and all nodes within these components will consume the same version and yield the same version. With a function $scc : \mathcal{L} \mapsto \mathcal{L}$ which returns some canonical node within the qualified strongly connected component the argument belongs to, we can modify our meld versioning rules as in Figure 3.10. Then, many constraints would be equivalent and we do not store versions for every node (at this stage), rather, just for some canonical node for each of our qualified strongly connected components. The `[ASSIGN]` rule ensures that all nodes in a qualified strongly connected component are given equivalent versions to the canonical node of their component. In an implementation, the `[ASSIGN]` rule can be implemented as a concluding linear pass converting expensive set- or bit-vector-based versions into more inexpensive forms such as integers

$$\texttt{[EXTERNAL]}\quad \frac{\ell \xrightarrow{o} \ell' \quad \neg\delta(\ell')}{\mathcal{C}_{scc(\ell')}(o) = \mathcal{C}_{scc(\ell')}(o) \diamond \mathcal{Y}_{scc(\ell)}(o)} \qquad \texttt{[INTERNAL]}\quad \frac{\neg\ell : *\_ = \_}{\mathcal{Y}_{scc(\ell)}(o) = \mathcal{C}_{scc(\ell)}(o)}$$

$$\texttt{[ASSIGN]}\quad \frac{\ell : \_}{\mathcal{C}_{\ell}(o) = \mathcal{C}_{scc(\ell)}(o) \quad \mathcal{Y}_{\ell}(o) = \mathcal{Y}_{scc(\ell)}(o)}$$

FIGURE 3.10: Versioning inference rules using our (qualified) strongly connected components.

Furthermore, the inference rules in Figure 3.10 can be implemented very efficiently because Tarjan's strongly connected component algorithm produces a (reverse) topological sorting of the graph. With the o-SVFG topologically sorted, we can implement the `[EXTERNAL]` and `[INTERNAL]` rules in linear time, and the `[ASSIGN]` rule can be implemented in another final linear pass, as mentioned.

3.5.2 o-SVFG Isomorphism

If two o-SVFGs are isomorphic, ignoring the o labels, then the versions assigned at each node corresponding to their object will be equivalent (in shape, not necessarily in value). Thus, if we perform versioning on say the o_1-SVFG, then later find that the o_2-SVFG is isomorphic to the o_1-SVFG (modulo edge labels), we can simply set $\mathcal{C}_\ell(o_2) = \mathcal{C}_\ell(o_1)$ and $\mathcal{Y}_\ell(o_2) = \mathcal{Y}_\ell(o_1)$ for all $\ell \in \mathcal{L}$. Versions of different objects o_1 and o_2 are used independently and never mixed, so new names are not required. Graph isomorphism in this case is trivial to check because both graphs are made up of the same nodes.

3.5.3 Foregoing Indirect Value-Flow Edges

We say that a version κ' of object o *relies* on version κ (also that of object o) if $\ell \xrightarrow{o} \ell' \wedge \kappa = \mathcal{Y}_\ell(o) \wedge \kappa' = \mathcal{C}_{\ell'}(o)$. This essentially produces a second conclusion to the premises in the `[VALUE-FLOW]` rule in Figure 3.9. Since versioning is a pre-analysis producing static versions, this can be determined before the main flow-sensitive analysis is performed and will allow us to forego parts of the SVFG and the storage of many versions, reducing memory pressure.

We introduce a relation $\kappa \xrightarrow{o} \kappa'$ to indicate the reliance of version κ' on version κ for object o. To determine all such reliance relations, we can simply apply the rule in Figure 3.11. With these reliances encoded before the main analysis, determining reliances before the main analysis allows us to forego (i.e., delete or free) all indirect value-flow edges in the SVFG, but we need to modify some rules to account for this.

$$\texttt{[VERSION-RELIANCE]}\quad \frac{\ell \xrightarrow{o} \ell' \quad \kappa = \mathcal{Y}_\ell(o) \quad \kappa' = \mathcal{C}_{\ell'}(o)}{\kappa \xrightarrow{o} \kappa'}$$

Figure 3.11: Inference rule to determine version reliance.

The modifications to the rules are presented in Figure 3.12. The [CALL] and [RET] rules now introduce a new version reliance as opposed to a new indirect value-flow. Importantly, the versions themselves are static as we chose the preversioning of δ-nodes to account for this, but reliances are not. The [VERSION-FLOW] rule replaces the [VALUE-FLOW] rule, doing the same thing as before, except using version reliances instead of indirect value-flows as a premise. It can almost be seen as a substitution of the old [VALUE-FLOW] rule since the [VERSION-RELIANCE] rule which produced the new premise as its conclusion ($\kappa \xrightarrow{o} \kappa'$) uses the old premise ($\ell \xrightarrow{o} \ell' \dots$) as its premise.

[CALL]
$$\frac{\mu(a_i)\ \ell : \_ = q(\dots, r, \dots) \quad o_{\mathit{fun}} \in pt(q) \quad \ell' : \mathit{fun}(\dots, r', \dots)\ a_1 = \chi(a_0) \quad \kappa = \mathcal{Y}_\ell(a) \quad \kappa' = \mathcal{C}_{\ell'}(a)}{pt(r) \subseteq pt(r') \quad \kappa \xrightarrow{a} \kappa'}$$

[RET]
$$\frac{\ell : p = q(\dots)\ a_{i+1} = \chi(a_i) \quad o_{\mathit{fun}} \in pt(q) \quad \mu(a_j)\ \ell' : ret_{\mathit{fun}} p' \quad \kappa' = \mathcal{Y}_{\ell'}(a) \quad \kappa = \mathcal{C}_\ell(a)}{pt(p') \subseteq pt(p) \quad \kappa' \xrightarrow{a} \kappa}$$

[VERSION-FLOW]
$$\frac{\kappa \xrightarrow{o} \kappa'}{pt_\kappa(o) \subseteq pt_{\kappa'}(o)}$$

FIGURE 3.12: Inference rule modifications from the main analysis rules in Figure 3.9 to account for the introduction of version reliances and the removal of indirect value-flow edges in the SVFG.

3.5.4 Storing Fewer Versions

Versions of objects at instructions are accessed in four rules: the [LOAD] and [STORE] rules, and the new [CALL] and [RET] rules. Previously, the [VALUE-FLOW] rule accessed versions at many instructions, but with our version reliances and the [VERSION-FLOW] rule, we no longer need to. Thus, it makes sense to only store versions at nodes in which they are now required, saving space.

We store versions at four instructions or nodes and forego the rest after determining version reliances: LOAD nodes, STORE nodes, δ-nodes, and δ_{src}-nodes. δ-nodes are the destinations of indirect calls and returns, and we now define δ_{src}-nodes to be those which *could* be, but are not (yet, perhaps), source nodes to new edges added to the SVFG in the original conception of our rules (that lead to δ-nodes). To encode this, we define a δ_{src} function as,

Definition 3.5.1 $\delta_{src} : \mathcal{L} \mapsto \mathbb{B}$ *such that*

$$\delta_{src}(\ell) = \textit{true} \Leftrightarrow \exists \ell' \in \mathcal{L}.\, \exists o \in \mathcal{O}.\, \delta(\ell') \wedge \ell \overset{o}{\nrightarrow} \ell' \wedge P(\ell \overset{o}{\rightarrow} \ell')$$

where again $P(\ell \overset{o}{\rightarrow} \ell')$ indicates the possibility of an edge $\ell \overset{o}{\rightarrow} \ell'$ being created during our original conception of the flow-sensitive analysis (Figure 3.9) due to on-the-fly call graph resolution. We must save versions for δ- and δ_{src}-nodes for the new `[CALL]` and `[RET]` (Figure 3.12) rules as they access versions stored at both such nodes.

3.5.5 Parallelisation

When the SVFG is split into o-SVFGs, versioning becomes an embarrassingly parallel problem. Work can be divided amongst multiple threads such that each one takes on a single o-SVFG. Each thread would pick an o-SVFG, perform (qualified) strongly connected component detection, would check for isomorphism (copying relevant versions and version reliances and moving on to the next o-SVFG if successful), perform meld versioning, determine and save version reliances, and finally permanently save some versions as described previously. Finally, after all o-SVFGs have been processed, all indirect value-flow edges can be deleted from the SVFG, freeing some memory for the main analysis to consume. The data structures used to perform versioning can be designed in such a way to ensure minimal locking which we discuss in Section 3.6.3.

3.6 Evaluation

This section describes our experiments comparing the performance and memory usage of our approach (VSFS) against SFS. The SVF framework [Sui and Xue, 2016b] contains an implementation of SFS as described in Section V of the original paper [Hardekopf and Lin, 2011] (i.e., without the authors' extra optimisations), and we implement our new versioned analysis alongside it for comparison. The points-to sets of both analyses are bit-vectors (with trailing zeroes stripped) which is in contrast to the original SFS implementation which used BDDs [Hardekopf and Lin, 2011].

The auxiliary analysis used is a flow-insensitive inclusion-based analysis boosted by wave propagation [Pereira and Berlin, 2009]. We use the core

Table 3.1: Time taken (s) and memory usage (GB) of SFS and VSFS.

Program	SFS		VSFS		Speedup	Memory reduction
	Time	Memory	Time	Memory		
dhcpcd	21.21	1.06	8.75	0.73	2.43×	1.46×
bash	123.72	5.60	27.12	2.01	4.56×	2.78×
gawk	533.41	10.31	134.63	4.20	3.96×	2.45×
mutt	268.19	12.34	55.43	3.58	4.84×	3.44×
lynx	1692.75	24.50	370.10	6.62	4.57×	3.70×
sqlite	779.47	20.10	235.26	10.75	3.31×	1.87×
xpdf	3429.98	83.20	573.71	15.96	5.98×	5.21×
emacs	OOM	OOM	4518.31	96.44	–	≥1.24×
git	OOM	OOM	4710.71	57.98	–	≥2.07×
kakoune	OOM	OOM	1786.74	37.12	–	≥3.23×
squid	OOM	OOM	OOM	OOM	–	–
wireshark	OOM	OOM	2409.09	50.01	–	≥2.40×
Geo. mean					4.09×	≥2.51×

bit-vector data structure (described in Section 4.1.2) and the bitwise-or operator defined upon it during the versioning phase for the labels/versions and the meld operator, respectively. After versioning (for each o-SVFG) we convert these core bit-vector versions into integers to save on space and on time (when performing comparisons). At nodes which yield what they consume, we only store versions once. We do not create separate o-SVFGs, rather we take advantage of how SVF condenses indirect value-flow edges and use the SVFG directly (Section 2.3.2). When processing an individual o-SVFG, we ignore (condensed) indirect value-flow edges whose label set does not include o. During this process, the actual SVFG is never modified, causing no issue when we parallelise versioning. We check for o-SVFG isomorphism for the o_1-SVFG and the o_2-SVFG by checking if the labels o_1 and o_2 appear on the same set of (condensed) indirect value-flow edges in the SVFG. Our versioning implementation implements all the optimisations described in Section 3.5 except that we evaluate and discuss multithreading separately in Section 3.6.3.

Overall, we see that since VSFS sees significant 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 points-to sets. We also find that the versioning phase benefits significantly from multithreading.

Table 3.2: Time (s) breakdown for VSFS.

Program	Auxiliary analysis	Memory SSA construction	SVFG construction	Versioning	Main analysis	Total
dhcpcd	2.41	0.37	0.11	0.76	4.91	8.75
gawk	16.08	3.85	0.95	21.14	90.38	134.63
bash	5.42	2.33	0.91	12.51	5.34	27.12
mutt	8.67	3.56	1.23	18.91	21.77	55.43
lynx	81.85	12.60	2.24	41.73	229.17	370.10
sqlite	78.02	14.87	2.03	45.98	90.59	235.26
xpdf	42.14	10.33	1.51	99.87	410.56	573.71
emacs	597.67	170.44	5.84	697.55	2989.19	4518.31
git	203.51	291.23	12.25	825.52	3355.20	4710.71
kakoune	93.24	48.82	3.45	273.36	1350.00	1786.74
squid	OOM	OOM	OOM	OOM	OOM	OOM
wireshark	102.14	17.25	3.70	247.06	2024.09	2409.09

3.6.1 Time

Statistics for time, from the start of the analysis onward (including the auxiliary analysis for example), are presented in Table 3.1 (in seconds). We note versioning here was performed singlethreaded. We see that on average we have a 4.09× speedup, a significant improvement over SFS. xpdf sees the greatest improvement with a 5.98× speedup. Unfortunately, we are missing many speedup values since 4 benchmarks could not be analysed with SFS within the memory limit of 120 GB and squid could not be analysed with neither SFS nor VSFS within the memory limit. If memory was available, we expect these benchmarks to see significant speedup, perhaps greater than average.

For context, Table 3.2 shows how much time each element of VSFS took (the total may differ slightly from the sum of the parts shown as we have omitted some less significant elements). We see that the main analysis almost always makes up the greatest proportion of the analysis. The versioning phase once takes up more time than the main analysis (bash) but this will be rectified (if it can even be regarded as a problem considering the overall speedup) with multithreading in Section 3.6.3.

3.6.2 Memory Usage

Statistics for memory (in GB) are also presented in Table 3.1. On average, we have a ≥2.51× reduction in memory usage. Of our benchmarks, 5 could not be analysed by SFS within the 120 GB memory limit (of which only squid

Table 3.3: Versioning (and total) time (s) for VSFS when versioning is performed with 1, 2, and 4 threads.

Program	1 thread	2 threads		4 threads	
	Versioning (total) time	Versioning (total) time	Versioning (total) speedup	Versioning (total) time	Versioning (total) speedup
dhcpcd	0.76 (8.75)	0.42 (8.26)	1.81× (1.06×)	0.27 (8.29)	2.83× (1.05×)
bash	12.51 (27.12)	6.83 (21.40)	1.83× (1.27×)	3.95 (18.49)	3.17× (1.47×)
gawk	21.14 (134.63)	11.18 (122.81)	1.89× (1.10×)	6.44 (119.93)	3.28× (1.12×)
mutt	18.91 (55.43)	10.42 (47.09)	1.81× (1.18×)	6.03 (42.89)	3.14× (1.29×)
lynx	41.73 (370.10)	22.71 (342.04)	1.84× (1.08×)	13.23 (335.55)	3.16× (1.10×)
sqlite	45.98 (235.26)	24.51 (216.08)	1.88× (1.09×)	14.40 (207.03)	3.19× (1.14×)
xpdf	99.87 (573.71)	56.54 (525.82)	1.77× (1.09×)	32.50 (500.93)	3.07× (1.15×)
emacs	697.55 (4518.31)	388.50 (4252.52)	1.80× (1.06×)	225.89 (4070.25)	3.09× (1.11×)
git	825.52 (4710.71)	459.80 (4329.47)	1.80× (1.09×)	272.39 (4136.60)	3.03× (1.14×)
kakoune	273.36 (1786.74)	158.30 (1697.27)	1.73× (1.05×)	103.38 (1611.43)	2.64× (1.11×)
squid	OOM	OOM	–	OOM	–
wireshark	247.06 (2409.09)	138.40 (2288.33)	1.79× (1.05×)	99.37 (2246.54)	2.49× (1.07×)
Geo. mean			1.81× (1.10×)		3.00× (1.15×)

could not be analysed by VSFS within the memory limit either). Thus, we cannot obtain a more accurate average memory reduction and must suffice with using the $\geq$ sign. The reduction ranges from 1.24× to 5.21× at most, but again, this may not paint an accurate picture. For example, analysing emacs ($\geq$1.24× memory reduction) with VSFS takes 96.44 GB so we expect SFS to use far more than 120 GB based on the pattern presented by the other benchmarks. Overall, this is a pleasing result indicating that the reduction in points-to sets does make a difference in memory usage.

3.6.3 Multithreading

Table 3.3 presents the versioning time in seconds (as well as the new total time in parentheses) when using 1, 2, and 4 threads. Memory is omitted for brevity as we only observed standard variance. Most impressive is the almost 2× (1.81×) average speedup in versioning when moving from 1 thread to 2 threads, almost reaching the theoretical best speedup. When moving to 4 threads, we do not see quite as impressive as a result, with a 3× average improvement but a good improvement nonetheless. The total time for the analysis also improves slightly as we see a 1.15× speedup on average using 4 threads for versioning rather than 1.

In our implementation, we have 3 mutexes, 1) to lock the worklist containing our objects to select an *o*-SVFG to work on, 2) to lock the data structure used to check for *o*-SVFG isomorphism (map lookup), and 3) to lock the data structure used to store versions at SVFG nodes which need them. Our 2nd and

(moreso) 3rd usage of mutexes could be improved a little which may bring a slight improvement in speedup. This is likely the cause for the less impressive speedup when using 4 threads (i.e., not as close to $4\times$ as we would like it to be).

Overall, implementing multithreading for versioning is extremely straightforward after a singlethreaded implementation is available. Importantly, this introduces multithreading to an aspect of the points-to analysis very simply unlike the complexity currently required to implement multithreading in the main phase [Zhao et al., 2018], and in either case, it appears that these approaches could work together. In the remainder of this dissertation, we will always be using VSFS (with 4-thread versioning) to evaluate our techniques.

3.7 Related Work

In Section 3.4 of their work, Lhoták and Chung [2011] 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 different merge points produce and equivalent points-to set (recall the $\kappa_1 \diamond \kappa_2$ version in the motivating example in Section 3.1). 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 points-to constraint solving can grow much faster than either label allocation or versioning.

Most similar to our work (particularly in the mode of operation of meld versioning) is *points-to graph equivalence* introduced by Hardekopf and Lin [2009] in their work on semi-sparse analysis. However, their approach is less precise in that they conservatively determine equivalences of entire IN and OUT sets. Thus, this approach cannot realise some of the same opportunities to collapse objects' points-to sets as our approach. For example, if only the points-to set of o differs from amongst multiple objects in the IN sets of ℓ and ℓ', our approach will ensure ℓ and ℓ' consume the same version for all non-o objects, whereas for points-to graph equivalence, the IN sets are considered different and thus no sharing can occur. Further, their pre-analysis is per-

formed 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.

Our versioning is an instance of offline variable substitution [Rountev and Chandra, 2000] 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 [Hardekopf and Lin, 2007b, 2009; Lhoták and Chung, 2011; Smaragdakis et al., 2013] and our technique bears an algorithmic similarity to some previous work [Hardekopf and Lin, 2007b, 2009].

3.8 Conclusion

In this chapter we have presented an object versioned flow-sensitive points-to analysis or versioned staged flow-sensitive analysis (VSFS), an improvement over staged flow-sensitive analysis (SFS) with benefits in 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 reducing storage and propagation costs. We also define a set of optimisations to make the versioning phase more efficient, including the introduction of parallelisation at no detriment to implementation simplicity. On average, in analysing our benchmarks, compared to SFS, our approach presents a speedup of over $4\times$ and a reduction in memory usage of $\geq 2.51\times$.

Compacting Points-To Sets 4

As we have seen so far, points-to analysis makes heavy use of sets and set operations (largely the union operation); the analysis revolves around points-to sets. It is thus incumbent to represent these sets well.

Bit-vectors are extremely compact sets of integers which have seen wide adoption as points-to sets in points-to analysis tools, appearing in the likes of Soot [Lhoták and Hendren, 2003], WALA [WALA, 2021], SVF [Sui and Xue, 2016b], and more [Hardekopf and Lin, 2007b]. In the literature, they have been shown to be more efficient than hash-based sets and sorted arrays [Lhoták and Hendren, 2003] and at times more efficient than *binary decision diagram* (BDD) representations [Hardekopf and Lin, 2007b]. With regards to BDDs specifically, BDD-based points-to analysis often requires difficult variable reordering to be efficient, and benefits may not outweigh those of using explicit representations such as B-trees [Bravenboer and Smaragdakis, 2009]. Bit-vectors are extremely easy to implement and use in points-to analysis and without any algorithmic change, only require a bijective mapping of abstract memory objects to integral identifiers. Then points-to sets would be made up of the identifiers which map to the pointed-to objects.

A bit-vectors is simply a contiguous array of words (the native size which the *instruction set architecture* (ISA) operates on). Each bit in a word represents an integral identifier and a set bit means that identifier is in the data structure. Concretely, the first bit of the first word represents 0, the second bit of the first word represents 1, and so on. For example, given a word size of

4 bits ($\langle \times \times \times \times \rangle$ where each $\times$ represents a bit of the word), as a bit-vector, the set $\{0, 3, 8, 9\}$ can be represented with an array of three words:

$$\begin{array}{l} [\,\langle 1001\rangle, \langle 0000\rangle, \langle 1100\rangle\,]. \\ \{\ 0 \quad 3 \qquad\qquad 89 \quad\ \} \end{array}$$

In this instance, the bits representing 0, 3, 8, and 9, are set to 1, and the remaining bits are 0. This efficiency in representing identifiers using bits, rather than entire words differentiates bit-vectors from explicitly represented sets. Bit-vectors are more efficient than their explicit counterparts in space in cases where redundant (zero-)words are few. Here, we use 3 words, rather than 4 as would be required in an explicitly represented data structure such as an array or hash-based set, to represent the 4 identifiers. On such bit-vectors, operations are very efficient and can take advantage of spatial locality and vectorisation. For example, the union operator becomes a linear bitwise-or between every pair of words in the same position in two bit-vectors.

Unfortunately, in points-to analysis, zero-words can often occur. Even, in our example above, the second (zero-)word is completely redundant. Worse still, consider the amount of redundant zero-words required for a very plausible points-to set such as $\{2, 5000\}$ which would make such a representation less practical. Sparse bit-vectors can solve this problem by omitting non-zero words and linking words through a linked list. Each non-zero word is paired with an offset representing the integral identifier which the first bit of that word refers to. Our example above would then become

$$\{\, 0\ \langle 1001\rangle \,\} \rightarrow \{\, 8\ \langle 1100\rangle \,\} \rightarrow \mathit{nil},$$

thus avoiding maintaining the zero-word. Despite removing redundant zero-words, sparse bit-vectors introduce other problems such as the loss of spatial locality and vectorisation opportunity, and the need for extra metadata in the way of links and offsets. We observe that the downsides of using either type of bit-vector for points-to analysis can be substantially minimised if the objects in the same points-to sets (or "co-pointees") have numerically close identifiers in the object-to-identifier mapping. That is, we want to make full use of each word, i.e., reduce the number of zero-**bits**, to improve both types of bit-vectors (though we will focus exclusively on contiguous bit-vectors in this chapter, our approach can be, and has been, successfully applied to both [Barbar and

Sui, 2021a]). For example, the objects being mapped to 0, 3, 8, and 9, would be ideally mapped as 0, 1, 2, 3, requiring only a single word in a sparse or contiguous bit-vector.

Developing such an ideal mapping is worthwhile since it can produce more compact points-to sets which would improve both sparse and contiguous bit-vectors and subsequently improve the performance of a points-to analyses built upon them. However, points-to relations are unpredictable making it extremely challenging to predict an ideal object-to-identifier mapping that accommodates every pointer's points-to set.

We require some indication of the points-to relations which will be discovered by the analysis. The auxiliary analysis in a staged analysis (like SFS and VSFS) provides us with a sound over-approximation of this for a more expensive analysis (whose optimisation is more important). We can thus *cluster pointees* such that co-pointees—per the auxiliary analysis—are given numerically close identifiers. We find that, despite the auxiliary analysis being an over-approximation of the main analysis, a mapping that works well for the auxiliary analysis also works well for the main analysis. The result is that bit-vectors are smaller, using far fewer words, making representation more compact and operations faster, improving the main analysis in space and time.

In this chapter, we first go over some background on representing points-to sets as bit-vectors, followed by the introduction of the core bit-vector, an improved version of the contiguous bit-vector. We then formulate our approach as an integer programming problem where we aim to produce a better object-to-identifier mapping, optimising for compact points-to sets by generating constraints from the auxiliary analysis's results. However, integer programming is expensive, so we then explore doing so with agglomerative clustering, a more approximate approach. Popular in data mining [Maimon and Rokach, 2005], agglomerative clustering helps us produce a good mapping with negligible overhead. We then end the chapter by evaluating our approach upon VSFS and discussing related work. Most of the work in this chapter has been published at OOPSLA [Barbar and Sui, 2021a].

4.1 Representing Points-To Sets as Bit-Vectors

In this section, we describe the internal workings of contiguous and sparse bit-vectors, and their performance issues, in time and space, when points-to sets contain objects whose mapped-to identifiers are not numerically close.

4.1.1 Contiguous and Sparse Bit-Vectors

Contiguous bit-vectors are implemented as arrays of words. In the best-case where words are used to the fullest, a bit-vector can represent a set a of i integers within the range 0 to n in $\lceil \frac{i}{\mathcal{W}} \rceil$ words, where $\mathcal{W}$ is the word size, in bits, of the ISA. In the worst case, that same set would be represented in $\lceil \frac{n}{\mathcal{W}} \rceil$ words. Bitwise operations upon the entire bit-vector, such as *or* and *not*, could be implemented by performing the operation on individual words (and the corresponding word of the other bit-vector operand in the case of binary operations). Fortunately, many of these bitwise operations map directly to set operations, such as bitwise-or being an analogue to the union operation on a set of integers. Such operations are amenable to vectorisation and can take advantage of the spatial-locality afforded by the contiguous nature of the words. Concretely, the bitwise-or operation ($|$), analogous to the most important operation in points-to analysis, of two bit-vectors of size n made up of words $w_1, ..., w_n$ and $w'_1, ..., w'_n$, respectively, would be given by

$$
\begin{array}{cc}
 & [\quad w_1, \quad w_2, \quad \dots, \quad w_n \quad] \\
| & [\quad w'_1, \quad w'_2, \quad \dots, \quad w'_n \quad] \\
\hline
 & [\quad w_1 \mid w'_1, \quad w_2 \mid w'_2, \quad \dots, \quad w_n \mid w'_n \quad].
\end{array}
$$

For bitwise-or, if one operand is longer than the other such that the longer one contains m more words, the shorter one can be regarded as being padded at the end with m zero-words since in the resulting bit-vector the last m words would be verbatim the last m words of the longer bit-vector. It should be noted that some implementations require bit-vectors to be statically sized (e.g., C++'s `std::bitset` [ISO/IEC, 2017]) while others (e.g., that implemented by LLVM [LLVM BitVector, 2021]) strip trailing zero-words by maintaining the length of the bit-vector as extra metadata. When we use the term bit-vector or standard bit-vector, we refer to the latter and only implicitly represent the length (we will regard any array $[\dots]$ as having an implicit length field).

To use bit-vectors as points-to sets, we need to assign abstract memory objects, our pointees, to integral identifiers. For example, consider we have 10 000 memory objects (o_0 to o_{9999}), a quantity not foreign when analysing typical programs, and that the points-to sets of some pointers p and q, at some

point during an analysis, are

$$pt(p) = \{o_1, o_{4500}, o_{9999}\}, \text{and}$$
$$pt(q) = \{o_1, o_4, o_8\}.$$

And let us consider a naive mapping of each object to its subscript such that $o_n \mapsto n$. As bit-vectors, our two points-to sets would be represented as

$$pt(p) = [\,\langle 01_1 00\rangle, \ldots, \langle 1_{4500} 000\rangle, \ldots, \langle 0001_{9999}\rangle\,], \text{and}$$
$$pt(q) = [\,\langle 01_1 00\rangle, \langle 1_4 000\rangle, \langle 1_8 000\rangle\,],$$

where the word size $\mathcal{W}$ equals 4 and we subscript set bits with the identifier they correspond to for readability. We see that $pt(p)$ is impractically large, requiring 2500 words (10 000 bits) for a set of just 3 pointees. This is precisely the worst case number of words required of $\lceil \frac{n}{\mathcal{W}} \rceil = \lceil \frac{10000}{4} \rceil$ where the best case would be $\lceil \frac{3}{4} \rceil$. $pt(q)$ is more reasonable despite still being imperfect, requiring 3 words (12 bits) for 3 pointees, though they would fit into 1 word ($\lceil \frac{3}{4} \rceil = 1$). Furthermore, if we were to include $pt(p)$ in $pt(q)$ ($pt(p) \subseteq pt(q)$), $pt(q)$ would grow to the size of $pt(p)$, making us then represent both $pt(p)$ and $pt(q)$ with 10 000 bits each for sets with 3 and 5 pointees (whose corresponding set bits are spread amongst just 3 and 5 words respectively). This is impractical and makes bit-vectors a risky choice given a poor mapping.

Sparse bit-vectors aim to overcome the abundance of zero-words by only storing words which have at least one set bit. Since there are now "holes" in the data structure, we can no longer use contiguous memory and need to mark each word with an offset indicating the integral value which the first bit in that word represents (the word part can be a fixed-size array of words to incorporate some of the benefits afforded by standard bit-vectors). The offset/word pairs are often linked together in a linked list to aid arbitrary insertion which is necessary for efficient in-place bitwise-or operations.

Now we can represent $pt(p)$ and $pt(q)$ using sparse bit-vectors as

$$\begin{aligned} pt(p) &= \{\, 0\ \langle 01_1 00\rangle \,\} \to \{\, 4500\ \langle 1_{4500} 000\rangle \,\} \\ &\quad \to \{\, 9996\ \langle 0001_{9999}\rangle \,\} \to \textit{nil}, \text{and} \\ pt(q) &= \{\, 0\ \langle 01_1 00\rangle \,\} \to \{\, 4\ \langle 1_4 000\rangle \,\} \to \{\, 8\ \langle 1_8 000\rangle \,\} \to \textit{nil}, \end{aligned}$$

thus requiring 3 words per bit-vector (and some extra metadata) and freeing

us of all zero-words. Representation of $pt(p)$ has improved drastically, but that of $pt(q)$ has become worse when we account for the extra metadata in the way of offsets and links. However, if we were to fulfil our $pt(p) \subseteq pt(q)$ constraint, then the resulting $pt(q)$'s representation has improved as it would only require 5 words (and some metadata) to represent, not 2500 words. Of the disadvantages of representing reasonably sized bit-vectors as sparse bit-vectors is the loss of vectorisation, spatial locality, the extra logic required to perform operations, and the extra metadata required. The bitwise-or operation, presented in Algorithm 1, is more complicated and is now riddled with conditionals (e.g., lines 3 and 6), indirect references (e.g., lines 5 and 8), and heap allocations for nodes (e.g., line 4). Thus, in practice, sparse bit-vectors can be less efficient than their contiguous counterpart in addition to the potential for extra memory use due to metadata.

The disadvantages of using both contiguous and sparse bit-vectors are amplified by a poor object-to-identifier mapping. Due to the unpredictability of points-to relations, an object-to-identifier mapping better than ensuring that the n memory objects of a program are given identifiers 0 to n is difficult. If we assume that the objects present in $pt(p)$ and $pt(q)$ are not present in any other points-to set it would be ideal to remap these objects to identifiers which are close to each other. For example, consider the mapping

$$o_1 \mapsto 0 \quad o_{4500} \mapsto 1 \quad o_{9999} \mapsto 2 \quad o_4 \mapsto 3 \quad o_8 \mapsto 4,$$

and the subsequent representation as bit-vectors rather than sparse bit-vectors:

$$\begin{aligned} pt(p) &= [\,\langle 1_0 1_1 1_2 0\rangle\,], \text{and} \\ pt(q) &= [\,\langle 1_0 001_3\rangle, \langle 1_4 000\rangle\,]. \end{aligned}$$

Now, we have represented $pt(p)$ with a single word, as a bit-vector, and $pt(q)$ with 2 words, also as a bit-vector. After fulfilling the $pt(p) \subseteq pt(q)$ constraint, the result would remain compact with $pt(q)$ becoming $[\,\langle 1_0 1_1 1_2 1_3\rangle, \langle 1_4 000\rangle\,]$, and in fact would be a best-case representation, yielding much better performance.

However, in reality, these objects may occur in multiple points-to sets and have requirements of closeness to many other objects such that achieving an ideal mapping is far more complicated. In fact, a perfect mapping could be impossible. To address this, we explore techniques such as integer program-

ming and hierarchical clustering to find a superior mapping by leveraging the over-approximate points-to data produced by the auxiliary analysis. But first, we briefly describe an improved contiguous bit-vector that can better take advantage of a better object-to-identifier mapping.

Algorithm 1: Bitwise-or of two sparse bit-vectors.

Input : s_1, s_2 – sparse bit-vectors (linked lists of $\{\mathit{offset},\ \mathit{word},\ \mathit{next}\}$ structures).
Output: u – bitwise-or of s_1 and s_2.

```
1  c1 ← s1; c2 ← s2;
2  while c1 ≠ nil ∧ c2 ≠ nil do
3      if c1.offset = c2.offset then
4          u.append(new { c1.offset ⟨c1.word | c2.word⟩ });
5          c1 ← c1.next; c2 ← c2.next;
6      else if c1.offset < c2.offset then
7          u.append(new { c1.offset ⟨c1.word⟩ });
8          c1 ← c1.next;
9      else
10         u.append(new { c2.offset ⟨c2.word⟩ });
11         c2 ← c2.next;
12     end
13 end
   // c1 or c2 may not yet be nil; append remainder.
14 while c1 ≠ nil do
15     u.append(new { c1.offset ⟨c1.word⟩ });
16     c1 ← c1.next;
17 end
18 while c2 ≠ nil do
19     u.append(new { c2.offset ⟨c2.word⟩ });
20     c2 ← c2.next;
21 end
```

4.1.2 Core Bit-Vector

Let us take a look at the *core bit-vector*, our improved version of the standard bit-vector. It has a compact representation like a contiguous bit-vector (little

metadata) while reducing many zero-words by also stripping leading zero-words, unlike standard bit-vectors which only strip trailing zero-words.

Consider a points-to set with 5 objects whose identifiers map to 9995, 9996, 9997, 9998, and 9999 and that the native word size $\mathcal{W}$ is 4. These identifiers would ideally fit within two words. As a sparse bit-vector, this points-to set is

$$\{\, 9992\ \langle 0001_{9995} \rangle \,\} \rightarrow \{\, 9996\ \langle 1_{9996}\, 1_{9997}\, 1_{9998}\, 1_{9999} \rangle \,\} \rightarrow nil,$$

using just two words and associated metadata. However, as a bit-vector, this points-to set would look like

$$[\, \langle 0000 \rangle, \ldots, \langle 0000 \rangle, \langle 0001_{9995} \rangle, \langle 1_{9996}\, 1_{9997}\, 1_{9998}\, 1_{9999} \rangle \,],$$

which contains 2500 words with only two meaningful words. The problem is that bit-vectors contain an explicit representation (of zero) for the integers 0 to the first non-zero bit, no matter how far that may be.

To remedy this we use *core bit-vectors* which use an offset similar to sparse bit-vectors to state the bit of the first word and strips leading and trailing zero words to maintain only the "core". Note, however, that unlike a sparse bit-vector, zero words may occur between the first and last (non-zero by definition) words. For example, the points-to set above would be represented as

$$\{\, 9992\ [\, \langle 0001_{9995} \rangle, \langle 1_{9996}\, 1_{9997}\, 1_{9998}\, 1_{9999} \rangle \,] \,\}$$

saving us 2498 words. If we were to insert identifier 9984, it would become

$$\{\, 9984\ [\, \langle 1_{9984} 000 \rangle, \langle 0000 \rangle, \langle 0001_{9995} \rangle, \langle 1_{9996}\, 1_{9997}\, 1_{9998}\, 1_{9999} \rangle \,] \,\}$$

which uses only one word more than an equivalent sparse bit-vector but has the benefit of storing words contiguously and with less metadata—the offset and the (implicit) length—therefore potentially saving both space and time. Algorithm 2 details the bitwise-or operation of two core bit-vectors. In contrast to Algorithm 1, operations are all performed on contiguous arrays, rather than across indirect links, and loops do not contain conditionals, making the process more amenable to vectorisation. When the offset is set to 0, the data structure acts as a standard contiguous bit-vector.

Algorithm 2: Bitwise-or of two core bit-vectors.

Input : c_1, c_2 – core bit-vectors ($\{\mathit{offset}, [\mathit{words}]\}$).
Output: u – bitwise-or of c_1 and c_2.

// Determine which input starts "earlier" and which "later".
if $c_1.\mathit{offset} < c_2.\mathit{offset}$ **then**
  $e \leftarrow c_1$; $l \leftarrow c_2$;
else
  $e \leftarrow c_2$; $l \leftarrow c_1$;
end
$u.\mathit{offset} \leftarrow e.\mathit{offset}$;
$ei \leftarrow 0$; // Index into earlier input.
while $ei < \left\lfloor \frac{l.\mathit{offset}}{\mathcal{W}} \right\rfloor - \left\lfloor \frac{e.\mathit{offset}}{\mathcal{W}} \right\rfloor$ **do**
  if $ei < e.\mathit{words.length}$ **then**
    // Append words in e which are not in l.
    u.append($e.\mathit{words}[ei]$);
  else
    // Nothing left in e; add zero-words till start of l.
    u.append(0);
  end
  $ei \leftarrow ei + 1$;
end
$li \leftarrow 0$; // Index into later input.
// Append bitwise-or of words common to both inputs.
while $ei < e.\mathit{words.length} \wedge li < l.\mathit{words.length}$ **do**
  u.append($e.\mathit{words}[ei]$ | $l.\mathit{words}[li]$);
  $ei \leftarrow ei + 1$; $li \leftarrow li + 1$;
end
// ei **or** li may not have reached end; append remainder.
while $ei < e.\mathit{words.length}$ **do**
  u.append($e.\mathit{words}[ei]$);
  $ei \leftarrow ei + 1$;
end
while $li < l.\mathit{words.length}$ **do**
  u.append($l.\mathit{words}[li]$);
  $li \leftarrow li + 1$;
end

Table 4.1: Time taken (s) and memory usage (GB) for VSFS using standard bit-vectors and core bit-vectors.

Program	Bit-vector		Core bit-vector		Speedup	Memory reduction
	Time	Memory	Time	Memory		
dhcpcd	8.29	0.74	8.08	0.69	1.03×	1.07×
gawk	119.93	4.24	128.45	4.09	0.93×	1.04×
bash	18.49	2.03	18.31	1.92	1.01×	1.05×
mutt	42.89	3.60	40.26	3.49	1.07×	1.03×
lynx	335.55	6.64	324.07	5.64	1.04×	1.18×
sqlite	207.03	10.83	202.64	10.38	1.02×	1.04×
xpdf	500.93	15.99	509.47	15.40	0.98×	1.04×
emacs	4070.25	97.30	4043.31	93.86	1.01×	1.04×
git	4136.60	58.54	4085.54	50.78	1.01×	1.15×
kakoune	1611.43	37.19	1592.13	35.07	1.01×	1.06×
squid	OOM	OOM	OOM	OOM	–	–
wireshark	2246.54	49.98	2179.94	37.41	1.03×	1.34×
Geo. mean					1.01×	1.09×

4.1.2.1 Effectiveness

We have implemented our core bit-vector with a `std::vector` from the C++ STL. For ease of implementation, our standard bit-vector (which we have been using throughout this dissertation) is a core bit-vector with the offset set to 0. The slither of overhead that introduces should be indiscernible. Finally, since our processor is a 64-bit processor, our native word size is 64 bits.

Table 4.1 shows the time and memory required to analyse our benchmarks with VSFS using a standard bit-vector and our core bit-vectors[1]. We see there is no practical difference in time (two regressions do appear, which from our experience, fall roughly within the margin of error). As for memory, where the margin of error is much smaller, we do see a small improvement overall. On average, using core bit-vectors reduces memory usage by 1.09×, and is quite significant for some benchmarks such as git and wireshark where about 8 and 12 gigabytes are saved, respectively. Moving onward, we will use core bit-vectors for all experiments.

[1]In previous work [Barbar and Sui, 2021a], we have also compared standard and core bit-vectors against sparse bit-vectors and found that the former *usually* perform better than the latter, with the exceptions being in the presence of a "unlucky" naive, especially, or an unideal object-to-identifier mapping.

4.2 Compacting Points-To Sets

Assigning memory objects to identifiers based on points-to relations can be framed as an optimisation problem, making *integer programming* (IP) well suited. Unfortunately, the resulting integer programs are computationally expensive to solve, but the formulation helps clarify the problem and its optimal solution. We first formulate compacting points-to sets as an optimisation problem solved by IP in Section 4.2.1. Then, we present a more approximate and much more efficient approach in Section 4.2.2. There we provide a brief introduction to hierarchical clustering and then detail how we can apply this technique to objects in points-to sets, concluding with two simple optimisations to improve efficiency and the resulting mapping.

4.2.1 Integer Programming Formulation

It is hard to determine a good identifier mapping before any points-to analysis is performed as it is unknown which objects are pointed to by the same pointers and should thus be assigned close identifiers. Inspired by staged analysis [Hardekopf and Lin, 2011], we use an auxiliary analysis as a good indication of which objects may occur in the same points-to sets, given that the auxiliary analysis's points-to sets are supersets of the main analysis's. Therefore, creating a mapping for the auxiliary analysis should create a good mapping for the main analysis, or at least, better than a blind or random mapping.

We let the constant $\mathcal{W}$ be the word size of the ISA (in bits), and for each points-to set $P = \{o_{x_1}, o_{x_2}, \dots, o_{x_n}\}$ produced by the auxiliary analysis, we have the following known integer variables:

1. n is the number of objects in P, and
2. $w = \lceil \frac{n}{\mathcal{W}} \rceil$ is the minimum number of words required for a bit-vector representation of P in the, potentially impossible, best case scenario.

And the following unknown integer variables:

1. m_{x_i}, where $1 \leq i \leq n$, is the identifier object o_{x_i} will be assigned to in our new mapping, and
2. f is some offset multiplier for where the identifiers, m_{x_i}, start.

Our goal is to produce mappings from o_{x_i} to m_{x_i} such that the least possible number of words can represent all points-to sets (as close as possible

to each w) as core bit-vectors. In the best case scenario, each pair of objects in each points-to set are assigned identifiers within w of each other, that is, $\frac{\left|m_{x_i} - m_{x_j}\right|}{\mathcal{W}} < w$ for $1 \leq i \leq n$ and $1 \leq j \leq n$, with the minimum m_{x_i} assigned to a multiple of $\mathcal{W}$, i.e., $f \cdot \mathcal{W}$. That is, the minimum m_{x_i} is aligned on a word boundary. Aligning to $\mathcal{W}$ is important to avoid co-pointees being mapped to identifiers within w of each other, but needlessly crossing word boundaries. For example, given $\mathcal{W} = 4$ and $pt = \{o_1, o_2\}$, it would be unoptimal to assign o_1 to 7 and o_2 to 8 despite that $\frac{|m_1 - m_2|}{\mathcal{W}} = \frac{|7-8|}{4} < 1$ since we would require 2 words in a bit-vector instead of 1 (which can be achieved with various assignments such as that of o_1 to 2 and o_2 to 3). This produces the best possible allocation because it ensures points-to sets, as bit-vectors, are as small as they can possibly be (of size w), if the problem allows.

We need to determine the best possible values for m and f to achieve this. As an integer program, we can encode these requirements as the following constraints for each points-to set $P = \{o_{x_1}, o_{x_2}, \ldots, o_{x_n}\}$:

$$\begin{aligned} m_{x_i} &\geq f_P \cdot \mathcal{W} \\ m_{x_i} &< f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W} \\ f_P &\geq 0, \end{aligned} \tag{C1}$$

which ensures that all objects in a points-to set are mapped to identifiers between some starting offset and within the minimum number of words required to represent that points-to set. We also require that objects are uniquely mapped to identifiers, so for all pair of identifiers m_i, m_j, where $i \neq j$,

$$\left|m_i - m_j\right| > 0, \tag{C2}$$

which can be expressed as the constraints

$$\begin{aligned} m_i - m_j &< L \cdot b_{ij}, \\ m_i - m_j &> -L + L \cdot b_{ij} \\ b_{ij} &\geq 0 \\ b_{ij} &\leq 1, \end{aligned} \tag{C3}$$

where L is a constant set to be larger than any possible identifier, and b_{ij} are Boolean variables (as forced by the constraints upon them). This construct

asserts that $m_i - m_j$ must never be zero, i.e., that they are not equal. When b_{ij} is 0, the first two constraints ensure the difference is in the open range $(-L, 0)$, and when set to 1, they ensure the difference is in the open range $(0, L)$.

Unfortunately, such an integer program may be impossible to solve due to the pigeonhole principle. Consider that an object o (which we will map to m) occurs in $\mathcal{W}$ points-to sets, none of which contain common elements aside from o, each of which with $w = 1$ and $n > 1$. It would not be possible for the m corresponding to o to be assigned within $\mathcal{W}$ of all of the identifiers assigned to o's co-pointees and be in the same word. Rather, it can only be in the same word as $\mathcal{W} - 1$ of its co-pointees. The $\mathcal{W}$-th co-pointee must be, at best, in a word before or after that which m falls into. To remedy this, we provide a "tolerance" for each points-to set to extend the range of identifiers the objects of a points-to set can fall in. Thus we introduce another unknown for each points-to set:

3. t is a tolerance multiplier for where the identifiers assigned to objects of a points-to set can end.

Thus, for each pointee o_{x_i} in points-to set $P = \{o_{x_1}, o_{x_2}, \dots, o_{x_n}\}$, we reframe the constraints in C1 as (changes underlined):

$$\begin{aligned} m_{x_i} &\geq f_P \cdot \mathcal{W} \\ m_{x_i} &< f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W} \underline{+ t_P \cdot \mathcal{W}} \\ f_P &\geq 0 \\ t &\geq 0. \end{aligned} \tag{C4}$$

Again, the first constraint ensures that m_{x_i} begins aligned at some offset into the identifier space ($f_P \cdot \mathcal{W}$). The second ensures that it does not extend w words away from the offset ($f_P \cdot \mathcal{W} + w_P \cdot \mathcal{W}$) as before, but now we allow for some tolerance ($t_P \cdot \mathcal{W}$). As before we must also ensure that the identifiers are unique (Constraints C2/C3). This set of constraints becomes trivial to solve with absurdly large tolerances and circumvents our goal of small points-to sets. Thus, our objective is to minimise the tolerances as they introduce extra words over the (potentially impossible) ideal. That is, we minimise $t_{P_1} + \dots + t_{P_N}$, where N is the number of points-to sets we take into consideration, to

give us the least required number of words in a core bit-vector[2].

Proof 4.2.1 *For each points-to set P, representing P as a core bit-vector uses at best w_P words. The constraints encode an identifier mapping which makes each points-to set P fit into w_P words,* ***except*** *that a tolerance for the number of words is allowed. Optimising for minimum tolerance multipliers (i.e. the sum of t_P for all points-to sets P, where each tolerance t_P would then be used in $t_P \cdot W$) ensures that each points-to set is represented with as few words as possible, thus producing an ideal mapping.*

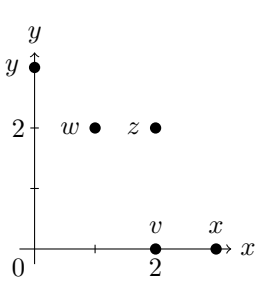
Finally, as mentioned previously, this is too computationally expensive to solve. There are a few ways to make the integer programs produced cheaper to solve, of them is the regioning we introduce to improve clustering in Section 4.2.4 which allows groups of objects to be considered separately, but regardless, IP as a solution to this problem simply does not scale with current techniques for large programs. In fact, we were only able to find a solution with this IP formulation for trivial programs where our approximate solution was able to produce an optimal solution anyway. The remainder of this section is dedicated to solutions which are more approximate but far more scalable.

4.2.2 Hierarchical Clustering

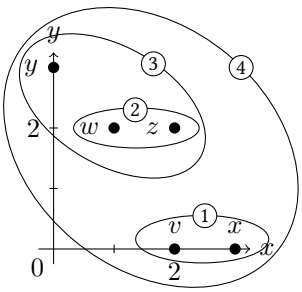
Our aim is to assign pointees which appear in points-to sets together to identifiers which are numerically close together. We find clustering to be a more practical way to achieve this goal. Clustering is the process of grouping close or similar items together given a set of items with some measure of similarity or distance between them such as the Jaccard Index for sets [Jaccard, 1912] or the Euclidean distance for coordinates.

There is no notion of a "perfect clustering", rather success is determined depending on the problem for which clustering is employed to solve. Hierarchical clustering can be agglomerative (bottom-up) or divisive (top-down) [Maimon and Rokach, 2005]. An agglomerative approach starts with clusters of a single item and successively merges clusters until there remains only one cluster, and a divisive approach starts with a single cluster, divides it, and then successively divides clusters until only single-item clusters remain. We

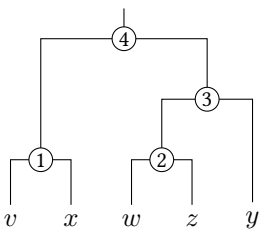
[2]We remind again that this minimal representation is for the points-to sets produced by the auxiliary analysis which may differ from those produced by the main stage.



(a) Items on a plane.



(b) Agglomerative clustering process with the Euclidean distance. The numbers on the clusters represent the order in which they are created.



(c) Resulting dendrogram.

FIGURE 4.1: Agglomerative clustering process (b) of coordinate data (a) and the resulting dendrogram (c).

focus on the former due to the availability of an extremely performant C++ implementation [Müllner, 2013]. Both approaches are hierarchical in that the resulting clusters are made up of smaller clusters. The resulting hierarchy is represented by a dendrogram, a tree where child nodes are the clusters which form parent nodes, also clusters [Maimon and Rokach, 2005]. Leaf nodes are the individual objects themselves.

To show how agglomerative clustering works, we walk through a simple example in Figure 4.1. Consider some items located at coordinates: v at $(2, 0)$, w at $(1, 2)$, x at $(3, 0)$, y at $(0, 3)$, and z at $(2, 2)$, as in Figure 4.1a, and that we want to cluster these items according to their closeness using the Euclidean distance as a measure for that. The process is shown on the same plane in Figure 4.1b. For our example, we set the linkage criterion as the *single* linkage criterion [Murtagh, 1983] such that the distance between two clusters is the minimum distance between any two items in each of those clusters.

To begin, each item is considered to be part of its own cluster, and we

search for the closest two clusters (according to the linkage criterion) and merge them. The pairs v and x, and w and z both have the shortest Euclidean distance between them (1.0), so we randomly pick a pair, say v and x, and merge them as cluster ①. Then we merge the other pair, w and z, as cluster ② since they are now the two closest clusters. The next two closest clusters are now the cluster containing w and z (cluster ②) and the single-item cluster containing y, since w and y have a distance of $\sqrt{(0-1)^2+(3-2)^2} = 1.41$ between them. They are merged as cluster ③. Now we have only two clusters remaining, so we merge them as ④ since they are obviously the closest two clusters, and thus agglomerative clustering is complete. The dendrogram in Figure 4.1c shows the resulting clusters for our example. The entire dendrogram represents one cluster, but if we cut it, we can form multiple clusters. For example, if we cut the dendrogram one level down (i.e., one step backwards from the end of the clustering process), we have two clusters, one with v and x, and one with w, y, and z. From this, we can infer that according to our definition of "close" (the linkage criterion and closeness measure), v and x are close and w, y, and z are close, relatively.

Linkage Criteria In our example, determining the distance between two single-item clusters is straightforward—calculate the Euclidean distance—but determining the distance between two clusters where at least one contains more than one object was more complicated and we required a *linkage criterion*. Being that clustering is an approximate process, multiple linkage criteria exist to optimise for different data types. In this work, we focus on three, single-linkage, complete-linkage, and average-linkage, partly because of the existence of an extremely performant implementation of them [Müllner, 2013] and partly because of how common they are. Clustering with any of these linkage criteria can be implemented in $\mathcal{O}(n^2)$ time in the worst-case given n items to cluster [Maimon and Rokach, 2005; Müllner, 2013]. For our data, clustering is very fast and so we perform clustering with each of these criteria and choose the one which produces the most promising mapping on-the-fly.

Single-Linkage The distance between two clusters under the single-linkage criterion is the smallest distance between any member of one cluster to any member of the other [Murtagh, 1983]. That is, assuming we have a distance function (for our problem, we use the distance defined in Definition 4.2.1) for

two objects, $d : \mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}$, the distance between clusters C_1 and C_2, is

$$\mathit{min}\{d(o_1, o_2) \mid o_1 \in C_1, o_2 \in C_2\}.$$

Complete-Linkage The distance between two clusters under the complete-linkage criterion is the largest distance between any member of one cluster to any member of the other [Murtagh, 1983]. This is similar to the single-linkage criteria, except that it, very harshly, blocks clusters whose members are not *all* close. Formally, with the same d and clusters C_1 and C_2 as before, this is given by

$$\mathit{max}\{d(o_1, o_2) \mid o_1 \in C_1, o_2 \in C_2\}.$$

Average-Linkage Unlike single- and complete-linkage, the distance calculated with average-linkage considers all members of both clusters. The distance given is the average distance from any member of one cluster to any member of the other cluster [Murtagh, 1983]. For clusters C_1 and C_2 and our distance function d, this is given by

$$\frac{1}{|C_1|\,|C_2|} \sum_{o_1 \in C_1} \sum_{o_2 \in C_2} d(o_1, o_2).$$

4.2.3 Clustering Objects

With hierarchical clustering, we can cluster objects in such a way that co-pointees of smaller points-to sets are clustered together early, and hence are regarded as "closer" and thus more suitable for close identifiers. A simple linear scan of points-to sets, and assigning identifiers during the scan, is not sufficient because relationships between pointees are not transitive.

For example, consider the two points-to sets $\{o_a, o_b, o_c, o_d, o_e\}$ and $\{o_a, o_f\}$. If we were to perform a linear scan assigning identifiers sequentially starting from the first points-to set, we would produce the mapping

$$o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2 \quad o_d \mapsto 3 \quad o_e \mapsto 4 \quad o_f \mapsto 5,$$

which forces the first points-to set into two words (which is ideal), but unfortunately, also forces the second points-to set into two words (which is not ideal; it can be represented in one word). Clustering can better understand relationships between pointees allowing us to produce a better mapping.

To cluster objects, we require a more suitable measure of distance between two objects as we define below.

Definition 4.2.1 *(Object distance). The distance between two objects is the minimum number of words required to represent any points-to set in which both objects appear in, as a bit-vector. If they never appear in the same points-to set, their distance is* ∞.

This definition encodes how far apart, word-wise, two objects would be in the theoretical best case scenario (which may be impossible; recall our discussion on the pigeonhole principle and tolerances in Section 4.2.1). We want the clustering process to approximate this theoretical limit as closely as possible.

Revisiting our example, the object distance between o_a and o_f, $d(o_a, o_f)$, is 1 since o_a and o_f appear in two points-to sets, one of which can be represented in 2 words and the other in 1 word, and $min(1, 2) = 1$. $d(o_b, o_f) = d(o_c, o_f) = d(o_e, o_f) = \infty$ since these objects never appear in the same points-to set. The remaining object pairs have a distance of 2 because they appear together only in the first points-to set which requires two words to represent in the ideal case.

With this definition, we can build a distance matrix by calculating the distance for each pair of objects. This can be done by filling the distance matrix with ∞, scanning all points-to sets, and updating the distance between each pair of objects in a points-to set if that points-to set requires fewer words to represent as a bit-vector than is recorded in the distance matrix. For each points-to set, this is a quadratic process in the number of objects in that points-to set. In practice, we only need to consider unique points-to sets since duplicate points-to sets obviously require the same number of words to represent. Our experience is that many points-to sets are duplicates.

With a distance matrix, clustering can be performed to obtain a dendrogram. We are not interested in cutting the dendrogram, rather, we are interested in the relative prioritisation of object pairs that the clustering algorithm found, that is, which objects are closer to each other compared to other objects. For example, in the previous scenario, the clustering algorithm would quickly determine that o_a and o_f must be close together, before determining the closeness between o_a, o_b, o_c, o_d, and o_e. With a counter starting from 0, we can assign objects to identifiers by visiting objects depth-first in the dendrogram. The depth-first search ensures that the clusters within a cluster

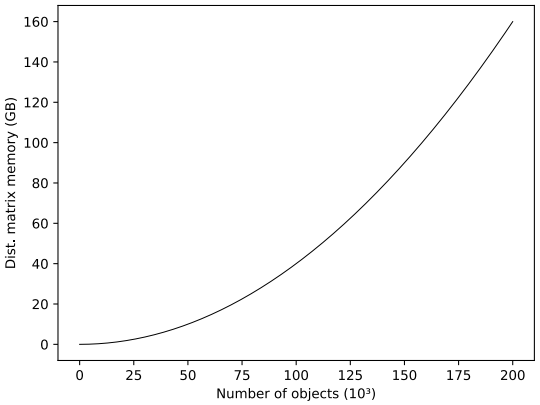


FIGURE 4.2: Memory used to represent various (condensed) distance matrices.

are visited before visiting adjacent, less related clusters. That is, within each cluster with more than one object (including the large cluster of all objects), the identifiers for the objects in that cluster are sequential. This is extremely fast being linear in the size of the dendrogram which has $2n-1$ nodes, where n is the number of objects.

4.2.4 More Efficient Region-Based Clustering

A distance matrix implemented as a literal matrix, rather than as a hash table for example, is more efficient. Since the distance matrix is symmetrical (the distance of o to o' is the same as the distance from o' to o), a condensed distance matrix which is a linear array forming the upper triangle of the matrix [Müllner, 2013] is sufficient. The number of elements in such a matrix would be $\frac{n(n-1)}{2}$ where n is the number of objects we are clustering. In other words, the number of elements, and hence memory usage, grows quadratically and quickly becomes problematic. In `fastcluster` [Müllner, 2013], the hierarchical clustering implementation we use, this data structure is made up of `doubles` which are 64 bits or 8 bytes.

To show why this is a problem, Figure 4.2 shows the memory required to represent the distance matrix for various amounts of objects up to 200 000. Importantly, the distance matrix is required only for clustering and can be freed immediately afterwards. So, if the main phase analysis uses more memory than the distance matrix, saving memory is pointless. Otherwise, the distance matrix drives up the maximum resident set size, which may be im-

practical.

To remedy this, we group objects into independent *regions*, and perform clustering on a per region basis without sacrificing precision. The distance matrix is actually sparse, in that many objects do not share a meaningful relationship, rather they can be allocated distant identifiers without any effect because they never appear in a points-to set together. For example, in the points-to sets $\{o_a, o_b\}$ and $\{o_c, o_d\}$, the objects o_a and o_c do not share any meaningful relation, and no matter how distant their mapped identifiers are, bit-vector representations will not be adversely affected. To region objects, from the points-to sets of the auxiliary analysis, we create an undirected graph where there exists a path between objects o and o' if they occur together in a points-to set, and determine the connected components of the graph[3]. Objects in the same connected component share a relationship, either directly, or transitively, and the difference in the identifiers allocated can be meaningful and so they should be clustered together.

Since objects in distinct regions do not share a relationship, we can build individual distance matrices for objects within the same region and cluster those objects separately from the objects in other regions. Rather than starting our identifier counter from 0, we simply continue the counter from where it left off when assigning identifiers for the objects in the previous clustered region. With this, we can save memory with smaller distance matrices, and time by sequentially clustering regions since clustering grows non-linearly. Furthermore, we can concurrently cluster separate regions, but in our experience, clustering is fast enough for our problem that this is not worth the implementation complexity. If concurrent clustering is foregone, we can maintain a single distance matrix at a time, further saving memory. For example, if n objects are split into two equal sized regions, we would require two distance matrices of $\frac{\frac{n}{2}(\frac{n}{2}-1)}{2}$ distances, and, if not clustered concurrently, at separate times, so only one such distance matrix is required before being freed.

Another benefit of regioning is that certain regions can be assigned identifiers blindly, that is, without any clustering. In regions with fewer than $\mathcal{W}$ objects, all identifier allocations are equivalent since any number of objects $\mathcal{W} - n$, where $n < \mathcal{W}$, all require a single word to represent (modulo some

[3]We are careful to say "exists a path" rather than "exists an edge" as it makes the graph smaller without losing meaning for our purpose. Practically, for each points-to set, an edge can be added from the "first" object in a points-to set to every other object in that points-to set, thus we add add a linear, not quadratic, number of edges per points-to set.

Table 4.2: Region statistics for our benchmark programs.

Program	Objects	Regions[4]	Non-trivial regions	Objects in non-trivial regions (proportion)	Objects in largest region
dhcpcd	2398	3096	1	586 (24.44%)	586
gawk	4007	3084	1	1172 (29.25%)	1172
bash	4294	3247	2	655 (15.25%)	564
mutt	5880	4239	1	2443 (41.55%)	2443
lynx	5829	5283	3	2004 (34.38%)	1754
sqlite	6570	4723	8	2590 (39.42%)	2002
xpdf	12 124	12 904	1	3599 (29.68%)	3599
emacs	17 505	29 407	1	7366 (42.08%)	7366
git	26 481	33 271	2	15 206 (57.42%)	15 076
kakoune	29 413	17 084	22	15 934 (54.17%)	12 603
squid	54 678	34 532	14	23 943 (43.79%)	22 371
wireshark	59 912	72 595	10	6478 (10.81%)	5502

concerns around the crossing of word boundaries as will be discussed in Section 4.2.5). Clustering these objects in such small regions brings no benefit so they can be assigned any identifiers within $\mathcal{W}$ of each other immediately. We call these regions *trivial regions*.

For our benchmark programs, Table 4.2 shows the number of objects, the number of regions, the number of non-trivial regions (those with more than $\mathcal{W} = 64$ objects), the number of objects in non-trivial regions, and the size of the largest region. Without regioning, it would be as though we had a single region with all the objects. We see that trivial regions dominate which greatly reduces the clustering workload. For 5 benchmarks, there exists only 1 non-trivial region and only 3 benchmarks have more than 10 (at most 22, for kakoune). Only 2 benchmarks have more than 50% of objects in non-trivial regions, with most hovering 30% and 40%.

4.2.5 Word-Aligned Identifier Mapping

We can produce better object-to-identifier mappings by considering the regions defined above and ensuring that objects from different regions are never assigned to identifiers within the same word. Consider two regions of objects, R_1 which contains o_a, o_b, and o_c, and R_2 which contains o_d, o_e, and o_f, and

[4]We note that a bug in our implementation produces slightly different results from run to run for this value. We observe no effect on final results and believe it may be a bug in how this value is calculated. We list the largest value.

that we assign identifiers as

$$R_1 : o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2, \text{and}$$
$$R_2 : o_d \mapsto 3 \quad o_e \mapsto 4 \quad o_f \mapsto 5.$$

As a core-bit vector (though this problem applies equally to sparse bit-vectors), all points to sets with objects from R_1 (non-empty subsets of $\{o_a, o_b, o_c\}$) take the form

$$\{\, 0\, [\, \langle \times \times \times 0 \rangle\,]\, \},$$

where each $\times$ can be 1 or 0. This is perfect in that we have achieved the theoretical limit of always representing the points-to sets in R_1 with a single word (modulo metadata such as offsets or links). On the other hand, under this object-to-identifier mapping, points-to sets with objects from R_2 (non-empty subsets of $\{o_d, o_e, o_f\}$) can take one of 3 forms:

$$\begin{array}{rll} \{\, 0\, [\, \langle 0001 \rangle\,]\, \}, & & \{o_d\} \\ \{\, 4\, [\, \langle \times \times 00 \rangle\,]\, \}, & \text{or} & \{o_e\}, \{o_f\}, \{o_e, o_f\} \\ \{\, 0\, [\, \langle 0001 \rangle, \langle \times \times 00 \rangle\,]\, \}. & & \{o_d, o_e\}, \{o_d, o_f\}, \{o_d, o_e, o_f\} \end{array}$$

That is, to represent the non-empty subsets of $\{o_d, o_e, o_f\}$, we would require either 1 or 2 words rather than the theoretical 1 word which is in fact possible. The problem is that regions are sharing words for their objects with other regions despite there never being a points-to set with objects from both regions (by definition). Though R_1 and R_2 are trivial regions, this problem applies to non-trivial regions too.

We can avoid this by aligning identifiers to words when we begin allocating identifiers to each region. Aligning to words for each region would mean that the identifiers for the objects in a region begin at an identifier divisible by $\mathcal{W}$. Returning to our example above, performing word-aligned identifier assignment would result in a mapping akin to

$$R_1 : o_a \mapsto 0 \quad o_b \mapsto 1 \quad o_c \mapsto 2, \text{and}$$
$$R_2 : o_d \mapsto 4 \quad o_e \mapsto 5 \quad o_f \mapsto 6.$$

We have given the same mapping to the objects in R_1 since it was word-aligned, starting at 0, a multiple of $\mathcal{W} = 4$, but have now started the identifiers

in R_2 from 4, another multiple of $\mathcal{W} = 4$. Now we can represent all points-to sets with objects from R_1 as

$$\{\, 0 \,[\, \langle \times \times \times 0 \rangle \,]\,\},$$

as before, and all points-to sets with objects from R_2 as

$$\{\, 4 \,[\, \langle \times \times \times 0 \rangle \,]\,\},$$

thus we represent all such points-to sets with the fewest possible number of words (just 1).

Word-aligning identifier assignment leaves gaps. For example, in the mapping defined in the previous paragraph, no object maps to identifier 3. This is inconsequential since points-to sets only contain objects from the region they are a part of. Concretely, in our example, there is no points-to set containing objects from R_1 (assigned from 0 to 2) that would contain 4 objects in total and thus benefit from that fourth object being assigned the identifier 3. Furthermore, with core bit-vectors (and sparse bit-vectors), any excess leading zero-words caused by word alignment would be inconsequential. Though the clustering is performed on the results of the auxiliary analysis, this holds true for the main phase analysis too since each points-to set produced in the main phase is a subset of a points-to set produced by the auxiliary analysis. In other words, the main phase cannot produce larger regions.

4.3 Evaluation

In our implementation, we use the very capable `fastcluster` [Müllner, 2013] to perform clustering. In theory the remapping process would be to perform clustering, produce an object-to-identifier mapping, and then change the identifiers used for objects everywhere such that the old mapping becomes nonexistent to the analysis. This would be a fast linear scan over the data structures used by SVF, however drastic architectural changes to SVF would be required. Our solution is for objects to have an "internal" identifier in the points-to set and an "external" identifier. External identifiers correspond to the original naive mapping and internal identifiers correspond to the mapping produced from clustering. Objects are then stored in points-to sets as the internal identifier, and returned (e.g., during iteration) as the external identifier.

Table 4.3: Number of words required (using core bit-vectors) in the theoretical best case, with the original mapping, with the mappings produced by clustering using the single, average, and complete linkage criteria, and the improvement versus the original mapping. The bolded value represents the mapping predicted to be best after the auxiliary analysis, and this is what is compared in the final column.

Program	Theoretical[5]	Original	Single	Average	Complete	Reduction vs. original
dhcpcd	409 323	2 905 131	**706 571**	707 786	806 254	4.11×
gawk	13 406 086	78 698 059	**16 373 637**	26 756 858	26 756 208	4.81×
bash	1 239 857	13 781 189	**1 481 237**	2 202 234	1 721 602	9.30×
mutt	3 612 935	38 669 518	**6 145 554**	18 273 265	15 246 309	6.29×
lynx	10 361 701	72 268 279	**17 724 839**	24 154 545	21 431 753	4.08×
sqlite	14 701 928	79 270 630	**24 452 981**	34 243 349	29 327 077	3.24×
xpdf	77 304 422	414 082 955	157 577 371	**154 518 757**	160 730 837	2.68×
emacs	1 146 131 222	3 423 630 147	**1 657 629 802**	1 939 863 478	1 781 467 932	2.07×
git	383 512 116	2 029 028 891	**943 768 707**	1 635 748 917	1 382 608 474	2.15×
kakoune	202 991 907	1 566 374 078	**495 318 140**	1 103 217 759	819 368 200	3.16×
squid	725 679 945	13 068 308 477	**1 671 494 361**	4 278 939 653	6 775 020 227	7.82×
wireshark	42 555 392	1 779 021 608	161 322 197	163 924 857	**148 053 785**	12.02×
Geo. mean						4.41×

In short, only the points-to sets know about the new mapping. This introduces some slight overhead that in an ideal situation would not be incurred. Otherwise, algorithms remain unchanged.

The process of creating and evaluating a mapping is fast. Thus, we produce 3 mappings, each from clustering with a different linkage criterion. We then choose the most promising result (fewest words to represent the points-to sets of the auxiliary analysis) for the mapping used in the main analysis.

4.3.1 Linkage Criteria and Required Words

Table 4.3 shows the number of words required to represent the points-to sets produced by the main analysis in the theoretical (potentially impossible) best case, under the original naive mapping, and under a mapping produced from clustering with single, average, and complete linkage criteria. In bold is the mapping that evaluated as best when applied to the auxiliary analysis's points-to sets and then chosen to perform the main analysis with. The final column shows the reduction in words required by the chosen mapping against the original mapping.

[5]There is an implementation bug occasionally causing slight indeterminism in the average size of the points-to sets of address-taken variables. This is not affecting the final result (top-level variable points-to sets) but causes variation in the number of words required to theoretically represent all points-to sets. Values are rounded arithmetic means of results from 3 runs.

Table 4.4: Time taken (s) and memory usage (GB) for VSFS (using core bit-vectors) without and with a mapping produced by clustering.

Program	Unclustered		Clustered			Speedup	Memory reduction
	Time	Memory	Clustering Time	Total Time	Memory		
dhcpcd	8.08	0.69	0.06	7.70	0.67	1.05×	1.03×
gawk	128.45	4.09	0.23	118.56	3.34	1.08×	1.23×
bash	18.31	1.92	0.09	18.14	1.83	1.01×	1.05×
mutt	40.26	3.49	0.38	40.14	2.87	1.00×	1.21×
lynx	324.07	5.64	0.49	321.98	5.19	1.01×	1.09×
sqlite	202.64	10.38	1.36	203.35	10.07	1.00×	1.03×
xpdf	509.47	15.40	4.75	483.46	10.71	1.05×	1.44×
emacs	4043.31	93.86	43.72	3880.10	72.06	1.04×	1.30×
git	4085.54	50.78	95.20	3938.11	41.78	1.04×	1.22×
kakoune	1592.13	35.07	23.90	1448.03	24.98	1.10×	1.40×
squid	OOM	OOM	53.86	5145.80	89.21	–	≥1.35×
wireshark	2179.94	37.41	4.01	1500.88	21.84	1.45×	1.71×
Geo. mean						1.07×	≥1.24×

Overall, we see an average reduction in required words of 4.41×, in the range of 2.07× to 12.02×. In all cases, the mapping chosen for the main analysis turned out to be the best one. For all but 2 benchmarks (xpdf and wireshark), this was the mapping produced using the single linkage criterion. A likely reason could be that points-to sets for those benchmarks where the single linkage criterion worked best may have clusters of objects that are rarely co-pointee with objects outside their cluster (i.e., if two points-to sets share an object, they likely share most other objects too). Where the complete linkage criterion worked best the opposite would be true. The average linkage criterion would better resolve cases where the situation is mixed. Overall, for most benchmarks, the number of words required with our mapping is around twice as much as the theoretical limit or less. Unfortunately, we cannot know what the true minimum is as it is simply too expensive to apply our IP method.

In the following sections, we discuss time and memory improvements for VSFS, but we note that clients benefit too. For example, an alias analysis may perform many points-to set intersections over now more compact points-to sets, and general iteration over points-to sets would traverse fewer words.

4.3.2 Time

Table 4.4 shows the time (in seconds) VSFS takes without and with the mapping produced by clustering. Generally, we do not see much improvement

except for in wireshark and slight improvement in kakoune. Notably, wireshark saw the greatest reduction in words required, presumably of some key points-to sets too (i.e., those which appear in many union operations). Previous work [Barbar and Sui, 2021a] had seen greater improvement (more akin to that seen in wireshark here) but this was performed on SFS (and counted the time of the main analysis only) which performs far more unions than VSFS. This indicates that this approach becomes increasingly important, time-wise, when analyses grow in the number of operations required.

Column 4 shows the time taken to perform clustering, from creating the distance matrix to creating 3 mappings and evaluating them. Generally, this takes an insignificant portion of total time, except in the case of some larger benchmarks. We find however that the bulk of this time is spent constructing the distance matrix, not in clustering or evaluation of a mapping. For git for example, constructing the distance matrix takes between 85 and 93 seconds. We suspect this can be significantly improved, helping time and making the process truly insignificant to the analysis.

4.3.3 Memory

Memory usage reduction is more successful than time improvement. Table 4.4 also shows memory usage (in GB) of VSFS without and with the mapping produced by clustering. On average we see a reduction of over 1.24×. For some benchmarks, namely dhcpcd, bash, and sqlite, we see no real difference, but others such as wireshark, kakoune, squid, and xpdf considerably improve when considering the number of gigabytes required. squid can now be analysed by VSFS, with room to spare, handling the last of our benchmarks exhausting allocated memory.

4.4 Related Work

Toussi and Khademzadeh [2013] have suggested assigning objects of the same type (or with a subtyping relation) to consecutive identifiers, producing a more sophisticated mapping for Java-based analyses. They also discuss a bit-vector (the ranged bit-vector) which is similar to the core bit-vector we propose but which uses a static over-approximated offset and length in such a way that a pointer of type T only needs to consider type T objects (or subtype of T objects) in its points-to set. While type filtering for strongly-typed lan-

guages such as Java can improve *both* performance and precision [Lhoták and Hendren, 2003], this is not the case for weakly-typed languages such as C and C++. In points-to analysis for C and C++ where objects do not have a set type, typing objects may require heap cloning, and pointers may point to objects of any type [Balatsouras and Smaragdakis, 2016; Barbar et al., 2020]. This incurs (sometimes very significant) overhead and type filtering may involve assumptions that are not suitable for all practitioners [Barbar et al., 2020]. Here, we were able explore object clustering to more *generally* create a better mapping in staged analyses where the auxiliary analysis soundly over-approximates the analysis it is staging.

4.5 Conclusion

In this chapter, we discussed using bit-vectors as points-to sets and reducing their footprint. We first introduced the core bit-vector to strip both leading and trailing zeroes, allowing us to save memory on redundant zero-words. We then looked at producing better object-to-identifier mappings, first through integer programming (a more theoretical exercise), and then more practically through the hierarchical clustering of objects. On average, our approach was able to reduce the number of words required to represent the points-to sets produced by VSFS, implemented as core bit-vectors, by 4.41×.

Hash Consed Points-To Sets 5

When analysing real-world programs, many pointers may yield exactly the same points-to sets during constraint resolution. This sometimes becomes more prevalent as analyses become more precise. For example, unlike flow-insensitive analysis which computes a single points-to set for each pointer, flow-sensitive analysis computes and maintains points-to sets at different program points, but unfortunately introduces many duplicate points-to sets despite the reduction wrought by VSFS.

To see this in practice, Table 5.1 provides the proportions of duplicate points-to sets under flow-insensitive analysis and VSFS. Columns 2 and 5 show the number of points-to sets maintained in the analyses which do not end the analysis as the empty points-to set. Columns 3 and 6 list the number of those points-to sets which refer to the 5 most common points-to sets, again excluding those ending the analysis as the empty set. Columns 4 and 7 list the proportions those top 5 most common points-to sets make up of the aforementioned total. We see that, on average, the 5 most common points-to sets are referred to by around 57% and 93% of pointers for flow-insensitive analysis and VSFS, respectively. Clearly, repeatedly representing the same points-to sets is redundant, memory-wise. We also see that more precise analyses could bring extra redundancy in order to achieve that extra precision.

Furthermore, since many resulting points-to sets are the same, it stands to reason that many may have reached that result through the same union operations, which are also redundant. Thus, many union operations are in

Table 5.1: Statistics on prevalence of duplicate points-to sets in our benchmark programs.

Program	Flow-insensitive			VSFS		
	Points-to sets	Top 5 points-to sets	Proportion	Points-to sets	Top 5 points-to sets	Proportion
dhcpcd	21 422	12 512	58.41%	112 400	103 480	92.06%
gawk	48 539	28 963	59.67%	1 542 123	1 506 481	97.69%
bash	37 365	27 287	73.03%	278 165	256 092	92.06%
mutt	66 034	44 902	68.00%	580 419	544 190	93.76%
lynx	86 873	59 195	68.14%	1 169 126	1 075 077	91.96%
sqlite	146 215	115 799	79.20%	1 325 494	1 263 059	95.29%
xpdf	106 398	53 009	49.82%	3 235 805	3 159 250	97.63%
emacs	249 832	159 156	63.71%	18 020 717	17 673 305	98.07%
git	242 125	125 353	51.77%	10 390 640	9 056 823	87.16%
kakoune	201 445	68 250	33.88%	8 510 969	8 176 369	96.07%
squid	377 684	195 567	51.78%	34 943 134	29 896 669	85.56%
wireshark	334 031	151 587	45.38%	3 036 916	2 743 574	90.34%
Geo. mean			57.18%			93.05%

fact duplicates of operations which have been previously performed. This has implications on performance as conducting points-to set unions produced by the set constraints forms a bulk of analysis time.

Both the number of duplicate points-to sets tracked and the number of unions performed can be reduced but most previous solutions have been analysis-specific requiring algorithmic changes which may not be applicable to other points-to analyses. For example, either, or both, can be achieved by merging equivalent pointers offline [Rountev and Chandra, 2000; Hardekopf and Lin, 2007b, 2011, 2009; Barbar et al., 2021] or online [Pearce et al., 2003; Hardekopf and Lin, 2007a; Lei and Sui, 2019], selectively applying precision [Lhoták and Chung, 2011; Smaragdakis et al., 2011], or carefully choosing how to solve constraints [Pearce et al., 2007; Pereira and Berlin, 2009]. Despite these efforts, duplication still exists and pushing the boundaries through algorithmic changes to the points-to analysis may lead to increasingly diminishing returns on performance.

In this chapter, we aim to explore solutions solely at the data structure level (algorithm-agnostic), ignoring anything that is analysis-specific—thus is easily applicable to a range of points-to analyses—to reduce the influence of these duplicate operations and points-to sets on time and space. We leverage the idea of hash consing [Goubault, 1994; Filliâtre and Conchon, 2006; Braibant et al., 2014; Hash Consing, 2020], which aims to quickly identify structurally equivalent values, from the functional programming community, to help solve the problem of duplicate points-to sets and unions operations.

Hash consing is the process of maintaining single immutable representations of data structures which can then be shared elsewhere referentially [Sondergaard and Sestoft, 1990; Referential Transparency, 2017]. In our context, this means that each unique points-to set is maintained only once such that points-to sets becomes persistent.

Originally, hash consing was used to memoise object construction to avoid creating the same object twice, transforming construction into a hash table lookup of the elements of the object. If we view our union operation as a constructor, taking two points-to sets to create a new one, we can transform many union operations into hash table lookups (of a pair of references), which would be much cheaper than standard set unions as points-to sets become larger. Thus hash consing is a means for efficient memoisation allowing us to perform faster set unions. During points-to set resolution, we build up hash tables of previously performed operations, and use those results if the same operation occurs again.

Moreover, with points-to sets being represented as references we can perform fast comparisons between such sets in constant, instead of linear, time. Thus, we also explore the possibility of practically skipping some set operations completely by exploiting mathematical set properties. For example, since each points-to set is represented as a reference to a sole representation of its value, the operands of a union such as $X \cup Y$ can be compared cheaply for equality. If X and Y are equal, the result is simply X ($/Y$), since the union operation is idempotent, allowing us to skip work. We refer to this and similar optimisations as property operations.

Our approach is efficient yet simple to implement, independent to the points-to analysis used, maintains precision, and works alongside the many algorithmic advances listed earlier. Moreover, our approach does not mandate a specific representation of points-to sets as long as each pointer would otherwise be assigned discrete points-to sets. A large motivation for this chapter is to see how effective a simple technique (hash consing) can be for points-to analyses based on bit-vectors. Most of the work in this chapter has been published at SAS [Barbar and Sui, 2021b].

5.1 Motivating Example

In this section, we show the duplication of points-to sets and operations that occurs in flow-insensitive analysis. Let us consider the small program frag-

ment in Figure 5.1a. Figure 5.1b shows the constraints produced to analyse this program fragment flow-insensitively according to the rules in Figure 2.3. Since the analysis is flow-insensitive, we solve for a points-to set per variable. We use $\{o\}_p$ to denote the value of $pt(p)$ when it, for example, contains the points-to target o. In analysing the program fragment, we assume $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, $pt(r) = \{o_3, o_4\}$, and the remaining points-to sets are empty.

$$
\begin{aligned}
&1 : *p = r; \\
&2 : *q = r; \\
&3 : x = *p; \\
&4 : y = *q;
\end{aligned}
$$

(a) Program fragment.

$$
\begin{aligned}
&1 : \forall o \in pt(p).\ pt(r) \subseteq pt(o) \\
&2 : \forall o \in pt(q).\ pt(r) \subseteq pt(o) \\
&3 : \forall o \in pt(p).\ pt(o) \subseteq pt(x) \\
&4 : \forall o \in pt(q).\ pt(o) \subseteq pt(y)
\end{aligned}
$$

(b) Constraints.

$$
\begin{aligned}
&1 :\ \{o_3, o_4\}_r \subseteq \{\ \}_{o_1} && 1 :\ \{o_3, o_4\}_r \subseteq \{o_3, o_4\}_{o_1} \\
&2 :\ \{o_3, o_4\}_r \subseteq \{\ \}_{o_2} && 2 :\ \{o_3, o_4\}_r \subseteq \{o_3, o_4\}_{o_2} \\
&3 :\ \{o_3, o_4\}_{o_1} \subseteq \{\ \}_x && 3 :\ \{o_3, o_4\}_{o_1} \subseteq \{o_3, o_4\}_x \\
&4 :\ \{o_3, o_4\}_{o_2} \subseteq \{\ \}_y && 4 :\ \{o_3, o_4\}_{o_2} \subseteq \{o_3, o_4\}_y
\end{aligned}
$$

(c) Operations.

$$
\begin{gathered}
pt(p) = \{o_1\} \quad pt(q) = \{o_2\} \quad pt(r) = \{o_3, o_4\} \quad pt(x) = \{o_3, o_4\} \\
pt(y) = \{o_3, o_4\} \quad pt(o_1) = \{o_3, o_4\} \quad pt(o_2) = \{o_3, o_4\} \\
pt(o_3) = \{\ \} \quad pt(o_4) = \{\ \}
\end{gathered}
$$

(d) Result.

Figure 5.1: Example program fragment in (a), constraints generated for flow-insensitive analysis in (b), operations performed to fulfil the constraints in (c), and final results in (d). Initially, we assume $pt(p) = \{o_1\}$, $pt(q) = \{o_2\}$, $pt(r) = \{o_3, o_4\}$, and remaining points-to sets are empty. Duplicate points-to sets and operations are highlighted in grey.

In Figure 5.1c, operations are numbered with the constraints they correspond to and duplicate operations are highlighted in grey. Ultimately, each constraint actually results in the same two union operations being performed so 6 of the 8 operations are duplicates. In real-world programs, such points-to sets may be large, containing hundreds or thousands of objects, meaning

repeatedly performing these unions can be expensive. The final resulting points-to sets of the analysis are shown in Figure 5.1d, with duplicates also highlighted in grey. We see that 5 of the 9 points-to sets have occurred before, pointing to much duplication. This can be problematic as points-to sets grow, with statically sized representations, or as analyses introduce more variables. The problems presented here can become more apparent when analysing more complicated programs or using a more precise analysis.

5.2 Approach

This section introduces hash consed points-to sets and their application to points-to analysis. We then describe optimisations that can use hash consing to efficiently exploit set properties for further performance improvement.

5.2.1 Hash Consed Points-To Sets

Hash consing is used to create immutable data structures that can be shared (referentially) to avoid duplication. A common example of hash consing is string interning [Gosling et al., 2014, §3.10.5] whereby a compiler or runtime stores strings in a global pool and assigns pointers to strings in that global pool rather than private copies. In our context, we want points-to sets to be stored once in a global pool, so that we deal with references to points-to sets rather than concrete points-to sets during the analysis.

To do this, whenever a points-to set is created, we perform an interning routine whereby we check if that points-to set exists in our global pool, and

- If it exists, return a reference to the equivalent set in the global pool.
- Otherwise, add the points-to set to the global pool and return a reference to the newly added points-to set.

This process can be achieved by a single hash table mapping each points-to set to a single canonical reference. Now, instead of using $pt(p)$ during the analysis, we use $pt_r(p)$ which is a reference to the points-to set of p in the global pool. Dereferencing a points-to set reference as $dr(pt_r(p))$ would be equivalent to $pt(p)$ and can be used to, for example, iterate over the points-to set. Given that $pt_r(p) = pt_r(q)$, $dr(pt_r(p))$ and $dr(pt_r(q))$ would also be equivalent and actually be accessing the same singly stored points-to set in

the global pool. This can save significant amounts of memory when duplicate points-to sets are common.

On its own, this process does not save time, and may cost more time to perform the interning routine, especially as we perform many unions creating points-to sets which need to be interned. Since each unique points-to set exists once in the program, we can efficiently memoise operations, including the union operation. This can be achieved by a hash table, which we call an operations table, mapping two points-to set references to the points-to set reference which refers to the result of the actual operation. The union between two points-to set references $pt_r(p) \cup pt_r(q)$ can be performed by looking up the union operations table with the $\langle pt_r(p), pt_r(q) \rangle$ pair as the key (i.e., operation), and

- If the key exists in the operations table, returning the associated value, i.e., the reference to the result of the operation.

- Otherwise, performing a concrete union between the sets $dr(pt_r(p))$ and $dr(pt_r(q))$, interning the result, associating the operation with the result in the operations table, and returning it.

With many union operations being duplicates, those would be performed as constant time hash table lookups, rather than linear time set unions (in a core bit-vector representation, for example) which can be expensive depending on the sizes of points-to sets. The intersection and difference operations can also be memoised the same way, if necessary.

Without hash consing, memoising operations would not be efficient as we would need to hash entire points-to sets, i.e., we would map $\langle pt(p), pt(q) \rangle$ to another concrete points-to set rather than mapping a reference pair to a reference. Collisions would also be expensive to resolve as determining equality would then be linear in the size of the colliding points-to set pairs. With references, equality can be determined in constant time.

Figure 5.2 shows how our analysis would look for the example in Figures 5.1. Of the 8 union operations, 6 are performed as cheap lookups in the operations table in Figure 5.2b because the first time we perform a concrete operation, we cache it in the operations table, and perform a fast lookup on subsequent attempts to perform that operation. As in Figure 5.2c, we associate with every variable a reference to a points-to sets in the global pool

$$\begin{array}{|r|}\hline \{o_1\} \mapsto r_1 \\ \{o_2\} \mapsto r_2 \\ \{o_3, o_4\} \mapsto r_3 \\ \{\,\} \mapsto r_4 \\ \hline \end{array}$$

(a) Global pool mapping points-to sets to references.

$$\begin{array}{|l|}\hline \langle r_3, r_4 \rangle \mapsto r_3 \\ \langle r_3, r_3 \rangle \mapsto r_3 \\ \hline \end{array}$$

(b) Union operations table.

$$pt_r(p) = r_1 \quad pt_r(q) = r_2 \quad pt_r(r) = r_3 \quad pt_r(x) = r_3 \quad pt_r(y) = r_3$$
$$pt_r(o_1) = r_3 \quad pt_r(o_2) = r_3 \quad pt_r(o_3) = r_4 \quad pt_r(o_4) = r_4$$

(c) Result.

FIGURE 5.2: Global pool of points-to sets in (a), the union operations table in (b), and the result in (c) using references instead of concrete points-to sets for the analysis in Figure 5.1.

(Figure 5.2a) rather than concrete points-to sets. In all, we must only store 4 concrete points-to sets.

5.2.2 Exploiting Set Properties

In this section, we describe some optimisations which exploit the properties of sets to further improve efficiency of union operations on hash consed points-to sets. We note that even though our rules only perform unions, practical implementations may perform intersection and difference operations. Furthermore, clients may perform some of these operations too. For example, an alias analysis built on a points-to analysis will perform intersection tests. These operations can be memoised in the same way as unions above, and we exploit their properties in this section too.

Commutative operations For commutative operations such as unions and intersections, performing an operation twice with the operands flipped is duplication. This would not be detected in the operations tables. For example, assuming $pt_r(p) = X$ and $pt_r(q) = Y$, if we perform $X \cup Y = Z$ for the first time, we would store a mapping from the pair $\langle X, Y \rangle$ to the result Z in the union operations table. If the analysis was to later perform $Y \cup X$, it would not find the operation memoised, despite the result also being Z, as $\langle Y, X \rangle$ would not be cached in the union operations table.

To resolve this, operations should always be ordered deterministically. This is easy to achieve with hash consing because points-to sets are references

and can be compared in constant time. Now, to perform $X \cup Y$ or $Y \cup X$, we would perform the operation in the same order depending on whether X is "less than" Y, and so only a single instance would be stored in the union operations table.

Property Operations In some cases, the result of an operation can be determined instantly with only trivial comparisons and no concrete operation or hash table lookup. We refer to these cases as *property operations*, and we describe these cases for unions, intersections, and differences below. We set E to refer to the empty points-to set, and for commutative operations (i.e., unions and intersections), we assume the operands have already been ordered and that the reference E is the least reference (so it is always the first operand in the commutative operations it appears in).

Unions Given the ordered union operation between references X and Y, $X \cup Y$, and that the result would be R,

$$X = E \Rightarrow R = Y, \text{ and}$$
$$X = Y \Rightarrow R = X.$$

All operations in Figure 5.2b are actually property operations and caching is unnecessary.

Intersections Given the ordered intersection operation between references X and Y, $X \cap Y$, and that the result would be R,

$$X = E \Rightarrow R = E, \text{ and}$$
$$X = Y \Rightarrow R = X.$$

Difference Given the difference operation between references X and Y, $X \setminus Y$, and that the result would be R,

$$X = E \Rightarrow R = E,$$
$$Y = E \Rightarrow R = X, \text{ and}$$
$$X = Y \Rightarrow R = E.$$

Preemptive Memoisation After performing an actual operation and caching that operation in the operation table, we can preemptively cache other operations too by exploiting standard set properties. This would avoid performing an actual operation later if the analysis needed that result. An implementation can choose which operations are worth preemptively memoising and which are not.

Unions Assume the ordered operation $X \cup Y = R$ is not a property operation. If $X \neq R$, we can instantly determine and cache

$$X \cup R = R, \text{ and}$$
$$X \cap R = X,$$

and similarly if $Y \neq R$,

$$Y \cup R = R, \text{ and}$$
$$Y \cap R = Y.$$

We guard with the conditions $X \neq R$ and $Y \neq R$ because in these cases, the preemptively cached unions would be property unions.

Intersections Assume the ordered operation $X \cap Y = R$ is not a property operation. If $R \neq E$ and $X \neq R$, we can instantly determine and cache

$$X \cap R = R, \text{ and}$$
$$X \cup R = X,$$

and similarly if $R \neq E$ and $Y \neq R$,

$$Y \cap R = R, \text{ and}$$
$$Y \cup R = Y.$$

Here, we are also not interested in preemptively memoising when $R = E$ because we would be then caching property operations.

Difference Assume the difference operation $X \setminus Y = R$ is not a property operation. If $R \neq E$ and $X \neq R$ we can instantly determine and cache

$$X \cup R = X, \text{ and}$$
$$X \cap R = R,$$

and similarly if $R \neq E$,

$$Y \setminus R = Y,$$
$$R \setminus Y = R, \text{ and}$$
$$R \cap Y = E.$$

5.3 Evaluation

In our implementation, we have not modified any algorithms, rather we have only modified how points-to sets are represented. Concretely, when an analysis attempts to perform a union or access a points-to set, for example, our new code is called. For our hash consed points-to sets, we map concrete points-to sets (core bit-vectors) to unique integer identifiers (which act as our references), and a second map, implemented as an array for performance, mapping those identifiers back to the concrete points-to set. This allows us to use 32-bit identifiers, rather than 64-bit addresses as would be required if our references were pointers.

Our operations tables are implemented as maps (`std::unordered_map` from the C++ STL) mapping two such identifiers to another with a hash function that simply concatenates the two 32-bit identifier operands. In the future, we would like to try a more performant map implementation. Map insertions and lookups are common, for interning and for unions, so we expect that the `std::unordered_map` might be leaving some performance on the table, in both time and memory, as it is known to not perform particularly well due to certain mandates of the C++ standard.

We also note that our implementation does not include garbage collection. Thus, points-to sets which are no longer in use accumulate in the global pool, as well as their operations in the operations tables. This has an effect on memory though we still see improvement in memory usage as shown in the next section.

Table 5.2: Time taken (s) and memory usage (GB) for flow-insensitive points-to analysis without and with hash consed points-to sets.

Program	Mutable		Hash consed		Speedup	Memory reduction
	Time	Memory	Time	Memory		
dhcpcd	2.41	0.46	2.03	0.40	1.18×	1.15×
gawk	15.08	1.33	10.24	1.19	1.47×	1.12×
bash	5.29	0.98	3.00	0.82	1.76×	1.19×
mutt	8.31	1.53	6.77	1.29	1.23×	1.19×
lynx	80.18	2.62	79.10	1.59	1.01×	1.65×
sqlite	78.78	6.57	61.24	5.39	1.29×	1.22×
xpdf	40.18	4.28	32.90	3.67	1.22×	1.17×
emacs	578.17	33.26	477.21	27.33	1.21×	1.22×
git	195.44	15.88	147.35	13.04	1.33×	1.22×
kakoune	86.85	9.22	74.57	6.89	1.16×	1.34×
squid	362.45	26.23	311.45	14.86	1.16×	1.77×
wireshark	90.74	14.53	76.87	9.31	1.18×	1.56×
Geo. mean					1.26×	1.30×

5.3.1 Flow-Insensitive Analysis

Table 5.2 shows the time and memory of a flow-insensitive analysis with and without hash consing. As the auxiliary analysis for VSFS discussed throughout this dissertation, the analysis is boosted by wave propagation [Pereira and Berlin, 2009].

In time, we see an average improvement of 1.26×, with a range from 1.01× (no improvement) to 1.76×. Practically speaking, this analysis already performs quite well so any improvement is certainly welcome. Similarly, we see an average reduction in memory usage of 1.3×, with reduction ranging from 1.12× to 1.77×. Now, no benchmark requires more than 32 GB to analyse (1, previously), 1 benchmark requires more than 16 GB to analyse (2, previously), and 3 benchmarks require more than 8 GB to analyse (5, previously). Again, for memory, this analysis already performed quite well.

Table 5.3 lists the union operations performed by the flow-insensitive analysis and categorises them as concrete (unique) unions, property unions, lookups, and preemptive unions. We see that concrete unions only form about 3% of all unions on average and this proportion does not exceed 10% for any benchmark. This perhaps points to the notion that there is still much room for improvement (we note though that SVF does not implement all optimisations present in the literature such as location and pointer equivalence [Hardekopf

Table 5.3: The number of concrete, property, lookup, and preemptive unions for flow-insensitive analysis (and the proportion of the total in parentheses).

Program	Concrete	Property	Lookup	Preemptive	Total
dhcpcd	5458 (4.39%)	39 981 (32.16%)	69 261 (55.71%)	9632 (7.75%)	124 332
gawk	7475 (2.72%)	138 396 (50.31%)	116 345 (42.29%)	12 886 (4.68%)	275 101
bash	1416 (0.95%)	125 292 (84.53%)	18 896 (12.75%)	2625 (1.77%)	148 228
mutt	7857 (3.13%)	119 688 (47.72%)	109 273 (43.57%)	13 998 (5.58%)	250 815
lynx	25 143 (1.94%)	150 775 (11.64%)	1 077 459 (83.18%)	42 035 (3.24%)	1 295 411
sqlite	8025 (0.81%)	382 278 (38.48%)	588 181 (59.21%)	14 893 (1.50%)	993 377
xpdf	29 714 (4.40%)	196 387 (29.08%)	393 863 (58.32%)	55 425 (8.21%)	675 388
emacs	98 571 (4.84%)	636 598 (31.28%)	1 106 930 (54.39%)	193 041 (9.49%)	2 035 139
git	131 447 (9.12%)	480 171 (33.32%)	610 182 (42.35%)	219 109 (15.21%)	1 440 908
kakoune	56 828 (5.91%)	323 407 (33.61%)	478 367 (49.72%)	103 606 (10.77%)	962 207
squid	108 246 (4.25%)	647 240 (25.44%)	1 623 164 (63.79%)	165 913 (6.52%)	2 544 563
wireshark	49 522 (2.54%)	561 098 (28.81%)	1 245 721 (63.96%)	91 359 (4.69%)	1 947 700
Geo. mean	(3.07%)	(33.75%)	(48.64%)	(5.45%)	

and Lin, 2007b]). On average, property unions make up about 34% of unions and unions which resolve to a lookup make up almost 50% of unions.

5.3.2 VSFS

Table 5.4 shows the time taken and memory used by VSFS with and without hash consing. For time, we see a similar effect to that of flow-insensitive analysis, with an average speedup of about 1.19×, ranging from 1.07× to 1.31×. In terms of practical impact on a human operator, we do see some respectable improvement. For example, on the slower end, analysing emacs takes about 12 minutes fewer, down from about 65 to 53 minutes and on the faster end, analysing gawk goes from about 120 to 90 seconds.

Memory tells a similar but slightly more effective story with an average reduction of 1.28× ranging from 1.11× to 1.47×. Only the first 4 benchmarks in the table and sqlite see a reduction in memory usage of less than 1.3×. We see very practical improvement with, for example, emacs requiring 20 fewer gigabytes, squid requiring around 30 fewer gigabytes, and wireshark requiring about 5 and a half fewer gigabytes, having required 21.84 GB previously. All benchmarks can now be analysed with VSFS within 64 GB, squarely in the realm of consumer hardware.

We also note that this improvement has come despite the application of our techniques which reduce the number of points-to sets and unions (VSFS) and compact points-to set representations (core bit-vectors, compacting through clustering). In a sense, this is an optimisation which can squeeze a few extra seconds or gigabytes out of an implementation, catching some

Table 5.4: Time taken (s) and memory usage (GB) for VSFS without and with hash consed points-to sets.

Program	Mutable		Hash consed		Speedup	Memory reduction
	Time	Memory	Time	Memory		
dhcpcd	7.70	0.67	6.70	0.59	1.15×	1.13×
gawk	118.56	3.34	90.52	2.90	1.31×	1.15×
bash	18.14	1.83	15.17	1.65	1.20×	1.11×
mutt	40.14	2.87	34.42	2.53	1.17×	1.13×
lynx	321.98	5.19	302.31	3.92	1.07×	1.32×
sqlite	203.35	10.07	174.51	8.39	1.17×	1.20×
xpdf	483.46	10.71	399.95	8.17	1.21×	1.31×
emacs	3880.10	72.06	3155.43	49.90	1.23×	1.44×
git	3938.11	41.78	3339.26	30.46	1.18×	1.37×
kakoune	1448.03	24.98	1123.69	17.61	1.29×	1.42×
squid	5145.80	89.21	4611.79	60.80	1.12×	1.47×
wireshark	1500.88	21.84	1276.60	16.12	1.18×	1.35×
Geo. mean					1.19×	1.28×

Table 5.5: The number of concrete, property, lookup, and preemptive unions for VSFS (and the proportion of the total in parentheses).

Program	Concrete	Property	Lookup	Preemptive	Total
dhcpcd	11 968 (0.07%)	14 979 551 (81.90%)	3 283 393 (17.95%)	15 416 (0.08%)	18 290 327
gawk	129 480 (0.05%)	204 024 076 (80.11%)	50 358 240 (19.77%)	167 781 (0.07%)	254 679 576
bash	3547 (0.02%)	11 349 430 (78.61%)	3 080 211 (21.33%)	4996 (0.03%)	14 438 183
mutt	22 727 (0.03%)	61 751 625 (87.61%)	8 678 406 (12.31%)	30 010 (0.04%)	70 482 767
lynx	63 684 (0.01%)	622 937 164 (86.30%)	98 748 841 (13.68%)	79 537 (0.01%)	721 829 226
sqlite	36 124 (0.02%)	186 364 451 (81.77%)	41 463 179 (18.19%)	46 920 (0.02%)	227 910 673
xpdf	100 035 (0.01%)	886 088 543 (79.97%)	221 722 933 (20.01%)	123 359 (0.01%)	1 108 034 870
emacs	484 230 (0.01%)	6 626 504 811 (85.49%)	1 123 816 883 (14.50%)	657 167 (0.01%)	7 751 463 091
git	515 285 (0.01%)	5 968 894 934 (84.15%)	1 123 045 537 (15.83%)	630 943 (0.01%)	7 093 086 697
kakoune	449 844 (0.02%)	2 253 521 396 (76.21%)	702 597 617 (23.76%)	568 914 (0.02%)	2 957 137 770
squid	502 938 (0.01%)	4 117 489 437 (77.96%)	1 163 177 210 (22.02%)	600 902 (0.01%)	5 281 770 486
wireshark	7 154 852 (0.17%)	2 743 586 645 (65.82%)	1 409 445 126 (33.81%)	8 316 285 (0.20%)	4 168 502 907
Geo. mean	(0.02%)	(80.29%)	(18.75%)	(0.03%)	

redundancy other techniques have missed. When concrete unions are more expensive to operate on (more objects through heap cloning, for example) or points-to sets are more numerous, we expect this technique to be more necessary, at least until the sources of these costs are found and (in a more algorithmic way) quashed.

Table 5.5, similar to the previous section, breaks down the number of union operations performed by VSFS. Aside from the fact that VSFS performs far more unions, results differ significantly from those of the flow-insensitive analysis. We see that no benchmark, except wireshark (0.17%), requires more than 0.1% of unions to be concrete unions (0.02% on average) and that property

operations make up a larger proportion compared to flow-insensitive analysis at about 80% on average. Unions which resolve to a lookup make up about 19% of all unions on average.

5.3.3 Effect of Preemptive Memoisation

In previous work [Barbar and Sui, 2021b] we have found that preemptive memoisation does not have a discernible effect on time. This is because preemptive memoisation reduces the number of concrete unions *after* the application of our techniques in this chapter (i.e., after other techniques have made the most expensive operations cheaper, e.g., transforming n particularly expensive unions into one concrete union followed by $n - 1$ lookups). As it stands, concrete unions take up a small portion all unions even without preemptive memoisation. As input programs grow and concrete unions start to have a noticeable effect on time (e.g., when points-to sets become unreasonably large or when more points-to sets and operations are unique), the role preemptive memoisation plays can become more significant. As expected, we saw a slight increase in memory usage due to storing more operations in the operations table where each entry takes 12 bytes, modulo any table overhead.

5.4 Related Work

In unpublished work, Heintze [1999] described splitting points-to sets into two parts: a unique part (called an overflow list) and a shared part. The shared part can be described as hash consing and thus implements a finer-grained hash consing since it does this on subsets rather than entire sets. However, no memoisation is performed, and doing so would be less effective due to the overflow list where, for example, two sets may be equivalent but not share any parts (i.e. the unique parts are different and the shared parts are different). The data structure is also much more difficult to implement whereas what we have presented can be retrofitted onto most set-like data structures exposing necessary operations (largely the set union operation). An implementation of Heintze's set is available in Soot [Lhoták and Hendren, 2003] as the `SharedHybridSet`.

Using (reduced ordered) BDDs to represent points-to sets naturally performs sharing of data. BDDs in the context of points-to analysis are acyclic graphs representing a set of points-to relations, and nodes can be merged

or removed according to certain conditions which are analogous to shared points-to set subsets. However, BDD performance (in time and space) relies heavily on a "variable ordering" and finding an ideal variable ordering is impractical (in the same way our IP approach in Chapter 4 is impractical), though approximations exist. BDDs in the literature have been found to be an overall better [Hardekopf and Lin, 2009; Berndl et al., 2003] and worse [Bravenboer and Smaragdakis, 2009; Hardekopf and Lin, 2007b] choice than more explicit set data structures (such as bit-vectors). This is likely due to varying tweaking of parameters like variable ordering, quality of the analysis at hand (various optimisations may affect how well different data structures perform at different magnitudes), and due to implementation details close to the machine. Generally, BDDS are more complex than basic sets (like bit-vectors) in regards to implementation, and we experienced minimal implementation burden adding the approach presented in this chapter to SVF's existing facilities.

Hash consing has also more generally been explored for static analysis to represent, for example, memory maps and program states [Manevich et al., 2002; Cuoq et al., 2012], invocation graphs [Choi and Choe, 2011], subtrees [Ball and Rajamani, 2001], and constants [Hubert et al., 2011] with success. Static analyses are ripe for hash consing and memoisation because they are by nature approximations designed to capture a *class* of runtime data and so contain many duplicate data structures, operations, or both.

5.5 Conclusion

In this chapter, we discussed hash consed points-to sets to produce a persistent data structure which can be used to reduce duplication in points-to sets, saving space, and in operations, saving time. During points-to analysis resolution, unique points-to sets are represented once and are referred to through cheaper references. Through memoisation and cheap operand comparisons on the references, the effect of duplicate operations is reduced. For flow-insensitive and -sensitive analyses, we saw an average speedup of 1.26× and 1.19× and an average reduction in memory usage of 1.3× and 1.28×, respectively.

Conclusion 6

6.1 Applying the Techniques in this Dissertation Together

In Table 6.1, we have compared the time taken and memory required for SFS with standard bit-vectors as points-to sets (from Table 3.1) against VSFS with all the optimisations presented during this dissertation (from Table 5.4), that is, VSFS with efficient versioning on 4 threads, core bit-vectors, an object-to-identifier mapping produced through clustering, and hash consed points-to sets. Overall, we see an average speedup of 5.92×, ranging from 3.17× to 8.58×. We are missing many speedup values due to memory being exhausted analysing 5 benchmarks with SFS. For memory usage reduction, we see an average of ≥3.97×, ranging from 1.79× to 10.18×. Similar to what was discussed about the true memory usage reduction in Section 3.6.2, we cannot know exactly how much larger than 3.97× the true average is (or if it is exactly 3.97×). Especially, consider benchmarks emacs and squid which use the most memory under our approach. With our approach, they use around 50 and 60 GB of memory (still within the bounds of consumer hardware), respectively, so we would expect their memory usage for SFS to be far more than 120 GB. In either case, we are pleased with the overall result.

There always exists a certain conflict and harmony in combining improvements to a single analysis. All of our techniques can work together; none *completely* subsume any other, especially as analyses and input programs grow with time. However, they do overlap, and thus, in a way, conflict. For ex-

Table 6.1: Time taken (s) and memory usage (GB) for SFS (with standard bit-vectors) and all of our techniques combined: VSFS, efficient versioning (4 threads), core bit-vectors, an object-to-identifier mapping produced through clustering, and hash consed points-to sets.

Program	SFS		Our approach		Speedup	Memory reduction
	Time	Memory	Time	Memory		
dhcpcd	21.21	1.06	6.70	0.59	3.17×	1.79×
gawk	533.41	10.31	90.52	2.90	5.89×	3.56×
bash	123.72	5.60	15.17	1.65	8.15×	3.39×
mutt	268.19	12.34	34.42	2.53	7.79×	4.88×
lynx	1692.75	24.50	302.31	3.92	5.60×	6.25×
sqlite	779.47	20.10	174.51	8.39	4.47×	2.39×
xpdf	3429.98	83.20	399.95	8.17	8.58×	10.18×
emacs	OOM	OOM	3155.43	49.90	–	≥2.41×
git	OOM	OOM	3339.26	30.46	–	≥3.94×
kakoune	OOM	OOM	1123.69	17.61	–	≥6.82×
squid	OOM	OOM	4611.79	60.80	–	≥1.97×
wireshark	OOM	OOM	1276.60	16.12	–	≥7.44×
Geo. mean					5.92×	≥3.97×

ample, a good object-to-identifier mapping becomes far less significant in the presence of hash consing (especially) or even VSFS because the number of concrete unions and the number of actual points-to sets stored is significantly reduced. This can be seen by the superior results obtained in our previous work [Barbar and Sui, 2021a] where we evaluated using improved object-to-identifier mappings on SFS (rather than VSFS). Another example is the impact of hash consing. In previous work [Barbar and Sui, 2021b], we evaluated hash consing using LLVM's sparse bit-vector data structure for points-to sets (often more expensive than a bit-vector or core bit-vector assuming a mapping not too unfavourable) and SFS (for the flow-sensitive analysis) and we saw far more drastic improvement than we saw when we used core bit-vectors with a clustered mapping and VSFS as in this dissertation.

That said, where one technique falls short, another can make up for it, bringing our techniques together in harmony. As an example, since the pre-analysis in VSFS is based on auxiliary information, there remain many redundant points-to sets and propagations. Although hash consing cannot eliminate this redundancy, it can reduce it by storing points-to sets once to be referred to many times and by caching unions, almost eliminating some by short-circuiting with a property union. Ultimately however, for practition-

ers, the important part is that we see an overall improvement, regardless of how much or how little each technique has contributed. Thus, practitioners interested in applying these techniques (rather than implementing them) do not need to make a choice between them—they all contribute, no matter how little or much—and the described conflict is of scientific interest moreso than pragmatic interest.

6.2 Summary

In all, we have discussed a variety of techniques to improve the data structures behind points-to analysis and subsequently the analysis itself. We introduced *versioned staged flow-sensitive analysis* (VSFS), an improvement over *staged flow-sensitive analysis* (SFS), utilising meld versioning to effectively merge points-to sets and reduce the number of required unions. We had done away with the SVFG for the analysis, sufficing with version reliances. We also described how meld versioning can be parallelised, bring simple parallelisation to a non-trivial aspect of the analysis.

More generally, we looked at points-to sets and how they could be improved. First, we described the core bit-vector to better represent points-to sets since it strips both leading and trailing zero-words. We then worked on improving the required object-to-identifier mapping for more compact points-to sets. To do this, we exploited the fact that the auxiliary analysis in SFS/VSFS is a sound over-approximation of the main phase. We started with an integer programming solution which would be optimal (for the auxiliary analysis) but too expensive for real world programs. We then looked at using hierarchical clustering, which though more approximate, was able to be used to produce a good mapping. Finally, we ended with the application of hash consing and memoisation to points-to sets, which additionally allowed us to perform property operations.

We implemented our techniques in LLVM-based open source points-to analysis framework SVF. Overall, in comparing our techniques to SVF's implementation of SFS on 12 open source programs, we saw an average 5.92× speedup and an average reduction in memory usage of ≥3.97×. Originally, we were unable to analyse 5 of our benchmarks within our memory limit of 120 GB, but could do so with good performance after the application of our techniques.

6.3 Future Work

We believe there is still much which can be improved for points-to analysis data structures. It also appears that flow-sensitive analysis could eventually become efficient enough to act as a baseline of sorts (the space which flow-insensitive analysis currently occupies).

Though encoding less redundancy than SFS, VSFS still contains much. It would be worth exploring refining versions on-the-fly during the main phase, where flow-sensitive points-to information is available. It is also possible that further refinements can be made offline.

We believe that there is room for non-random or non-naive object-to-identifier mappings that do not rely on an auxiliary analysis nor on types. This could improve the auxiliary analysis itself before more information is made available for a mapping from it. It would be interesting to train a machine learning model on a corpus of programs to find patterns in them indicating which objects would likely appear as co-pointees.

Our hash consing implementation does not include garbage collection, allowing unused points-to sets and operations to accumulate in the global pool and the operations tables. Tailoring garbage collection specifically for points-to analysis appears to be beneficial.

Bibliography

Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. University of Copenhagen, Denmark.

George Balatsouras and Yannis Smaragdakis. 2016. Structure-Sensitive Points-To Analysis for C and C++. In *International Static Analysis Symposium (SAS '16)*. Springer, Germany, 84–104. https://doi.org/10.1007/978-3-662-53413-7_5

Thomas Ball and Sriram K. Rajamani. 2001. Bebop: A Path-Sensitive Interprocedural Dataflow Engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*. ACM, USA, 97–103. https://doi.org/10.1145/379605.379690

Mohamad Barbar and Yulei Sui. 2021a. Compacting Points-to Sets through Object Clustering. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 159 (Oct. 2021), 27 pages. https://doi.org/10.1145/3485547

Mohamad Barbar and Yulei Sui. 2021b. Hash Consed Points-To Sets. In *International Static Analysis Symposium (SAS '21)*. Springer, Germany, 25–48. https://doi.org/10.1007/978-3-030-88806-0_2

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2020. Flow-Sensitive Type-Based Heap Cloning. In *34th European Conference on Object-Oriented Programming (ECOOP '18, Vol. 166)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 24:1–24:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.24

Mohamad Barbar, Yulei Sui, and Shiping Chen. 2021. Object Versioning for Flow-Sensitive Pointer Analysis. In *2021 IEEE/ACM International Sym-*

posium on Code Generation and Optimization (CGO '21). IEEE Computer Society, USA, 222–235. https://doi.org/10.1109/CGO51591.2021.9370334

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. 2003. Points-to Analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. ACM, USA, 103–114. https://doi.org/10.1145/781131.781144

Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. 2014. Implementing and reasoning about hash-consed data structures in Coq. *Journal of Automated Reasoning* 53, 3 (2014), 271–304. https://doi.org/10.1007/s10817-014-9306-0

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, USA, 243–262. https://doi.org/10.1145/1640089.1640108

Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, USA, 39–50. https://doi.org/10.1145/1455770.1455778

Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, USA, 2325–2342.

Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities using Deep Graph Neural Network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33. https://doi.org/10.1145/3436877

Woongsik Choi and Kwang-Moo Choe. 2011. Cycle elimination for invocation graph-based context-sensitive pointer analysis. *Information and Software Technology* 53, 8 (2011), 818–833. https://doi.org/10.1016/j.infsof.2011.03.003

Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. 1996. Effective Representation of Aliases and Indirect Memory Operations in SSA Form. In *Proceedings of the 6th International Conference on Compiler Construction (CC '96)*. Springer, Germany, 253–267. https://doi.org/10.1007/3-540-61053-7_66

crux-bitcode 2021. crux-bitcode. https://github.com/mbarbar/crux-bitcode. Last accessed on 14 January 2022.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM '12)*. Springer, Germany, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

CVE 2014. CVE-2014-6271. Available from the NIST's NVD, CVE-ID CVE-2014-6721.. https://nvd.nist.gov/vuln/detail/CVE-2014-6271 Last accessed on 15 January 2022.

Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '17)*. ACM, USA, 329–340. https://doi.org/10.1145/3092703.3092729

Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-Based Control Flow Integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, USA, 28–39. https://doi.org/10.1145/3274694.3274739

Jean-Christophe Filliâtre and Sylvain Conchon. 2006. Type-Safe Modular Hash-Consing. In *Proceedings of the 2006 Workshop on ML (ML '06)*. ACM, USA, 12–19. https://doi.org/10.1145/1159876.1159880

Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Transactions on Software Engineering Methodology* 17, 2, Article 9 (May 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

Rakesh Ghiya, Daniel Lavery, and David Sehr. 2001. On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, USA, 47–58. https://doi.org/10.1145/378795.378806

James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional, USA.

Jean Goubault. 1994. Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing. *Journées Francophones des Langages Applicatifs* (1994), 222–238.

Samuel Z. Guyer and Calvin Lin. 2005. Error Checking with Client-Driven Pointer Analysis. *Science of Computer Programming* 58, 1–2 (oct 2005), 83–114. https://doi.org/10.1016/j.scico.2005.02.005

Ben Hardekopf and Calvin Lin. 2007a. The Ant and the Grasshopper: Fast and Accurate Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, USA, 290–299. https://doi.org/10.1145/1250734.1250767

Ben Hardekopf and Calvin Lin. 2007b. Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis. In *International Static Analysis Symposium (SAS '07)*. Springer, Germany, 265–280. https://doi.org/10.1007/978-3-540-74061-2_17

Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, USA, 226–238. https://doi.org/10.1145/1480881.1480911

Ben Hardekopf and Calvin Lin. 2011. 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)*. IEEE Computer Society, USA, 289–298. https://doi.org/10.1109/CGO.2011.5764696

Hash Consing 2020. Hash consing. https://en.wikipedia.org/wiki/Hash_consing.

Nevin Heintze. 1999. Analysis of Large Code Bases: The Compile-Link-Analyze Model. (1999). http://web.archive.org/web/20050513012825/http://cm.bell-labs.com/cm/cs/who/nch/cla.ps (unpublished).

Michael Hind and Anthony Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *International Static Analysis Symposium (SAS '98, Vol. 1503)*. Springer, Germany, 57–81. https://doi.org/10.1007/3-540-49727-7_4

Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast Online Pointer Analysis. *ACM Transactions on Programming Languages and Systems* 29, 2 (apr 2007), 11–es. https://doi.org/10.1145/1216374.1216379

Laurent Hubert, Nicolas Barré, Frédéric Besson, Delphine Demange, Thomas Jensen, Vincent Monfort, David Pichardie, and Tiphaine Turpin. 2011. Sawja: Static Analysis Workshop for Java. In *Formal Verification of Object-Oriented Software*. Springer, Germany, 92–106. https://doi.org/10.1007/978-3-642-18070-5_7

ISO/IEC. 2017. *ISO/IEC 14882:2017 — Programming languages — C++* (fifth ed.). International Organization for Standardization, Switzerland. 1605 pages.

Paul Jaccard. 1912. The distribution of the flora in the Alpine zone. *New Phytologist* 11, 2 (1912), 37–50. https://doi.org/10.1111/j.1469-8137.1912.tb05611.x

Chris Lattner and Vikram Adve. 2004. 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)*. IEEE Computer Society, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

Anatole Le, Ondřej Lhoták, and Laurie Hendren. 2005. Using Inter-Procedural Side-Effect Information in JIT Optimizations. In *Proceedings of the 14th International Conference on Compiler Construction (CC '05)*. Springer, Germany, 287–304. https://doi.org/10.1007/11406921_22

Yuxiang Lei and Yulei Sui. 2019. Fast and Precise Handling of Positive Weight Cycles for Field-Sensitive Pointer Analysis. In *International Static Analysis Symposium (SAS '19)*. Springer, Germany, 27–47. https://doi.org/10.1007/978-3-030-32304-2_3

Ondrej Lhoták and Kwok-Chiang Andrew Chung. 2011. Points-to Analysis with Efficient Strong Updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, USA, 3–16. https://doi.org/10.1145/1926385.1926389

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC '03)*. Springer, Germany, 153–169.

Ondřej Lhoták and Laurie Hendren. 2006. Context-Sensitive Points-to Analysis: Is It Worth It?. In *Proceedings of the 15th International Conference on Compiler Construction (CC '06)*. Springer, Germany, 47–64. https://doi.org/10.1007/11688839_5

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (2015), 44–46. https://doi.org/10.1145/2644805

V. Benjamin Livshits and Monica S. Lam. 2003. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, USA, 317–326. https://doi.org/10.1145/940071.940114

LLVM BitVector 2021. https://llvm.org/doxygen/BitVector_8h_source.html. Last accessed on 16 April 2021.

Oded Maimon and Lior Rokach. 2005. *Data Mining and Knowledge Discovery Handbook* (1st ed.). Springer, USA. https://doi.org/10.1007/b107408

Roman Manevich, G. Ramalingam, John Field, Deepak Goyal, and Mooly Sagiv. 2002. Compactly Representing First-Order Structures for Static Analysis. In *Proceedings of the 9th International Symposium on Static Analysis (SAS '02)*. Springer, Germany, 196–212. https://doi.org/10.5555/647171.716101

Fionn Murtagh. 1983. A Survey of Recent Advances in Hierarchical Clustering Algorithms. *Comput. J.* 26, 4 (11 1983), 354–359. https://doi.org/10.1093/comjnl/26.4.354

Daniel Müllner. 2013. fastcluster: Fast Hierarchical, Agglomerative Clustering Routines for R and Python. *Journal of Statistical Software, Articles* 53, 9 (2013), 1–18. https://doi.org/10.18637/jss.v053.i09

Esko Nuutila and Eljas Soisalon-Soininen. 1994. On Finding the Strongly Connected Components in a Directed Graph. *Inform. Process. Lett.* 49, 1 (1994), 9–14. https://doi.org/10.1016/0020-0190(94)90047-7

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, USA, 475–484. https://doi.org/10.1145/2594291.2594318

David J. Pearce. 2016. A space-efficient algorithm for finding strongly connected components. *Inform. Process. Lett.* 116, 1 (2016), 47–52. https://doi.org/10.1016/j.ipl.2015.08.010

David J Pearce, Paul HJ Kelly, and Chris Hankin. 2003. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '03)*. IEEE Computer Society, USA, 3–12. https://doi.org/10.1109/SCAM.2003.1238026

David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Transactions on Programming Languages and Systems* 30, 1 (Nov. 2007), 4:1–4:42. https://doi.org/10.1145/1290520.1290524

Fernando Magno Quintao Pereira and Daniel Berlin. 2009. 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). IEEE Computer Society, USA, 126–135. https://doi.org/10.1109/CGO.2009.9

Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, USA, 320–331. https://doi.org/10.1145/1133981.1134019

Referential Transparency 2017. What is Referential Transparency? https://www.sitepoint.com/what-is-referential-transparency.

Atanas Rountev and Satish Chandra. 2000. 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)*. ACM, USA, 47––56. https://doi.org/10.1145/349299.349310

Simplified-STL 2022. Simplified-STL. https://github.com/SVF-tools/Simplified-STL. Last accessed on 29 September 2022.

Yannis Smaragdakis, George Balatsouras, and George Kastrinis. 2013. 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)*. ACM, USA, 253––270. https://doi.org/10.1145/2509136.2509524

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, USA, 17–30.

Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming (ECOOP '18, Vol. 109)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Germany, 23:1–23:28. https://doi.org/10.4230/LIPIcs.ECOOP.2018.23

Harald Sondergaard and Peter Sestoft. 1990. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica* 27, 6 (Jan. 1990), 505–517.

Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, USA, 32–41. https://doi.org/10.1145/237721.237727

Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2Vec: Value-Flow-Based Precise Code Embedding. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27. https://doi.org/10.1145/3428301

Yulei Sui and Jingling Xue. 2016a. On-Demand Strong Update Analysis via Value-Flow Refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16)*. ACM, USA, 460–473. https://doi.org/10.1145/2950290.2950296

Yulei Sui and Jingling Xue. 2016b. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC '16)*. ACM, USA, 265–266. https://doi.org/10.1145/2892208.2892235

Yulei Sui, Hua Yan, Zheng Zheng, Yunpeng Zhang, and Jingling Xue. 2018. Parallel Construction of Interprocedural Memory SSA Form. *Journal of Systems and Software* 146 (2018), 186–195. https://doi.org/10.1016/j.jss.2018.09.038

Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing* 1, 2 (1972), 146–160. https://doi.org/10.1137/0201010

Hamid A Toussi and Ahmed Khademzadeh. 2013. Improving Bit-Vector Representation of Points-To Sets Using Class Hierarchy. *International Journal of Computer Theory and Engineering* 5, 3 (2013), 494–499. https://doi.org/10.7763/IJCTE.2013.V5.736

David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-Sensitive Pointer Analysis for Symbolic Execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. ACM, USA, 197–208. https://doi.org/10.1145/3368089.3409698

David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, USA, 350–360. https://doi.org/10.1145/3180155.3180251

WALA 2021. The T. J. Watson Libraries for Analysis (WALA). http://wala.sf.net/.

Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. ACM, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI '95)*. ACM, USA, 1–12. https://doi.org/10.1145/207110.207111

WLLVM 2021. Whole Program LLVM in Go. https://github.com/SRI-CSL/gllvm. Last accessed on 14 September 2021.

Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. 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)*. ACM, USA, 327–337. https://doi.org/10.1145/3180155.3180178

Suan Hsi Yong, Susan Horwitz, and Thomas Reps. 1999. Pointer Analysis for Programs with Structures and Casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*. ACM, USA, 91–103. https://doi.org/10.1145/301618.301647

Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. 2018. Parallel Sparse Flow-Sensitive Points-to Analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC '18)*. ACM, USA, 59–70. https://doi.org/10.1145/3178372.3179517