Adaptive Constraint Solving for Information Flow Analysis

by

Santanu Kumar Dash

submitted to the University of Hertfordshire
in partial fulfilment of the requirements of the degree of
Doctor of Philosophy (PhD)

Submitted: November, 2014

Abstract

In program analysis, unknown properties for terms are typically represented symbolically as variables. Bound constraints on these variables can then specify multiple optimisation goals for computer programs and find application in areas such as type theory, security, alias analysis and resource reasoning. Resolution of bound constraints is a problem steeped in graph theory; interdependencies between the variables is represented as a constraint graph. Additionally, constants are introduced into the system as concrete bounds over these variables and constants themselves are ordered over a lattice which is, once again, represented as a graph. Despite graph algorithms being central to bound constraint solving, most approaches to program optimisation that use bound constraint solving have treated their graph theoretic foundations as a black box. Little has been done to investigate the computational costs or design efficient graph algorithms for constraint resolution. Emerging examples of these lattices and bound constraint graphs, particularly from the domain of language-based security, are showing that these graphs and lattices are structurally diverse and could be arbitrarily large. Therefore, there is a pressing need to investigate the graph theoretic foundations of bound constraint solving.

In this thesis, we investigate the computational costs of bound constraint solving from a graph theoretic perspective for Information Flow Analysis (IFA); IFA is a sub-field of language-based security which verifies whether confidentiality and integrity of classified information is preserved as it is manipulated by a program. We present a novel framework based on graph decomposition for solving the (atomic) bound constraint problem for IFA. Our approach enables us to abstract away from connections between individual vertices to those between sets of vertices in both the constraint graph and an accompanying security lattice which defines ordering over constants. Thereby, we are able to achieve significant speedups compared to state-of-the-art graph algorithms applied to bound constraint solving. More importantly, our algorithms are highly adaptive in nature and seamlessly adapt to the structure of the constraint graph and the lattice. The computational costs of our approach is a function of the latent scope of decomposition in the constraint graph and the lattice; therefore, we enjoy the fastest runtime for every point in the structure-spectrum of these graphs and lattices. While the techniques in this dissertation are developed with IFA in mind, they can be extended to other application of the bound constraints problem, such as type inference and program analysis frameworks which use annotated type systems, where constants are ordered over a lattice.

Acknowledgements

I would like to express my deep gratitude to my supervisor Professor Bruce Christianson for his unwavering support and excellent guidance during the course of this PhD. Bruce's wealth of research experience and technical knowledge helped crystallise my thoughts on how to conduct and present high quality research. Despite his busy schedule, he always made it a point to read every word in any document I wrote up and always gave me detailed feedback. On a personal note, I was deeply touched by the compassion, consideration and generousity he showed towards me during phases in my PhD when I felt lost. I feel really fortunate to have worked with him and have learnt a lot of things from him.

I would like to thank Professor Sven-Bodo Scholz who initiated me into Compiler Technology and Computer Architecture (CTCA) group. He was always very generous with his time whenever I approached him and patiently discussed research directions with me. I am deeply indebted to Professor Alex Shafarenko who hosted me in the CTCA group and always supported me in my research goals.

I had an excellent time in Science and Technology Research Institute (STRI) where I spent five years and made many great friends. I am grateful to my friends at STRI - Nilesh, Htoo (Phothar), Chaminda - for the scintillating lunch conversations we had. Thank you all for the good times we shared together. Thank you Lorraine, Michaella and Avis for cheerfully dealing with every piece of my paperwork.

My biggest source of strength throughout this endeavour has been my wife, Meethu. She has been my motivator-in-chief and the only reason I could embark on a PhD was because she ensured everything else outside my PhD was taken care of. She has been my rainbow on rainy days and my sunshine on sunny days. Thank you Meethu, I don't think I could have ever completed a PhD if you had not been there in my life.

I would like to thank my wife, my brother Ronny and my mother Rina for providing me with superhuman support as I was shuttling between continents to finish my studies and attend to my ailing father. This dissertation would not have seen light of day without their support. A special vote of thanks also goes to my parents-in-law who helped me out during the thesis-writing phase and always believed in my potential as a researcher.

My father went through great hardships to raise us and taught us the values of education and knowledge. Sadly, we lost him just during the course of my PhD. Seeing a Dr. title prepended to my name would give him immense joy. The values of simplicity, hard work, humility and honesty that he instilled in me will stay with me forever. I miss you Baba, always will. I promise to make this world a better place for others like you did for us. You are and forever will be my role-model.

To Baba, thank you for everything.

Contents

List of Figures

Chapter 1

Introduction

Flow analysis of computer programs is an essential step in driving many qualitative optimisations. It determines how values may propagate from one point in the program to another and whether such a flow is desirable or not from the perspective of the optimisation. There are two approaches to understanding the nature of flow in programs: static and dynamic. Static analysis inspects flow properties at compile time and reports properties for program flow without running the application. On the other hand, dynamic analysis involves observing the runtime trace of the application in order to identify properties for program flow. Consequently, the dynamic analysis is limited in scope and depends on the paths taken during the execution of the program. In contrast to this, existing techniques for static analysis consider all possible execution paths for an application while trying to understand the nature of flow through them. Therefore, in critical applications which demand an in-depth analysis, static analysis is often the preferred option.

1.1 Lattice-directed information flow

One such application is application security. Consider an Android^{TM} app that is freely available on the Google play store. A malicious developer can repackage well-known applications and distribute them through the store. Typically this application exploits permissions granted by the user to the application for using the platform to transmit sensitive information. For example, the permission granted by the user to the app for using the Internet could be misused. If the app is malicious, it may access and transmit personal information such as photos or videos of the user and transmit it over the Internet to a third party. The goal of application security is to prevent flow of sensitive information to public sinks and flow analysis is central to achieving this. Therefore, it is necessary to consider all possible flows in an application between the points at which sensitive permission is granted to the points at which it is transmitted in order to enforce application-level security.

Sometimes, it may be useful to specify a partial order on what can flow where. Lattice-based security models for computer programs are an instance of such an ordering. In such models, an ordering over discrete security labels describes permitted declassification pathways. Preserving confidentiality of privileged information mandates that no value in the program that is annotated with a high element from the policy lattice should flow to another value that is annotated with a low element. It is then the role of directional flow analysis to ensure that all points in the program that read values from a secure source do not write their results back to a public sink.

1.2 Verification of secure flow of information

Information Flow Analysis (IFA) is the preferred method for checking confidentiality (or its dual, integrity) of secret information as it flows through a program [112, 78, 87]. In its simplest form, IFA is an atomic bound constraint problem. The aggregate amount of information at a certain point i in a program is typically represented as an atomic label variable α_i - a variable with no deeper propositional structure. By inspecting the control flow of the program leading up to i, one identifies a lower bound lb_i on α_i. Additionally, by inspecting the control flow again, one identifies where the outputs produced at i are used and an upper bound for α_i is estimated. These bounds could be either other atomic variables or constants (elements from the security lattice). The pre-condition for preserving confidentiality is expressed as $lb_i \leq \alpha_i \leq ub_i$ and confidentiality is achieved if this condition is satisfied for all points in the program.

IFA is very similar to type inference in the presence of subtyping. In fact, much of the logical and algorithmic techniques for IFA are derived from the subtyping frameworks. Much like IFA, in a language that supports subtyping, flows between terms are validated against a pre-defined type lattice which describes the hierarchy amongst the types for these terms. However, every term in the program is seldom annotated with a type and types need to be derived for unannotated terms - a process that is commonly known as type inference. Types for unannotated terms are typically represented as variables and constraints on values that these variables can take is derived from data dependencies in the program. Given a constraint set C of inequalities between type variables, the constraint solver for subtyping frameworks checks if there an assignment of types to variables which satisfies all constraints in C as well as the pre-defined ordering amongst the types in the type lattice. This formulation of type inference and solutions for the same lend themselves well to the resolution of atomic bound constraints encountered in IFA.

Bound constraints on label variables are typically solved using graph theoretic techniques. Bound relationships that involve only variables are represented as a label-constraints graph. Upper bounds and lower bounds for a variable are identified through topological walks of the label-constraint graph by taking into account constant bounds on other variables. During this process, the policy lattice is queried frequently to answer meet, join and ordering queries involving constants that bound the variables. Solving these constraints involves identifying suitable substitutions for the variables that avoids violating the ordering of the policy lattice. Therefore, the efficiency of the resolution process is heavily dependent on the efficiency of its graph theoretic foundations.

Existing graph algorithms for solving these bound constraint problems use a one-size-fits-all approach and do not exploit the wide structural variety in either the lattices or the label-constraint graphs. Emerging instances of IFA require dealing with large and structurally diverse lattices as well as complicated programs with large constraint graphs. In the face of such intractable graphs, it is imperative to investigate the costs for graph algorithms that are used in the analysis, and design new ones that are better suited to the analysis.

1.3 Contributions

In this dissertation, we use graph decomposition to speed-up information flow analysis for programs where flow is governed by a security lattice. We consider two cases: the first case where expression are devoid of any security annotations (polymorphic expressions) and the second case where expressions are sparsely annotated with concrete security labels (annotated expressions).

For polymorphic expressions, we show that compaction of flow constraints graphs generated from polymorphic expressions is directly related to the transitive closure operation for DAGs. We propose a technique based on graph decomposition for speeding up the compaction process and show it to be highly effective even in face of intractable graphs. By means of the decomposition, we are able to abstract away from interconnections between vertices to those between sets of vertices (which we call clusters), which greatly reduces the computational costs. More importantly, provides seamless adaptivity over the entire structural spectrum lattices and constraint graphs. Thus we enjoy the fastest runtime for every point in the structure-spectrum of these label-constraint graphs as opposed to using a one-size-fits-all approach.

For annotated expressions, we investigate the computational costs for resolution of bound constraints on label variables for annotated expressions. We show that pre-processing lattices to answer lattice lookups for flow verification is the most compute-intense step. We then build upon the concept of clusters to design fast and adaptive algorithms that can preprocess a lattice to answer lattice lookups for pairwise meet, join and ordering relations in constant time. Similar to the simplification process, the computational costs of our lattice pre-processing algorithm are also a direct function of the latent scope for decomposition in the governing lattice.

The main contributions of this thesis are as follows:

1. Existing approaches to Information Flow Analysis have overlooked the pre-processing necessary for security lattices by assuming them to be trivially small. We derive a tighter complexity bound on atomic constraint solving for type-based IFA of programs by taking into account the costs for both pre-processing the security lattice and processing the constraints graph.

2. We introduce a novel notion of abstraction in lattice pre-processing by using decomposition of DAGs as an enabler. DAG decomposition lifts lattice pre-processing from the level of individual elements in the lattice to sets of elements. The resultant abstraction significantly cuts down the computational costs for pre-processing the lattice. An additional benefit of our approach is that the computational cost of the proposed pre-processing algorithm is a direct function of latent scope for decomposition in the lattice. Thereby, we achieve seamless adaptability throughout the structure-spectrum of lattices. We derive the asymptotic costs for the algorithm for pre-processing the lattice as well as the look-up costs.

3. We experimentally demonstrate the suitability of the proposed algorithms by testing them out with emerging examples of real-world security lattices and partial orders such as class hierarchies and powerset lattices which represent two ends of the structure-spectrum from trees to dense DAGs. Additionally, we also test our algorithm against randomly generated DAGs. For all these test cases, we show how our proposed algorithm based on graph decomposition make the pre-processing faster, as well as highly adaptive to the structure of the ordering under consideration.

4. We extend graph decomposition techniques to simplify constraint sets for expressions that are label-polymorphic. These are expressions that are devoid of program-

mer specified annotations, and a suitable substitution for label variables in these expressions cannot be obtained until they are put in a context with explicit annotations. Similar to the algorithms proposed for lattices, we develop algorithms for constraint simplifications that abstract away from individual vertices in the label-constraint graph to sets of vertices. We stress test the proposed algorithm with label-constraint graphs of expressions that have little scope for decomposition such as those derived from the standard library of FlowCaml [1]. We show that even in the face of such intractable graphs, the performance of the decomposition-based algorithm is favourable when compared to a standard algorithm.

1.4 Overview

The rest of the dissertation is organised as follows:

- We discuss a selection of related literature in chapter 2 including type-based program analysis, its applications (such as IFA), and its theoretical foundations such as decidability and complexity.

- We discuss lattice-directed type-based information flow analysis in chapter 3 and derive a tighter complexity bound for atomic constraint simplification and solving.

- In chapter 4, we extend graph decomposition based techniques to effectively pre-process a lattice for answering queries, like $\leq$, $\sqcup$ and $\sqcap$ for a pair of elements, in constant time. We present a novel pre-processing algorithm that enables constant time queries while seamlessly adapting its computation costs to the structure of the lattice.

- An experimental investigation of the benefits of our proposed algorithm for emerging applications for lattice-based security can be found in chapter 5

- We present analogous graph-decomposition based algorithms and experimental results for constraint simplification for label-polymorphic expressions in chapter 6.

- We review the significance of the thesis in chapter 7 and also discuss directions for future work.

[1]FlowCaml is the Caml language enhanced with security annotations. The author would like to thank Vicent Simonet and Francois Potter at INRIA, France for sharing the source code for FlowCaml

1.5 Publications

The following publications have been produced during the course of this research:

1. Santanu Kumar Dash, Sven-Bodo Scholz, Stephan Herhut, Bruce Christianson, *A scalable approach to computing representative lowest common ancestor in directed acyclic graphs*, Theoretical Computer Science 513(2), pp. 25-37.

2. Santanu Kumar Dash, Sven-Bodo Scholz, Bruce Christianson, *Modular design of data-parallel graph algorithms*, International Conference on High Performance Computing and Simulation (HPCS), 2013, pp. 398-404.

3. Santanu Kumar Dash, Sven-Bodo Scholz, Bruce Christianson, *Adaptive pre-processing of security lattices for Information Flow Analysis*, submitted to 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2015.

4. Santanu Kumar Dash, Bruce Christianson, *Boolean Matrix Product as a basis function in Information Flow Analysis*, under review at Information Processing Letters (IPL).

Chapter 2

Literature Review

2.1 Introduction

Computer programs find use in a wide variety of domains today, and have an equally wide array of requirements. Consequently, writing correct and effective programs is becoming increasingly hard. The drive to meet such demanding, and often conflicting, requirements has pushed program analysis to the forefront of computer science research lately. Through a combination of carefully crafted logical frameworks and efficient solving of constraints derived through these frameworks, automated program analysis relieves the programmer from the burden of meeting system requirements. Instead, due to the advances in program analysis, the programmer today can largely focus on building functionally correct programs and let program analysis verify, and potentially transform, programs to meet system requirements.

Since type systems are naturally designed to arrest data flows that violate policies of the type system, multiple forms of flow analysis can be achieved using type systems as an enabler. In type-based flow analysis, existing type-rules for the language are augmented with flow information and the type checking process is leveraged to identify undesired patterns of flow. IFA, which we discussed in chapter 1, is an instance of such an analysis. Augmenting the existing type system to conduct program analysis offers multiple advantages. Firstly, the soundness and correctness of the analysis are subsumed by that of the type system. Secondly, the type rules provide a localised setting for the analysis of language constructs which makes the analysis modular. And finally, since a program is type-checked anyway, a type-based flow analysis is efficient because it does not need a separate traversal of the syntax tree for the analysis. It is unsurprising, therefore, that type-based flow analysis have found application in diverse areas such as information flow analysis, alias analysis and resource reasoning etc.

There is a direct correspondence between context and flow sensitive program analysis, and the type discipline of polymorphic subtyping. Many instances of type-based flow analysis, in particular IFA, are based on polymorphic subtyping. The amount of aggregate information at a point in the program is a function of control flow in the program (directional analysis achieved through inclusion/subtyping on labels) and the context in which the flow is considered (polymorphism). Therefore we discuss the development of polymorphic subtyping in programming languages in section 2.2. This is necessary in order to understand the interplay of polymorphism and subtyping with other language features and the computational costs of deploying an analysis based on polymorphic subtyping.

From classical polymorphic subtyping found in type systems for modern day languages, we move onto applied polymorphic subtyping on annotated type systems in section 2.3. There, we discuss how polymorphic subtyping on type annotations can be used towards performing both flow-sensitive and context-sensitive program analysis. In particular, we discuss the advantages, disadvantages, computational costs and the equivalences between flow analysis and annotated types. Interestingly, both the classical polymorphic subtyping systems discussed in section 2.2 and the applied polymorphic subtyping systems discussed in section 2.3 can be reduced to a bound constraint problem which is the focus of this dissertation. A constraint based treatment of such systems is discussed in 2.2.3. The objective of the discussion in sections 2.2 and 2.3 is to demonstrate the relevance of adaptive bound constraint solving (which is the focus of this dissertation) to polymorphic subtyping systems and annotated type systems with polymorphic subtyping on the annotations.

Having discussed the theory behind polymorphic subtyping, we discuss two of its application areas in section 2.4. One of them is Type-based Information Flow Analysis (IFA) which is the focus of this dissertation. Type-based IFA has been successfully used in language-based security for checking and enforcing secure (protecting both confidentiality and its dual, integrity) information flow through programs. We also discuss advances made in alias analysis using polymorphic subtyping, where ownership properties of references are inferred to aid in removal of aliasing problems that are prevalent in object-oriented programs. Like IFA, alias analysis is an instance of an annotated type system that can pontentially benefit from the adaptive constraint solving techniques presented in this dissertation.

Having discussed the theory and applications of polymorphic subtyping, we discuss graph algorithms for solving the bound constraints encountered in polymorphic subtyping systems in section 2.5. This dissertation discusses adaptive approaches to simplifying and solving atomic bound constraints. Therefore, we discuss the graph algorithms that are used in the constraint solving, and recent advances that have been made in graph theory to accelerate these algorithms. Finally, we summarise the chapter in section 2.6

2.2 Polymorphic subtyping

Type-based flow analysis through polymorphic subtyping on labels has a wide range of applications. However, it is not always possible to infer the most general substitution for label variables that represent aggregate properties at program points. This is because type inference, and consequently type-based flow analysis is not decidable for every extension of the λ-calculus. In this section, we discuss the decidability of type inference for various extensions of the simply typed λ-calculus. Then for the decidable fragments which permit type inference with polymorphic subtyping, we discuss the computational costs to achieve type inference.

2.2.1 Polymorphism and decidability of type inference

`System F` [84] differs from the simply typed lambda calculus by introducing a notion of universal quantification over types. However, type inference for Curry-style variant of `System F` is undecidable without explicit type annotations [114]. This is a heavy price to pay for programming language design, and typically a restricted form of polymorphism called `let`-polymorphism [90] is used instead of the full-fledged polymorphism of `System F`. In `let`-polymorphism, type variables are only allowed to range over quantifier-free types (monotypes). This distinction between monotypes and polytypes (type schemes with quantifiers in prenex positions) makes the type inference simple and decidable. The Hindley-Milner type system [90] which forms the core of languages like ML is able to deduce the most general type for any program without the need for type annotations.

Having identified polymorphism that lends itself well to type inference, we now discuss integration of other language features such as references and recursion into a first order polymorphic calculus. Monomorphic recursion can be easily achieved in the Hindley-Milner type system by using a fixed point operator. Type inference for monomorphic recursion without let-polymorphism has the same linear cost as typing the simply typed λ-calculus. However, it is possible that the body of a recursive definition passes a polymorphic argument to itself. This situation is called polymorphic recursion (also know as Milner-Mycroft typability) and was first studies in [77]. It is known to be undecidable in the absence of explicit type annotations [54, 62]. Therefore, programming languages like ML normally support monomorphic recursion with a limited support for polymorphic recursion through a combination of `let`-polymorphism and monomorphism, or through explicit type annotations. Polymorphism and memory references do not mix well. It was shown in [110] that type inference in the presence of polymorphic references is undecidable. Therefore, a watered down version of references, known as value restriction [116], is used in mainstream programming languages that support first order polymorphism .

2.2.2 Type inference with subtyping

So far, we have discussed the type inference of the typed λ-calculus augmented with references and recursion. A combination of polymorphism, recursion and references provides a realistic platform for language development, and indeed it has been at the heart of languages such as ML. However, in this thesis, we work towards an efficient constraint solver for polymorphic subtyping on type annotations. In this context, it is important to understand the efforts that have been made towards mixing polymorphism and subtyping by the type theory community, and the theoretical challenges that have been faced. Hence, we now proceed to discuss the efforts that have been taken to include subtyping in the context of the core calculus discussed so far. Multiple efforts have been made towards bounded quantification - a combination of polymorphism and subtyping. Some of the early works that discussed type inference in the presence of subtyping were [74] and [45]. Both these works generated constraint sets from the syntax of the program which were indicative of type inclusions. Subsequently, these papers presented algorithms for solving these constraints and showed the inference to be sound and complete. While [74] interleaved the generation and simplification of the constraints, the algorithm presented in [45] dealt with structural subtyping and generated coercion sets for type conversions.

In subtyping-based systems, deciding whether a term can have a type is reduced to the problem of checking whether a system of set inclusion constraints, derived syntactically from the source code, has a solution. This was noted in [1] and a general framework for solving type inclusion constraints was proposed there. The techniques described in [1] were shown to be effective for a rich type system with $\top$, $\bot$, function types, constructor types, recursive types as well as restricted forms of union and intersection types. While [1] provided an elaborate framework for solving subtyping constraints, it stopped short of discussing subsumption of constrained type schemes. Subtyping of constrained type schemes was discussed in [111] and is necessary for two reasons: firstly, separate compilation through modules and functors which require signature matching, and secondly simplification of type schemes creates a need to check whether the simplified scheme subsumes the original scheme.

2.2.3 Type inference as constraint solving

It was observed in [81] that the constraint system is orthogonal to the type-theoretic aspects. Therefore, they proposed a variant of the Hindley-Milner type system called HM(X) which extended the original Hindley-Milner system with constraints, where X could be instantiated to a specific constraint system. This enabled instantiating the inference engine for a variety of constraints including unification and inclusion constraints. It was also discussed that, under certain restrictions on X, type inference would always compute the most general type for each term. The expressiveness of such an approach was validated through instances of HM(X) both equality and subsumption based inference in [90]. Indeed, it was shown [90] that the type system of ML can be expressed as a natural extension of HM(X) where X is instantiated to an equality constraint system. Further work on HM(X) has produced a syntactic soundness proof [89] and an extension of HM(X) with bounded existential and universal data-types [99]. The use of bounded existential and universal types uses polymorphic subtyping at the level of the type system. Therefore, our work on atomic bounded constraint solving over a lattice is applicable to such systems too.

2.2.4 Computational costs of subtyping

The decidability of HM(X) is dependant on the constraint system that X is initialised to. Since the focus of this research is subtyping constraints, we now discuss the decidability and complexity results for structural and non-structural subtyping. Checking satisfiability of structural subtyping inequalities for finite types (types represented as a finite tree i.e. not equi-recursive) over posets was shown to have a PSPACE lower bound in [107] and to be PSPACE-complete in [44]. The results presented in [107] were extended to include recursive types and atomic subtyping in [109] and complexity of checking statisfiability was shown to be DEXPTIME. While, it was shown in [107] that checking satisfiability over a lattice was achievable in PTIME, techniques presented in [107] only checked satisfiability and did not find a solution. A polynomial time algorithm for solving atomic constraints over a poset $\langle C, \leq \rangle$ of atomic subtypings such that $\langle C, \leq \rangle$ is a disjoint union of lattices was given in [108]. The satisfiability of non-structural subtyping constraints for finite and recursive types over posets was shown to be PSPACE-complete and DEXPTIME-complete in [80].

While satisfiability involves checking whether a set of constraints has a solution, a significant amount of work has also been done towards representing the constraints in as compact a manner as possible. This aids in both readability and efficiency of verification of subtyping relationships between type schemes. The subtype simplification problem for atomic subtyping, which involves constraints between atomic variables and constants, was discussed in [92]. The authors in [92] also established that the worst-case size of most general typings has an exponential lower bound. In subsequent work, it was shown that entailment of the subtyping relation (determining whether a set of constraints entails a subtyping relation) for simple constructed types had a structural lower bound of coNP-hard and an upper bound of coNP [53]. Recursive subtype entailment was studied in [55] and it was shown that nonstructural subtype entailment is PSPACE-hard for finite trees (simple types) as well as infinite trees (recursive types). It was also shown in [55] that for the case of structural subtyping, subtype entailment over infinite trees is PSPACE-complete when the ordering on the trees is generated by a lattice of type constants. A series of practical algorithms for simplifying constrained type schemes (polymorphic types) were suggested in [88, 85]. The simplifications discussed in these works were largely aimed at non-structural subtyping systems. Further algorithms for simplifying structural subtyping constraints were discussed in [101]. It was identified, through the advances in structural and non-structural subtyping systems, that complicated notions of subtyping involve quantifiers and a first order theory of subtyping constraints was developed in [104]. Subsequently, it was shown in [66] that the first order theory of structural subtyping of non-recursive types is decidable. This discussion is relevant to the dissertation because much of the logic used in this dissertation is based on first order theory of structural subtyping.

2.3 Type-based Flow Analysis

Annotated types i.e types augmented with flow labels, are a useful vehicle for flow analysis. In type-based flow analysis, the type derivation of a program forms the basis of static analysis. Each type rule provides a localised setting for the analysis of language constructs, and the ability to use existing types make type-based analysis an attractive proposition. One of the first papers to explore the equivalences between type systems and control flow analysis was [51] where a series of equivalences were established between type systems and control flow analysis. Since these equivalences were established, type-based flow analysis has been extensively used in the areas of language-based security [93, 50], alias analysis [31], resource-usage analysis[61, 57], differential privacy [37], etc.

Type-based flow analysis annotates the type structure of terms with a flow label. It then performs various forms of flow analysis, investigating the reachability between values at multiple program points by deriving relationships between the labels at the these points. The success of type-based flow analysis is due to the multiple advantages it offers over other forms of flow analysis such as constraint satisfaction and abstract interpretation. Since it can be built on top of the existing type system rules for a statically-typed language, it offers multiple advantages such as the following:

- **Simplicity**: Types provide a localized setting to reason about language constructs. Annotating types with static information and using type annotations as discriminators, one obtains a convenient basis for designing static analyses [82].
- **Efficiency**: Since statically typed programs need to be type-checked anyway, a traversal of syntax trees for type checking can be augmented with techniques for type-based flow analysis. This provides significant advantages over trying to perform a similar kind of analysis on a dynamically-typed program [82]. Type-based flow analysis has been successfully integrated with static-type checking to drive code specialisation without additional passes of the syntax tree in array-programming languages like Single-Assignment C [98].
- **Correctness**: Perhaps the biggest advantage of using types to perform flow-analysis is that the correctness of the analysis is subsumed by the correctness of the annotated type-system. The correctness of the type system with repect to semantics is known as type soundness. The well known method for proving type soundness that is based on progress and preservation carries over to type and effect systems as well [86].

A number of equivalences between flow analysis and annotated type systems were considered in [51] but these were largely context-insensitive comparison of the two techniques. The first description of a flow and context sensitive type-based analysis was given in [75]. Two context-sensitive analyses for a simply typed programming language were presented in [75]: one was inspired by let-polymorphism and another by polymorphic recursion. The computational complexity for the context-sensitive flow analysis for let-polymorphism was $O(n^7)$ and that for polymorphic recursion was $O(n^8)$ where n is the size of the explicitly typed program. These complexity measures were improved in [91] where $O(n^3)$ algorithms for the two cases were discussed. The reduction in complexity was achieved by avoiding copying constraints in label-polymorphic expressions to instantiation sites. Instead, the instantiations are remembered as a separate instantiation constraint at all sites where the label-polymorphic expression is used. The algorithm discussed in [91] further reduces the constraints to a context-free language (CFL) reachability problem over a flow graph for which cubic-time algorithms exist. In this dissertation, we design a novel approach to solving flow constraints derived from annotated type systems. Our approach leverages the nature of interdependencies between flow variables to design an adaptive solver. Therefore, the literature on type-based flow analysis discussed in this section can directly benefit from the content of this dissertation.

2.4 Applications of Type-based Flow Analysis

As mentioned above, our work on adaptive constraint solving for annotated type systems is directly relevant to multiple forms of flow analysis that are based on polymorphic subtyping over labels that describe flow properties. In this section, we discuss some instances of type-based flow analysis which are based on polymorphic subtyping. One of them, Information Flow Analysis, which is described below, is used as a test case to obtain experimental results in this dissertation.

2.4.1 Secure Information Flow Analysis

Type-based flow analysis has been successfully applied to language-based security for checking and enforcing confidentiality and integrity of privileged information [112, 103]. In this section, we discuss the advances in using type systems to ensure information flow control. We first discuss theoretical advances made in augmenting type systems to achieve information flow analysis. We then present work done towards incorporating these advances in type systems of real-world programming languages. Finally, we discuss theoretical advances on the policy side of things and show the different approaches taken to classify a flow as desirable or undesirable.

The invaraint that needs enforcement in information flow analysis (to achieve presenvation of confidentiality) is that no assignment to variables that can be publicly read be allowed in expressions that take any private variables as inputs. The seminal work that triggered research in static enforcement of secure information flow was [38]. In [38], a static approach to information-flow analysis was presented and it was shown how static analysis removes runtime overhead for security checks. This analysis prevents both explicit and implicit flows (through the control flow) statically. The framework presented in [38] was able to detect insecure information flow through both explicit (assignments) and implicit (control flow) channels. However, a proof of soundness proof was not developed until the analysis was formulated as a type-based flow analysis problem in [112].

Soundness was shown in [112] by proving non-interference. Informally, non-interference [48] is satisfied if the values of the public outputs of a program do not depend on its secret inputs in such a way that observing the values of the public outputs lets a malicious attacker guess the value of the secret input. When expressed in terms of program execution for a deterministic language, non-interference requires that if the program is run with different secret inputs, while holding the public values fixed, the public output must not change [50]. A type-based formulation of non-interference was given in [50], and since then the predominant technique for enforcing secure information flow statically is type-based information flow analysis. However, static enforcement is not a complete solution for enforcing secure information flow. For dynamically-typed languages like Javascript, it is impossible to check secure information flow through a static type-based approach. Indeed, for dynamic and hybrid approaches to enforcing secure information flow, we refer the reader to two excellent surveys in [50] and [93] that detail issues in and techniques for information flow control. Much of the material on static information flow analysis in this section is derived from [93]. This dissertation deals with static analysis of deterministic programs. More details about dynamic analysis can be found in [93]; our work is not directly applicable to dynamic analysis.

In secure type systems the types for variables and expressions are annotated with labels, and an ordering on the labels specifies a security policy for the use of the typed data. The static analysis proposed in [38], which is commonly known as Denning-style analyses, prohibits both undesired implicit and explicit information flows. This is achieved by keeping track of the aggregrate security level achieved through the control flow in addition to the data flow. The aggregate level of security achieved through the control flow is commonly known as the security level of the program counter (*pc* for short). The object of secure information flow is thus to prevent any inadvertent declassification at any point in the program to a level that is lower than the current *pc*. In other words, public side effects are disallowed in secret contexts. This enforcement scheme is know as flow-insensitive, since it does not allow the security classification of a program locations to vary. In contrast, a flow sensitive analysis through a type system was proposed in [59] for simple While programs where the type systems are parametrised over the choice of a flow lattice - this enables different abstractions for flow correctness at different program points. Additionally, it is also shown in [59] how any flow-sensitive program can be transformed into an equivalent program typable in a flow-insensitive type system. The concept of flow-sensitive information-flow security was originally discussed in [5]. However, the enforcement in [5] was achieved through a Hoare logic rather than a type system.

As far as practical implementations of information-flow security are concerned, work has been done to extend Denning-style analyses to many different programming languages. Such an analysis forms the core for information flow analysis frameworks like JFlow for Java [78, 79] and FlowCaml for Caml [87] and a similar framework for Haskell [67]. Central to these frameworks is how exceptions are handled in a mainstream programming language. Exceptions are different to other language constructs because they trigger unexpected control flow jumps in the program and therefore require special treatment. A novel treatment of secure exception handling is discussed in [9] where the programmer has the choice of whether to handle or not handle secret exceptions. The security mechanism ensures that in the former case, exceptions are never handled and in the latter case, they are always handled using mainstream restrictions. The work in [9] further shows that such an approach is sound with respect to termination-insensitive noninterference. Amongst other work for secure information flow for ML-like languages, expressive dynamic information-flow policies called flow-locks and the associated type system were presented in [28]. Flow-locks were subsequently recast using a knowledge based definition in [29] and extended to a role-based multi-principal settings in [30].

Having discussed efforts at enhancing type systems to achieve secure information flow in a setting where security annotations are ordered over a lattice with two elements, we now discuss some variations to lattice-based policy models that have been considered. A local flow policy that allows computations in its scope to implement information flow according to the local policy was discussed in [3]. A type and effect system that enforces such a local flow policy was also discussed in [3]. A notion of syntactic escape hatches that delimit the amount of information released, and a type system that achieves this, were discussed in [95]. The notion of delimited release was extended with code locality in [8] where a type system that forcibly disallows declassification in secret contexts was discussed.

Relaxed models of non-interference were discussed in [68] where the notion of relaxed noninterference generalises traditional pure noninterference. Relaxed non-interference give rise to interesting lattice orderings relevant to this dissertation and form one of the test cases for our experiments on answering lattice queries efficiently. A generalised framework of downgrading policies was presented in [68] where policies could be specified in a language and statically enforced through a type system. A notion of abstract non-interference was discussed in [47] where a more relaxed form of non-interference where the observational power of attackers are limited; it deals with attackers that observe only properties of data rather than exact values.

In [117], a model of information flow was presented with the class representing a collection of objects with the same structure as an abstract property. Thus, from the point of view of lattice-directed information flow control, classes would represent elements in the security lattice and the subclass relation would represent the ordering between the elements. Class-level non-interference mandates that a class is secure if observing the output of any of its public methods does not reveal any type information regarding its inputs.

2.4.2 Alias Analysis

Aliasing is a prevalent problem in object-oriented programming. Bugs due to unintended aliasing are hard to pin-point and can lead to unexpected results. Ownership types address this issue by introducing a notion of ownership of objects which directs how references can be passed and used. Proving that an invariant is preserved for a structure when a program is executed becomes difficult if there is unmitigated proliferation of references to the structure.

Ownership can be used to control accesses to objects and restrict passing of references. The restriction enables easier reasoning about programs for modular verification [41, 16, 76, 14, 12, 15], concurrency [26, 83, 25, 96, 27, 65, 24], security [69, 32, 13, 102] and memory management [118, 7, 119, 105].

There are two major approaches to enforcing topological restrictions (which object owns what references) on the program heap and enforcing encapsulation. Encapsulation of references in this fashion gives the programmer power to restrict where the references can be used and hence reduces problems related to aliasing. In the owners-as-dominators approach, a given object can only be accessed if one obtains permission to access the owner of the object. Consequently, program heaps are tree-structured i.e. an object is inside its owner. The owner-as-dominator discipline mandates that all references to an object pass through its owner. In contrast, the owner-as-modifier approach relaxes the owners-as-dominators approach. This means that an object can be modified by its owner and by its peers i.e. objects that have the same owner. An excellent survey of techniques for checking object ownership and limiting aliasing can be found in [31]. A large part of the material on static techniques for inference of Ownership Types that is relevant to this dissertation is abridged from [31] and these static techniques can potentially benefit from the results described in this dissertation.

Programs that make use of Ownership Types systems normally require annotations to express the types for objects and ownership properties. While this is trivial for small programs, it can be cumbersome for large programs. It is not just the application code that needs annotations but the library needs to be annotated as well to describe ownership properties. In view of this, it is necessary to automatically infer ownership properties wherever possible, to relieve the programmer of having to annotate every part of the application code and the library. One of the earliest techniques for inference of Ownership Types using constraints, called AliasJava was presented in [2]. In the AliasJava system presented in [2], the inference was too fine grained, as multiple alias parameters were used to describe ownership properties. Therefore, a class which represents a collection of values and methods was shown to end up with potentially hundreds of inferred parameters. Another approach to Ownership Type inference based on the escape analysis technique [23] was developed and presented in [23]. However, similar to [2], the ownership parametrisation was too expansive. Consequently, the proposed algorithm resulted in a large number of parameters.

As a part of the build-up towards literature on ownership inference, we discuss some advances that built on the results of the above-mentioned approaches and enabled precise and efficient aliasing analysis through ownership types. A generalisation of a points-to analysis was presented in [70] to infer uniqueness and ownership like properties for object-oriented programs. The presented tool is called Uno, and it combined constraint-based intraprocedural and interprocedural analyses to collect information about encapsulation properties. In subsequent research, an Andersen-style points-to analysis [6] was employed as part of a static algorithm to infer ownership properties for the owners-as-dominators and owners-as-modifiers protocols. In contrast to [70] where exclusive ownership is captured (if the contents of a field are passed temporarily to an object, the field is counted as non-owned even if it remains within the dominance boundary of the enclosing object), the approach in [72] captured the owner-as-dominator relationship and handled the exclusive ownership more precisely.

A static analysis technique that infers dominance relationship between objects was presented in [73]. It was shown how dominance inference is central to ownership type inference, and the dominance inference framework was used as a building block towards ownership type inference for the owners-as-dominators protocol with one parameter. The algorithm computed approximations of the object graphs, and candidate ownership annotations were derived from a dominance tree built using a variation of must-point-to information [39, 60]. Later in [58], this work was extended in two ways. Firstly, the framework in [58] accepted manual annotations to direct the inference and secondly, unlike [73], the inference in [58] provided manual optimality guarantees i.e. it types each variable with the most general type.

Another inference system for ownership properties of objects which accepted programmer annotations was presented in [40]. Similar to [58], the approach first generates a set of ownership constraints based on program semantics, and then encodes the constraints as a boolean satisfiability problem. After the constraints have been solved, the second part lets the programmers fine-tune the typing by specifying preferences for certain typings. These preferences can be specified by supplying partial annotations to the program. However, unlike [58], there is no ranking over typings and therefore, it is hard to scale the inference to larger programs.

2.5 Graph algorithms for solving subtyping constraints

In this thesis, we propose adaptive means to solve constraints arising in program analysis. In particular, we look at constraints that are produced when structural subtyping constraints are reduced to the level of atomic constraints, where both the l.h.s and r.h.s of the inclusion constraint are either a non-constructed variable or a non-constructed constant. The standard algorithm for solving such constraints is to represent the relation between the atomic variables as a directed graph, fuse any cycles in the graph, and perform a topological walk of the graph to obtain lower and upper bounds for all the variables by taking into account the constants that bound them [101]. For our work, we assume that the constants are themselves arranged as a lattice. Therefore, the constraint solving algorithm frequently needs to query the lattice for operations like join ($\sqcup$) and meet ($\sqcap$), and ordering ($\leq$) relationships involving constants. While the topological walk is straightforward, the lattice queries are non-trivial to answer, especially if the lattices under consideration are large in size.

In this section, we give an overview of graph theoretic approaches that have been proposed to answer lowest common ancestors (LCAs) for Directed Acyclic Graphs (DAGs) in the literature. DAGs are a natural representation for pre-processing lattices in order to answer lattice queries in constant time which is a requirement for an efficient atomic constraint solver. When a lattice is represented as a DAG, obtaining the LCA for a pair of vertices in the DAG is analogous to obtaining the $\sqcup$ of two elements in the corresponding lattice. Therefore, a discussion of LCA algorithms for trees and DAGs in general is important and relevant. The $\sqcap$ query is the dual of the $\sqcup$ query and the $\leq$ query is a special case of the $\sqcup$ query where, in a query involving two elements, one of the elements is the result of the $\sqcup$ operation. In section 2.5.1, we discuss approaches presented in the literature for pre-processing tree in order to answer LCA of two vertices in the tree in constant time. We discuss how these techniques have been extended to answering LCA queries for DAGs in general in section 2.5.2.

2.5.1 LCA computation in trees

While investigating multidimensional discrete range searching problems, the authors of [46] observed the equivalence between unidimensional range minimum searching and the LCA computation on Cartesian trees - a heap-ordered binary tree derived from a sequence of numbers. The unidimensional range minimum query is defined as follows.

Definition 1. *Given an n-element array $A[1..n]$, the range minimum index query $RMQ_{idx}(i, j)$ returns the index k of the smallest element $A[k]$ in the sub-array of A beginning at position i and ending at j.*

Since the Cartesian tree is a binary tree, efficient schemes needed to be developed for the computation of LCA on nodes belonging to a generic tree i.e. a tree where the vertices have fewer/more than two children. To achieve this a labeling scheme for nodes was proposed in [49]. This scheme was able to answer LCA queries in constant time after a linear time preprocessing. However, the preprocessing for the algorithm presented in [49] remained complicated until some of the preprocessing steps were removed in [97]. A parallel approach to computing the LCA of two nodes using the simplified algorithms was also presented in [97].

A completely different approach to preprocessing trees for computing LCAs of two nodes was presented in [21]. The approach relied on the Euler tour of the tree [35] to generate a sequence of integers as an input to the preprocessing phase.

It was shown in [21] that the LCA of nodes `u` and `v` is always encountered between first visits to `u` and `v` during the Euler Tour of the tree. Let E store nodes in Euler Tour sequence and D store the depths of those nodes in the same sequence. For any two nodes `u` and `v`, let u_{idx} and v_{idx} denoted the indices of the first occurrence of these nodes in E. Then, $RMQ_{idx}(u_{idx}, v_{idx})$ on the array D returns the index of the LCA of the two nodes and $E[RMQ_{idx}(u_{idx}, v_{idx})]$ returns the LCA itself. However, it was observed in [21] that the RMQ_{idx} queries on the depth array actually form restricted domain problem where consecutive entries differ by ±1. This restricted domain property was exploited to develop efficient schemes for answering the $\pm1 RMQ_{idx}$ and subsequently the LCA query in constant time after linear time preprocessing [4] [19]. In this disseration, we build upon these techniques to design graph algorithms for adaptive resolution of bound constraints.

2.5.2 LCA computation in DAGs

Interest in answering LCA queries vertex-pairs in DAG in constant time after pre-processing the DAG is recent and was initially studied in full detail in [20]. The authors reduced the all-pairs LCA problem to all pairs shortest distance query and proposed a solution that had a preprocessing time of $O(n^{2.688})$ (n being the number of vertices in the DAG) and constant query time. The exponent is derived from a modified matrix multiplication operations. Techniques from rectangular matrix multiplication discussed in [33] and [34] were used in [36] and [18] to further reduce the computational complexity of preprocessing to $O(n^{2.575})$. Similar to [20], the exponent in this case also results from a modified matrix multiplication operation.

Apart from the general results pertaining to LCA computation in DAGs, there have also been techniques developed to address special classes of DAGs. A path cover based approach to computing LCAs in DAGs having low width was discussed in [64]. The algorithm had a preprocessing time of [1] $\tilde{O}(n^2 w(G))$ where $w(G)$ is the width of the DAG and constant querying time. This approach was also validated for DAGs having small depth, and for such test cases, the algorithm was shown to possess the same worst case complexity as the costs reported in [36].

For sparse DAGs, techniques to compute all-pair representative LCAs with a time complexity of $O(nm)$ were discussed in [36] and [18], here m is the number of edges in the graph and n is the number of vertices in the DAG. It was further shown in [42] that all-pair representative LCAs can be computed in $O(nm_{red})$ where m_{red} is the number of edges in the transitive reduction of the DAG. Based on the results regarding the number of strongly independent vertices in random DAGs [17], the authors of [42] note that the complexity works out to $O(n^2 \log n)$. However, the worst case complexity for this algorithm stands unchanged at $O(nm)$ because computation of transitive reduction itself takes $O(nm)$ time.

Similar to the techniques based on reachability matrices reported in [36] and [18], we also use matrix multiplication as the basic ingredient in our approach. Therefore, we demonstrate the advantages of our algorithm by comparing it with the best reported algorithms based on matrix multiplication. As discussed earlier in this section, these algorithms have time and space costs of $O(n^{2.575})$ and $O(n^2)$ respectively [36, 18].

[1] $\tilde{O}(f(n)) = O(f(n)\,\mathrm{polylog}(n))$

2.6 Summary

We began this chapter by discussing the interplay of polymorphism and subtyping with other features of programming languages. This formed a basis for discussing the applications of polymorphic subtyping to program analysis. We discussed why a flow analysis framework based on polymorphic subtyping on type annotations is a superior approach to other forms of static flow analysis for a variety of optimisations. We also discussed two specific instances of annotated type systems applied to program analysis. Having discussed the computational costs of subtyping systems, we also showed why graphs algorithms are central to solving constraints on labels. In the rest of this dissertation, we develop novel graph algorithms for solving the atomic inclusion constraints encountered in program analysis. Such constraints are the foundational building block in type-based flow analysis where the discrete label constants are ordered over a lattice. In the next chapter, we discuss type-based IFA in detail which helps us understand the connection between annotated type systems and graph theory. Since the techniques we develop in this dissertation are tested specifically for IFA, the next chapter is also useful for interpreting our theoretical and experimental results.

Chapter 3

Information Flow Analysis

3.1 Introduction

Information Flow Analysis (IFA) verifies whether the flow of information through a program obeys a pre-defined security policy. The information could either be the input data to the program or a result computed from the input data using expressions described in the programming language. The motivation behind IFA is to prevent information that is deemed classified from getting inadvertently de-classified. Type checking for programming languages is a natural approach to inspecting how information flows through a program. Type checkers inspect every term in the grammar for type compatibility between expected inputs to an expression and actual parameters passed to it. Thus, the type information for an expression can be used to carry the amount of privilege associated with it as well, and information flow analysis can build on the type-checking process.

However, conventional type-checking mandates explicit declaration of types for program variables. It is often cumbersome to do so, and one of the prime reasons behind popularity of languages like OCaml, Haskell, etc is that the type checker can derive the most general type for an expression without a need for explicitly declaring types for variables. The type checker manages to derive the most general type for expressions through a process known as type inference. The inference process represents types for terms as variables, generates constraints on these variables depending upon where the terms are used, and finally solves the constraints to identify a suitable substitution for the variables.

The type inference process lends itself well to IFA. Similar to declaring types for terms, declaring privilege levels for all terms is cumbersome. Therefore, ideas from type inference can be used to perform information flow analysis as well. Similar to the type inference process, information flow analysis of programs consists of two steps. In the first step, flow constraints are generated from the program and rewritten to express them in a simplified manner. In the second step, the simplified constraints are resolved by checking against a predefined lattice of security labels which lays out the security policy. The type system is typically relied upon for generating flow constraints. Simplification is typically achieved by compacting flow constraint graphs derived from function abstractions such that, for a given function, only the input variables are related to the output variables. Resolution, on the other hand, involves querying a pre-defined lattice of security labels to assert confidentiality of information.

In this chapter, we show how information flow analysis can be achieved using type inference. We first discuss information flow analysis using type-checking where every expression is annotated with its type and the privilege level of the information it holds. Then, we discuss a typing approach which is based on constraints and lends itself well to both type inference and information flow analysis. We present all aspects of constraint based typing and IFA, such as constraint generation, constraint simplification and resolution of constraints, and also discuss the computational costs for all the aspects. The main technical contributions of this chapter are as follows:

- We estimate the computational costs of constraint simplification and resolution. Unlike similar works [10, 87, 78, 11, 52], our approach does not assume that lattice queries can be computed in constant time with little overhead. Instead, we precisely quantify the cost of pre-processing the lattice to answer queries in constant time.

- We identify the computational bottlenecks in both simplification and resolution of information flow constraints. We then formulate the computational costs of the bottlenecks in terms of Boolean Matrix Product (BMP), and hence present a compact representation of the complexity costs for the bottlenecks.

The rest of the chapter is organised as follows. We present information flow analysis using types in section 3.2 and a corresponding constraint based approach in section 3.3. We discuss constraint solving in section 3.4 and simplification of constraints on type schemes in section 3.5. We discuss the computational costs of a constraint-based approach to information flow analysis in section 3.6 and its bottlenecks in 3.7. Finally, we summarise the contents of this chapter and motivate the material for chapters 4, 5 and 6 in section 3.8.

3.2 IFA using annotated types

Information flow in programs consists of two kinds: explicit flow and implicit flow [120, 78]. Explicit flow is normally due to assignments made to program variables, whereas implicit flow is a consequence of program variables depending indirectly on the result of other expressions. Consider the following expression:

```
λs.λx.λy.(if (s > 0) then x = y else x = 5)
```

In the example above, the information flow from variables `y` to `x` is explicit. On the other hand, the value of `x` also depends indirectly on the value of `s`. During subsequent execution, any knowledge of the value of `x` can be used to guess properties of `s`. In other words, there is implicit flow of information from `s` to `x`.

There is a direct relation between implicit information flow and manipulation of memory references. In the function abstraction above, the value of `x` may not be observable after the function is applied unless `x` is a memory reference. For languages that support imperative features, information flow analysis needs to be aware of all references that could be manipulated in bodies of function abstractions. These references could potentially be read during execution of other abstractions and leak information implicitly. Due to this observation, implicit flow is often approximated as information flow through side-effects such as manipulation of memory references.

3.2.1 Capturing implicit and explicit flow through types

Type systems are a natural vehicle for enforcing information flow policies [86]. The primary objective of a type system is to ensure program safety, and information flow control can be achieved by leveraging on the type checking process. Explicit information flow is controlled by annotating types of expressions with *security levels* - an indicator of how secure the expression needs to be. In this dissertation, we typically use `l` as a variable representing a security level. The type checking process only allows explicit flow of information from an expression at a lower security level to one that is at a higher level. In order to control implicit flows, a standard technique is to use a *program counter* (`pc` in short). The value of `pc` is different at different points in the program and is indicative of the information that can be learned through knowledge of the control flow path taken to reach a particular point in the program. Type checking ensures that the side effect of an expression has a security level that is at least as restrictive as the `pc`.

Having described briefly how types can aid in information flow control, we are now ready to describe an annotated type system to achieve the same objective. We deal with three types of type constructors: unary type constructors for base types, binary type constructors for references, and ternary type constructors for function types. One may notice that the arity is one more than the standard arity for base types and references. This is because we now annotate types with levels for these type constructors. In addition to privilege levels, function types are also annotated with a $\mathtt{pc}$ variable for representing the net effect of applying the function. Hence, function types are quaternary constructors. The set of types used in our discussion is summarized in figure 3.1.

$$\mathtt{t} ::= \mathtt{unit} \mid \mathtt{b}^{\mathtt{l}} \mid \mathtt{t\,ref}^{\mathtt{l}} \mid (\mathtt{t} \xrightarrow{\mathtt{pc}} \mathtt{t})^{\mathtt{l}}$$

Figure 3.1: Set of types

We now present a set of type rules to enforce information flow control for the lambda calculus with references and let-polymorphism. While these type rules can be easily extended to incorporate polymorphism through universal quantification, we leave out polymorphism for time time being to keep the presentation simple. Polymorphism in the traditional and the constraint-based setting is discussed in section 3.3.2.

The typing rules are laid out in figure 3.2. Here Γ is a partial mapping from program variables to types and $\mathtt{M}$ is a partial mapping from memory locations to types. A typing judgement is typically written as $\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{e} : \mathtt{t}$ and read as: under the assumptions of the program counter $\mathtt{pc}$, the type environment Γ and types for memory locations contained in $\mathtt{M}$, the expression $\mathtt{e}$ has type $\mathtt{t}$.

The type rule V-Var is standard. A variable $\mathtt{x}$ has type $\Gamma(\mathtt{x})$. The rule V-Abs typechecks function abstractions. The type of the function carries with it a $\mathtt{pc}$ which is an aggregation of the function's latent side-effects. This piece of information is useful while typechecking function applications. Rule E-App typechecks function applications. Neither the result of the function nor its side effects should leak information about the function's identity. The former is achieved by ensuring that the function's security annotation $\mathtt{l}$ guards the type of the output expression (written as $\mathtt{l} \lhd \mathtt{t}$) while the latter is achieved by ensuring that the function body runs at $\mathtt{pc} \sqcup \mathtt{l}$.

The rule E-Ref checks memory allocation operations. When a memory location is created, the security level for the created location needs to be at least at the level $\mathtt{pc}$ to prevent implicit flows. Therefore, $\mathtt{pc}$ guards $\mathtt{t}$ in the premise of the rule E-Ref. Rule E-Deref checks that the result of looking up the contents of a reference does not leak information about the reference itself. This is achieved by ensuring that the reference

$$\frac{\mathtt{t} \in \Gamma(\mathtt{x})}{\Gamma, \mathtt{M} \vdash \mathtt{x} : \mathtt{t}} \text{ (V-VAR)}$$

$$\frac{\mathtt{pc}, \Gamma[\mathtt{x} \mapsto \mathtt{t}'][\mathtt{f} \mapsto (\mathtt{t}' \xrightarrow{\mathtt{pc}} \mathtt{t})^{\mathtt{l}}], \mathtt{M} \vdash \mathtt{e} : \mathtt{t}}{\Gamma, \mathtt{M} \vdash \mathtt{fix\,f}.\lambda\mathtt{x.e} : (\mathtt{t}' \xrightarrow{\mathtt{pc}} \mathtt{t})^{\mathtt{l}}} \text{ (V-ABS)}$$

$$\frac{\Gamma, \mathtt{M} \vdash \mathtt{v_1} : (\mathtt{t}' \xrightarrow{\mathtt{pc} \sqcup \mathtt{l}} \mathtt{t})^{\mathtt{l}} \qquad \Gamma, \mathtt{M} \vdash \mathtt{v_2} : \mathtt{t}' \qquad \mathtt{l} \lhd \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{v_1 v_2} : \mathtt{t}} \text{ (E-APP)}$$

$$\frac{\Gamma, \mathtt{M} \vdash \mathtt{v} : \mathtt{t} \qquad \mathtt{pc} \lhd \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{ref\,v} : \mathtt{t\,ref}^{*}} \text{ (E-REF)}$$

$$\frac{\Gamma, \mathtt{M} \vdash \mathtt{v} : \mathtt{t\,ref}^{\mathtt{l}} \qquad \mathtt{l} \lhd \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{!v} : \mathtt{t}} \text{ (E-DEREF)}$$

$$\frac{\Gamma, \mathtt{M} \vdash \mathtt{v_1} : \mathtt{t\,ref}^{\mathtt{l}} \qquad \Gamma, \mathtt{M} \vdash \mathtt{v_2} : \mathtt{t} \qquad \{\mathtt{pc} \sqcup \mathtt{l}\} \lhd \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{v_1} := \mathtt{v_2} : \mathtt{unit}} \text{ (E-ASSIGN)}$$

$$\frac{\Gamma, \mathtt{M} \vdash \mathtt{e_1} : \mathtt{s} \qquad \mathtt{pc}, \Gamma[\mathtt{x} \mapsto \mathtt{s}], \mathtt{M} \vdash \mathtt{e_2} : \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{let\,x} = \mathtt{e_1\,in\,e_2} : \mathtt{t}} \text{ (E-LET)}$$

Figure 3.2: A type and effect system for information flow analysis

annotation $\mathtt{l}$ guards the type of value at the dereferenced location. Rule E-ASSIGN involves writing to a memory location that has already been created. In order to successfully typecheck the assignment, we need to ascertain two things. Firstly, we need to check whether we have the privilege to write to the memory location. Similar to E-REF, this is achieved by checking whether $\mathtt{pc}$ guards the type of the value at the memory location $\mathtt{t}$. Secondly, we also need to ascertain that the assignment does not leak information about the reference that is being written into. Similar to E-DEREF, this is achieved by ensuring that the security annotation $\mathtt{l}$ guards $\mathtt{t}$ as well. Finally, we have the rule E-LET for ML-styled let-polymorphism. Note that $\mathtt{e_1}$ can now be assigned a polytype or a type scheme (represented by $\mathtt{s}$) which enables it to be used with different types within $\mathtt{e_2}$.

3.2.2 Subtyping and Information Flow

There is a natural relationship between subtyping and information flow. In subtyping, if a type $\mathtt{t}$ is expected in an expression, we can use another type $\mathtt{t}'$ provided $\mathtt{t}'$ is a subtype of $\mathtt{t}$. Similarly, in the case of information flow, we can always use a value with a security level $\mathtt{l}$ if the enforced security level in the expression is higher than $\mathtt{l}$. Indeed, subtyping paints a directed view of the program's information flow and has been extensively used in existing approaches to analyse information flow [52, 87, 120].

Introduction of subtyping in the type system presented in figure 3.2, or for any full-fledged type system, is fairly straightforward. For two base types $\mathtt{t}'$ and $\mathtt{t}$ annotated with security levels $\mathtt{l}_{\mathtt{t}'}$ and $\mathtt{l}_{\mathtt{t}}$ respectively, we say $\mathtt{t}'$ is subtype of $\mathtt{t}$ (written $\mathtt{t}' \leq \mathtt{t}$) if $\mathtt{l}_{\mathtt{t}'} \leq \mathtt{l}_{\mathtt{t}}$. In other words, for base types the security annotation is covariant i.e. a base type with a lower security level can be applied at all places that accept a base type with a comparatively higher security level. For establishing subtyping between constructed types, a common approach is to decompose the comparison down to arguments of the type constructor. For this purpose, it is necessary that the types being compared are structurally similar (commonly referred to as *structural subtyping*). The axioms in figure 3.3 show how the subtyping relation is decided for base types as well as constructed types, along with the rule for subtyping denoted by E-Sub. These axioms take into account the variance of the type argument(s), if any. Here, $\oplus$, $\odot$ and $\ominus$ denote covariance, invariance and contravariance, respectively. Thus, subtyping extends the partial order that normally exists amongst security levels. Here, $\mathtt{b}$ represented a base type, $\mathtt{ref}$ represents a memory reference type and the $\rightarrow$ represents a function type where the l.h.s of the $\rightarrow$ is the function's input and r.h.s. represents the function's output. The subtyping on base types is interpreted as follows. Since the base type is annotated with a covariant sign ($\oplus$), any base type $\mathtt{b}$ that carries an annotation $\mathtt{l}_1$ is a subtype of the same base type annotated with $\mathtt{l}_2$ if $\mathtt{l}_1 \leq \mathtt{l}_2$. The interpretation extends similarly to reference types and function types too.

$$\frac{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{e} : \mathtt{t}' \qquad \mathtt{t}' \leq \mathtt{t}}{\mathtt{pc}, \Gamma, \mathtt{M} \vdash \mathtt{e} : \mathtt{t}} \text{ (E-Sub)}$$

$$\mathtt{b}^{\oplus} \qquad \odot\,\mathtt{ref}^{\oplus} \qquad (\ominus \xrightarrow{\oplus} \oplus)^{\ominus}$$

Figure 3.3: Subtyping rule and variance on base types, references and function types

3.3 Constraint-based IFA

Having presented typechecking rules that can perform IFA in section 3.2, we now present a set of constraint-based typing rules for deriving privilege levels for unannotated expressions. The motivation behind a constraint-based typing and information flow analysis framework is that it lends itself well to inference of both types and security annotations. Consequently, the programmer does not need to declare type and security annotations for every expression.

We first introduce a first order logic for constraints and then show how an instance of this logic can be applied to information flow analysis using types. The material presented here is based on the constraint-based information flow analysis framework presented by Simonet et. al. in [101] and [87]. Terms and formulae in the logic are interpreted in the ground algebra detailed in section 3.2.

3.3.1 The language of constraints

Terms in the first-order logic for constraints are denoted in the grammar as τ as shown in the equation Terms in figure 3.4. They are either variables or type terms composed from a constructor. Term variables are interpreted by assignments ρ that map variables to ground terms. In addition to variables and type terms, hand-sides (denoted as ϕ in figure 3.4) are also a part of the terms. Hand-sides are either a term variable or an atomic constant and are useful in introducing atomic constants as a constraint on type terms. Such constraints are useful when the programmer wants to explicitly coerce the privilege level of a value or expression through language-level security annotations.

$$\tau ::= \alpha \mid \mathtt{c}(\tau, \cdots, \tau) \qquad \phi ::= \alpha \mid \mathtt{a} \tag{Terms}$$

$$\begin{aligned} \Gamma ::= {} & \langle \tau = \cdots = \tau \rangle \approx \cdots \approx \langle \tau = \cdots = \tau \rangle \mid \alpha \leq \alpha \mid \phi \sqsubset \phi \mid \\ & \mathtt{true} \mid \mathtt{false} \mid \mathtt{C} \wedge \mathtt{C} \mid \exists \alpha . \mathtt{C} \end{aligned} \tag{Constraints}$$

Figure 3.4: Grammar for terms and constraints

Constraints are predicates that take terms as inputs. They form the formulae in the logic. Equation Constraints in figure 3.4 describes the grammar for constraints. Constraints on values and structure of terms have two facets: structural equality between two terms (represented by the $\approx$ symbol) and equality of two terms (represented using the $=$ symbol). Restrictions on the range of substitutions for a term variable can also be specified in two ways: a strong subtyping constraint (denoted as $\leq$) which is between term variables and a weak subtyping constraint (denoted as $\sqsubset$) between term variables and hand-sides. Atomic constants can be introduced using weak subtyping constraints on term variables. Conjunction of constraints is an obvious choice to make the constraints language more expressive. Existential quantification is useful for introduction of variables by either the type checker or the solver. Its use will become evident shortly as we discuss constraint based typing rules.

$$\frac{}{\rho \vdash \mathbf{true}}\ (\text{I-True}) \qquad \frac{\begin{array}{c}\forall \tau, \tau' \in \overline{\overline{\tau}}_{\mathtt{1}} \cup \cdots \cup \overline{\overline{\tau}}_{\mathtt{n}}\ \ \rho(\tau) \approx \rho(\tau') \\ \forall \mathtt{i} \in [\mathtt{1}, \mathtt{n}]\, \forall \tau, \tau' \in \overline{\overline{\tau}}_{\mathtt{i}}\ \ \rho(\tau) = \rho(\tau')\end{array}}{\rho \vdash \overline{\overline{\tau}}_{\mathtt{1}} \approx \cdots \approx \overline{\overline{\tau}}_{\mathtt{n}}}\ (\text{I-Structure})$$

$$\frac{\rho(\alpha_1) \leq \rho(\alpha_2)}{\rho \vdash \alpha_1 \leq \alpha_2}\ (\text{I-Strongleq}) \qquad \frac{\rho(\phi_1) \sqsubset \rho(\phi_2)}{\rho \vdash \phi_1 \leq \phi_2}\ (\text{I-Weakleq})$$

$$\frac{\rho \vdash \mathtt{C}_1 \qquad \rho \vdash \mathtt{C}_2}{\rho \vdash \mathtt{C}_1 \wedge \mathtt{C}_2}\ (\text{I-Conjunction}) \qquad \frac{\rho' \vdash \mathtt{C} \qquad \rho' = \rho[\alpha \to \star]}{\rho \vdash \exists \alpha.\mathtt{C}}\ (\text{I-Existential})$$

Figure 3.5: Interpretation of constraints

Interpretation of constraints is described in figure 3.5. Constraints are interpreted in the ground algebra using a two place predicate $\cdot \vdash \cdot$. The first and second arguments to the predicate are an assignment and a constraint respectively. The rule for interpreting structural constraints is described in the rule I-Structure. In this rule we interpret a multi-skeleton which is a collection of structurally similar multi-equations. Each multi-equation is written as τ_i in short where i can range from `1` to `n`. Here, ρ is an assignment mapping variables in a type to ground terms. Multi-equations are sets of term variables that have the same interpretation and hence represent the same type. According to rule I-Structure, two terms can belong to the same multi-equation only if they have the same interpretation in the ground algebra. Two multi-equations belong to the same multi-skeleton if their interpretations in the ground algebra have the same structure. Constraints on the range of values a term variable can take are dictated by strong and weak subtyping constraints. The interpretation of these constraints is described in rules I-Strongleq and I-Weakleq. Both these rules ensure that given an assignment ρ, a term variable (or hand-side) can be constrained by another term variable (or hand-side) if the interpretation of the former under ρ is a strong subtype (or weak subtype) of the latter. The interpretation rule for conjunction of constraints is straightforward and described in I-CONJUNCTION. The interpretation of an existential constraint is described in rule I-EXISTENTIAL.

3.3.2 Type Schemes

In the traditional Damas-Milner type system, a type scheme $\mathtt{s}$ is of the form $\forall\overline{\alpha}.\mathtt{t}$ where the set of type variables $\overline{\alpha}$ are considered bound within $\mathtt{t}$. Damas-Milner styled polymorphism is described in figure 3.6. If a type environment Γ asserts that a term $\mathtt{e}$ has a type $\mathtt{t}$, then all type variables that do not belong to the free variables of Γ are universally quantified in the form of a type scheme. This generalisation in the Damas-Milner type system is shown in the rule DM-GEN in figure 3.6. Whenever the scheme is initialised as shown in the rule DM-INST, the universally quantified variables are replaced with concrete types.

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{t} \qquad \overline{\alpha} \# \mathtt{ftv}(\Gamma)}{\Gamma \vdash \mathtt{e} : \forall\overline{\alpha}.\mathtt{t}} \text{ (DM-GEN)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \forall\overline{\alpha}.\mathtt{t}}{\Gamma \vdash \mathtt{e} : [\overline{\alpha} \mapsto \overline{\mathtt{t}}]\mathtt{t}} \text{ (DM-INST)}$$

Figure 3.6: Polymorphism in the Damas-Milner type system

In constraint based typing (and consequently, information flow analysis), a typing judgement is typically a four place predicate whose parameters are a constraint $\mathtt{C}$, a type environment Γ, a term $\mathtt{e}$ and a scheme σ. A judgement is written as $\mathtt{C}, \Gamma \vdash \mathtt{e} : \sigma$ and is read as: under the assumptions about the free type variables of the judgement which is recorded in $\mathtt{C}$ and the types of the free program identifiers which is recorded in Γ, the term $\mathtt{e}$ has a type σ.

In order to better appreciate the nature of polymorphism in constraint-based information flow analysis, we describe the rules for generalisation and instantiation of types in figure 3.7. CT-GEN describes the generalisation rule. For the constraint based typing rules, we assume that a type $\mathtt{t}$ not containing any bound variables can also be viewed as the type scheme $\forall\phi[\mathtt{True}].\mathtt{t}$. In CT-GEN, $\mathtt{C}$ is the set of constraints that does not concern the type variables that are being generalised, and $\mathtt{D}$ is the set of those constraints which concern these type variables. It is to be noted that type variables which occur free in Γ may also appear free in $\mathtt{D}$ in addition to $\overline{\alpha}$. The second premise of CT-GEN is similar to

$$\frac{\mathtt{C} \wedge \mathtt{D}, \Gamma \vdash \mathtt{e} : \mathtt{t} \qquad \overline{\alpha} \# \mathtt{ftv}(\mathtt{C}, \Gamma)}{\mathtt{C} \wedge \exists \overline{\alpha}.\mathtt{D}, \Gamma \vdash \mathtt{e} : \forall \overline{\alpha}[\mathtt{D}].\mathtt{t}} \text{ (CT-GEN)}$$

$$\frac{\mathtt{C}, \Gamma \vdash \mathtt{e} : \forall \overline{\alpha}[\mathtt{D}].\mathtt{t}}{\mathtt{C} \wedge \mathtt{D}, \Gamma \vdash \mathtt{e} : \mathtt{t}} \text{ (CT-INST)}$$

Figure 3.7: Polymorphism in a constraint-based type system

DM-GEN. Conclusion of the rule generalises all the type variables $\overline{\alpha}$ but the generalisation is subject to the constraint D. This is reflected by augmenting C with the existential constraint $\exists \overline{\alpha}.\mathtt{D}$ in the conclusion of the rule. Instantiation is described in the rule CT-INST. It is important to note here that in the conclusion of the rule, C is not augmented with D without regard for the fact that C may already contain $\exists \overline{\alpha}.\mathtt{D}$. This is necessary because we may now substitute the bound variables $\overline{\alpha}$ with concrete types which may render D unsatisfiable even though $\exists \overline{\alpha}.\mathtt{D}$ may be satisfiable. For example, consider the set of natural numbers. While the existential term $\exists \mathtt{x} \in \mathtt{N}.(\mathtt{x} > 2)$ is true since there are many natural numbers greater than 2, $[1/\mathtt{x}].(\mathtt{x} > 2)$ is not, where $[1/\mathtt{x}]$ denotes substituting x with 1.

3.3.3 Constraint-based typing

We are now ready to described constraint-based typing rules that cater to information flow analysis in addition to typing terms in the calculus. The grammar of types described in figure 3.1 is now replaced with one that contains variables for type, privilege levels and the program counter. The new grammar is described in figure 3.8. The program counter and privilege levels are represented by the meta-variables π and λ respectively, while annotated types are represented by the meta-variable τ. Level variables are represented as δ and type variables are represented as β.

$$\tau ::= \beta \,|\, \mathtt{unit} \,|\, \mathtt{b}^{\lambda} \,|\, \tau\, \mathtt{ref}^{\lambda} \,|\, (\tau \xrightarrow{\pi} \tau)^{\lambda}$$

$$\pi, \lambda ::= \delta \,|\, \mathtt{l}$$

Figure 3.8: Meta-variables representing types and levels

We now present a set of constraint based rules for conducting type-based information flow analysis. The rules are described in figure 3.9 and are analogous to those described in figure 3.2. In these rules, variables of arbitrary kind are represented with the variable α; a set of such variables is represented by $\overline{\alpha}$. Every judgement begins with a constraint $\mathtt{C}$ which contains assumptions about free type and level variables. The constraint must be satisfiable for the judgement to be valid. For simpicity, we no longer have the map M of references to their types in the typing rules; the set of constraints C on pure expressions can be easily extended with the map to prove soundness in the presence of side-effects through reference manipulation as shown in [90]. Another important difference lies in the manner in which universal quantification is introduced in rule E-Let-C and how it is eliminated in the rule V-Var-C. Unlike the traditional Damas-Milner type system, we have constraints on universally quantified variables as described in section 3.3.2.

$$\frac{\Gamma(\mathtt{x}) = \forall\overline{\alpha}[\mathtt{D}].\tau \qquad \mathtt{C} \Vdash \exists\overline{\alpha}.\mathtt{D}}{\mathtt{C} \wedge \mathtt{D}, \Gamma \vdash \mathtt{x} : \tau} \text{ (V-VAR-C)}$$

$$\frac{\mathtt{C}, \pi, \Gamma[\mathtt{x} \mapsto \tau'][\mathtt{f} \mapsto (\tau' \xrightarrow{\pi} \tau)^{\lambda}] \vdash \mathtt{e} : \tau}{\mathtt{C}, \Gamma \vdash \mathtt{fix}\,\mathtt{f}.\lambda\mathtt{x}.\mathtt{e} : (\tau' \xrightarrow{\pi} \tau)^{\lambda}} \text{ (V-ABS-C)}$$

$$\frac{\mathtt{C}, \Gamma \vdash \mathtt{v}_1 : (\tau' \xrightarrow{\pi \sqcup \lambda} \tau)^{\lambda} \qquad \mathtt{C}, \Gamma \vdash \mathtt{v}_2 : \tau' \qquad \mathtt{C} \Vdash \lambda \lhd \tau}{\mathtt{C}, \pi, \Gamma \vdash \mathtt{v}_1\mathtt{v}_2 : \tau} \text{ (E-APP-C)}$$

$$\frac{\mathtt{C}, \Gamma \vdash \mathtt{v} : \tau \qquad \mathtt{C} \Vdash \pi \lhd \tau}{\mathtt{C}, \pi, \Gamma \vdash \mathtt{ref}\,\mathtt{v} : \tau\,\mathtt{ref}^{*}} \text{ (E-REF-C)}$$

$$\frac{\mathtt{C}, \Gamma \vdash \mathtt{v} : \tau\,\mathtt{ref}^{\lambda} \qquad \mathtt{C} \Vdash \lambda \lhd \tau}{\mathtt{C}, \pi, \Gamma \vdash !\mathtt{v} : \tau} \text{ (E-DEREF-C)}$$

$$\frac{\mathtt{C}, \Gamma \vdash \mathtt{v}_1 : \tau\,\mathtt{ref}^{\lambda} \qquad \mathtt{C}, \Gamma \vdash \mathtt{v}_2 : \tau \qquad \mathtt{C} \Vdash \{\pi \sqcup \lambda\} \lhd \tau}{\mathtt{C}, \pi, \Gamma \vdash \mathtt{v}_1 := \mathtt{v}_2 : \mathtt{unit}} \text{ (E-ASSIGN-C)}$$

$$\frac{\mathtt{C} \wedge \mathtt{D},\, \Gamma \vdash \mathtt{e}_1 : \tau' \qquad \overline{\alpha}\#\mathtt{ftv}(\mathtt{C}, \Gamma) \qquad \mathtt{C}, \pi, \Gamma[\mathtt{x} \mapsto \forall\overline{\alpha}[\mathtt{D}].\tau'] \vdash \mathtt{e}_2 : \tau}{\mathtt{C} \wedge \exists\overline{\alpha}.\mathtt{D}, \pi, \Gamma \vdash \mathtt{let}\,\mathtt{x} = \mathtt{e}_1\,\mathtt{in}\,\mathtt{e}_2 : \tau} \text{ (E-LET-C)}$$

Figure 3.9: Constraint based typing for IFA

3.4 Constraint rewriting and solving

Having described constraint based typing rules in section 3.3, we now describe the methodology for solving the constraints. The techniques for constraint solving described in this section are a succinct version of those described in [101] and [90]. First, we describe how constraints can be rewritten in a simplified form in section 3.4.1. Then, we describe the procedure for solving the simplified constraints in section 3.4.2.

3.4.1 Constraint expansion and decomposition

In this research work we assume that the subtyping relationship is structural in nature i.e. if a ground type (one that does not contain any free variables) or a variable representing a ground type is a subtype of another ground type or of a variable, then the two types/variables must have the same type constructor and hence the same structure. As mentioned in section 3.3.1, structural similarity is represented using the operator $\approx$ where $\alpha \approx \beta$ means that α and β have the same structure. For two types that are the same, we use the equality symbol (=).

Definition 2. *A constructed type is a type that is constructed from at least two other types or type variables using type constructors. A type variable representing a constructed type is called a constructed type variable.*

Definition 3. *A terminal is a type variable that does not represent a constructed type, and hence cannot be expressed as a combination of structurally simpler types or variables. A constraint is atomic if it involves only terminals.*

We now present techniques to simplify the constraints carried by the type of an expression down to atomic constraints i.e. constraints between variables that have been completely decomposed to remove any type constructor. There are two phases in this exercise which have been highlighted in figure 3.10. The Expansion rule highlights the first phase which shows how to expand constructed type variables.

The Expansion rule in figure 3.10 expands a constructed type variable to reflect the structural knowledge that we have about this variable; structural constraints involving multi-equations are expanded to conjunction of constraints between terminals. Here, $\overset{=}{\alpha}$ is a variable representing a multi-equation and $\mathtt{v_i(f)}$ represents the variance of the $\mathtt{i}^{th}$ argument in the type constructor `f`. Since this dissertation is about the structural subtyping discipline, knowledge about the structure of the constructed type variable can be derived from other structurally equivalent types that bound the variable. However, it must be

$$\xmapsto[\exists\tilde{\alpha}.[\langle\overline{\overline{\alpha}} = \mathtt{f}(\tilde{\alpha})\rangle \approx \langle\overline{\overline{\beta}} = \mathtt{f}(\tilde{\beta})\rangle \wedge_{\mathtt{i}} \alpha_{\mathtt{i}} \approx^{\mathtt{v_i(f)}} \beta_{\mathtt{i}}]]{\langle\overline{\overline{\alpha}}\rangle \approx \langle\overline{\overline{\beta}} = \mathtt{f}(\tilde{\beta})\rangle} \qquad \text{Expansion}$$

$$\begin{matrix} \alpha = \mathtt{f}(\tilde{\alpha}) \\ \beta = \mathtt{f}(\tilde{\beta}) \end{matrix} \xmapsto[\wedge_{\mathtt{i}}\alpha_{\mathtt{i}} \leq^{\mathtt{v_i(f)}} \beta_{\mathtt{i}}]]{\alpha \leq \beta} \qquad \text{Decomposition}$$

Figure 3.10: Rewriting constraints on type terms

noted that this does not mean that the arguments to the expanded type constructors are themselves terminal. For example, if we have a product type $\gamma \times \delta$, it could very well be the case that γ and δ are constructed types and there are further constraints on them after the applying the Expansion rule. In such a case, the expansion process repeats itself until we have constraints signifying structural equivalence of terminals.

Expansion gives us a set of constraints between constructed types that are themselves expressed using terminals as building blocks. Decomposition is the second step towards atomic constraints and operates on the results of expansion. In decomposition, we match terminals from the lhs and rhs of constraints representing structural equivalence (denoted by $\approx$). For example, if we have a constraint which says $\langle\omega_1 = \gamma_1 \times \delta_1\rangle \approx \langle\omega_2 = \gamma_2 \times \delta_2\rangle$ and we know through the typing rules that $\omega\mathtt{1} \leq \omega\mathtt{2}$, the Decomposition rule in figure 3.10 derives relationship between the terminals which are $\gamma_1 \leq \gamma_2$ and $\delta_1 \leq \delta_2$. It is to be noted that the exact subtyping relation between the terminals is governed by their variance.

3.4.2 Constraint solving

We now describe the technique to check solvability of atomic constraints. Figure 3.11 describes an arbitrary flow constraint graph which is bounded at the inputs and outputs with concrete privilege levels. For the sake of simplicity, we assume the absence of side-effects in this example. Hence, there is no mention of the variable *pc*. However, this simplification does not preclude integration of side-effects.

The privilege levels are represented as represented as variables $\alpha_1 \cdots \alpha_7$ in the flow constraint graph shown in figure 3.11. α_1 and α_2 are input variables in the flow constraint graph and α_6 and α_7 are output variables. The input variables α_1 and α_2 are at a privilege level of l_1 and l_2 respectively. The output variables at α_6 and α_7 are written back at levels l_3 and l_4 respectively.

The first step in solving atomic constraints is to determine bounds for level variables.

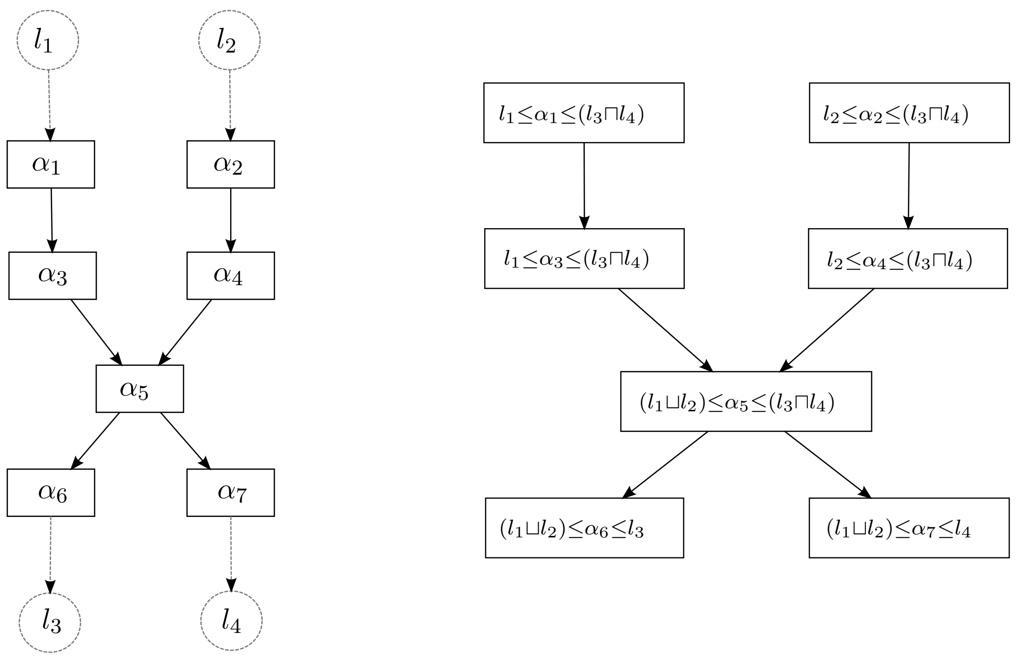

l_1
l_2
α_1
α_2
α_3
α_4
α_5
α_6
α_7
l_3
l_4
$l_1 \leq \alpha_1 \leq (l_3 \sqcap l_4)$
$l_2 \leq \alpha_2 \leq (l_3 \sqcap l_4)$
$l_1 \leq \alpha_3 \leq (l_3 \sqcap l_4)$
$l_2 \leq \alpha_4 \leq (l_3 \sqcap l_4)$
$(l_1 \sqcup l_2) \leq \alpha_5 \leq (l_3 \sqcap l_4)$
$(l_1 \sqcup l_2) \leq \alpha_6 \leq l_3$
$(l_1 \sqcup l_2) \leq \alpha_7 \leq l_4$

Figure 3.11: Atomic constraints

It is important to note that a level variable can have multiple lower and upper bounds depending on the connections in the flow constraint graph. For example, the lower bound for level variable α_5 is a combination of both l_1 and l_2, since α_5 is constrainted by both α_1 and α_2. Therefore, α_5 inherits the lower bounds of both α_1 and α_2. Since, l_1 and l_2 represent discrete elements in a security lattice, a stronger lower limit on α_5 is the least upper bound of l_1 and l_2 which is written as $l_1 \sqcup l_2$. Similar to the lower bound for α_5, the lower bounds for other level variables can be obtained using a forward topological walk of the flow constraint graph and dual to this, a reverse topological walk can derive the upper bounds for the level variables. If a level variable inherits multiple privileges levels as upper bounds, the levels can be unified using the greatest lower bound operator ($\sqcap$).

Solvability involves checking for constraints on level variables that are inconsistent with the security lattice. After the bounds on all level variables have been obtained using a forward and reverse topological walk, we check whether all lower bounds of each variable are at a lower privilege level than the upper bounds on that variable. This is achieved by verifying the relationship between the lower and upper bounds against the security lattice. If the relationship holds true i.e. all lower bounds on a variable are lower than the upper bounds on it, then the constraints are solvable. A valid substitution for the variables could then be the least upper bound of all lower bounds for the variable or the greatest lower bound of all upper bounds on the variable.

3.5 Simplification of type schemes

Efficiency of the solving process depends on the size of the constraint set at hand. Expansion as described in section 3.4.1 introduces a lot of new variables. For the constraint solving to be efficient, it is imperative to remove variables that cannot contribute to the correctness of information flow analysis. A number of optimisations have been proposed to reduce the size of the constraint graph in the literature [85, 101]. Of these, we discuss two important ones that contribute significantly to reducing the size of the constraints : *chain reduction* and *polarised garbage collection*. Together, the two have been shown to eliminate over 90% of all reducible constraints.

Chain reduction involves fusing a term variable into its unique bound. For example, if we have a multi-skeleton of the form $\langle\overline{\overline{\alpha}}\rangle \approx \langle\overline{\overline{\tau}}\rangle \approx \overset{\approx}{\tau}$ (here, $\overset{\approx}{\tau}$ represents a multi-skeleton) and this is the only multi-skeleton involving $\langle\overline{\overline{\alpha}}\rangle$, we can rewrite the multi-skeleton as $\langle\overline{\overline{\alpha}} = \overline{\overline{\tau}}\rangle \approx \overset{\approx}{\tau}$ because $\overline{\overline{\alpha}}$ is just a placeholder for a multi-equation. Chain reduction reduces the number of distinct multi-equations that are expanded into atomic constraints. Hence, it achieves a reduction in the size of the graph representing the atomic constraints, and consequently optimises the constraint solving process.

Polarised garbage collection is the other optimisation that is useful for reducing the size of the atomic constraints graph. It applies to label-polymorphic expressions and involves compacting the atomic constraint graph for these expressions. Computing the type scheme for a label-polymorphic expression typically yields a lot of intermediate variables. While they are essential during the generation of the scheme, they are not needed for information flow analysis once the scheme has been generated. Hence, one can eliminate these intermediate variables without affecting the correctness of constraint solving.

Consider the label-polymorphic expression `ternary_op` in figure 3.12. This expression takes in 3 inputs: `x`, `y` and `z`. Until the polymorphic expression is applied to values that are annotated with privilege levels in (ll. 6-7), it is not possible to tell the constant lower bounds on the input variables. Similarly, until the result is written back (ll. 8), it is not possible to know an upper bound on the output of `ternary_op`. Thus, at every instance the label-polymorphic expression is applied, all constraints on all universally quantified variables have to be copied as a part of the type scheme. In reality, however, all that needs to be copied are the constraint relationships between input and output variables. This is sufficient to verify whether a privilege violation at the inputs percolates down to the

```
let foo =
let ternary_op =
  fun x y z ->
    let r = op x y z in r
in
let p = ternary_op A^a B^b C^c
and q = ternary_op D^d E^e F^f
in [p; q]^h
```

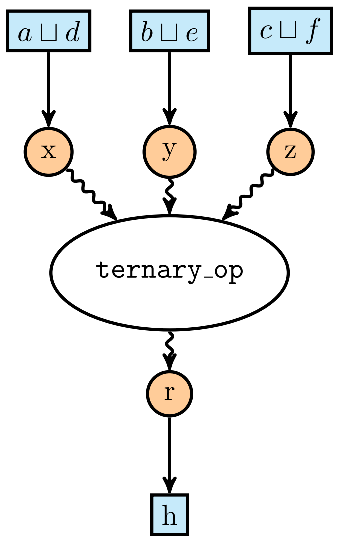


Figure 3.12: A label-polymorphic expression and its constraints graph

output. For example, we know from the accompanying constraints graph for `ternary_op` in figure 3.12 that input variable `x` influences the value of output variable `r`. Therefore, in order to enforce secure information flow between `x` and `r`, it is sufficient to check whether the relationship between the lower bound for `x` (i.e. $a \sqcup d$) and upper bound for `r` (i.e. h) obeys the security lattice.

Polarised garbage collection is a means to compact the atomic constraints graph in such a manner that intermediate type variables and constraints on them are not copied whenever a label-polymorphic scheme is initialised. In the first step, a transitive closure operation on the atomic constraints graph is performed. In the second step, only those relationships that involve an input variable and an output variable are preserved as part of the scheme.

3.6 Computational costs

In this section, we discuss the computational costs of both constraint solving and simplification. We first discuss the cost of constraint generation, expansion, decomposition and resolution as described in sections 3.3. Then we discuss the cost of the optimisations that are discussed in section 3.5.

3.6.1 Constraint Generation and Solving

Since constraints are derived from terms in the grammar of the programming language, the initial size of the set of constraints `C` is proportional to the size of the program under consideration. Let this size be represented by n_c. The first step in solving the generated constraints is unification - both at the level of structural similarity (skeletons) and at the level of equality (equations). The unification process is straightforward and similar to the standard procedure for type unification. It is described in further detail in [101]. The cost of unification is $O(n_c\alpha(n_c))$ where $\alpha(\cdots)$ is the inverse Ackermann function. The inverse Ackermann function is a very slow growing function and the computational complexity can be considered linear for all practical purposes.

After unifying type terms based on their structure and value into multi-skeletons and multi-equations, the next step is rewriting constraints in an atomic form as described in section 3.4.1. If we assume that a represents the arity of type constructors and h to be the maximum size of a multi-skeleton, then expansion and decomposition introduce a^h new variables for each of the variables in the input constraint. Thus, the two steps of expansion and decomposition have a cost of $O(a^h n_c)$.

Once atomic constraints have been obtained post expansion and decomposition, we represent them in the form of a graph which we call the *atomic constraints graph*. Representation of atomic constraints in the form of a graph aids in the resolution of those constraints. Constraints are resolved by walking through the graph derived from the atomic constraints. During this walk, we need to verify whether the upper and lower bounds for each terminal violate the pre-specified security policy. As already mentioned, this security policy is specified as an ordering amongst the atoms, which is described in the form of a security lattice. Each terminal is represented as a vertex in the graph and the subtyping relationship between terminals is represented as an edge in the graph. It is to be mentioned here that a terminal may have multiple upper and lower bounds depending on what its ancestors and descendants are in the atomic constraints graph, as described in section 3.4.2. In such a case, we need to be able to compute the join and meet of multiple elements in the lattice to verify whether bounds of each terminal respect the security policy. Satisfiability of atomic constraints can be checked through a topological walk of the atomic constraints graph. And, substituting each of the terminals with either its lower or upper bound yields a suitable solution. Assuming that these lattice operations can be computed in constant time, resolution of atomic constraints can be thus achieved in $O(a^h n_c + m)$ where m is the number of atomic constraints.

3.6.2 Scheme simplification

Chain reduction can be achieved by inspecting individual term variables and, if there exists a unique upper or lower bound for these variables, then equating the variables with the bound and storing the result as a multi-equation. Its cost is therefore directly proportional to the number of skeletons in the input constraint set which in turn proportional to the size of the program. The cost of chain reduction is therefore $O(n_c)$.

Polarised garbage collection on the other hand involves deriving flow dependencies between input and output variables. It only preserves flow relationships between input and output variables, and does away with all intermediate variables in a label-polymorphic expression. This involved performing a full transitive closure of the atomic constraints graph to derive which input variables influence what output variables. Thus, a naive implementation of polarised garbage collection has a complexity of $O((a^h n_c)^3)$ because transitive closure of a directed graph can be performed in cubic time.

3.7 Algorithmic bottlenecks

In the previous section, we outlined the computational costs for solving and simplifying atomic constraints. In this section, we discuss the bottlenecks in the simplification and resolution of information flow constraints and propose means to address them.

3.7.1 Bottlenecks in IFA

In section 3.6, it was mentioned that atomic constraints can be solved through a topological walk of the graph, provided the lattice queries can be answered in constant time. However, the cost of pre-processing the lattice to achieve constant time queries was ignored. In reality, answering lattice queries in constant time requires heavy pre-processing and its computational costs cannot be ignored. Answering lattice queries in constant time necessitates computing a full transitive closure of the lattice to establish ancestor-descendant relationships as a first step. Then, for all pair of vertices, it is necessary to identify a common ancestor (descendant) that has the highest (lowest) topological number for the $\sqcup$ ($\sqcap$) query.

Similarly, the costs for polarised garbage collection tend to be higher than other steps in IFA. This is once again due to its dependence on the transitive closure operation; identifying which input variables go on to influence what output variables requires computing a transitive closure of the atomic constraints graph which is nothing but a DAG. Thus, simplification of atomic constraints and answering lattice queries tend to be the bottlenecks in IFA due to their dependence on transitive closure of DAGs as a starting point.

3.7.2 Reduction of TC to BMP

It is common knowledge that transitive closure of DAGs can be achieved through matrix multiplication. In this section, we discuss how the transitive closure of DAGs can be reduced to the problem of multiplying boolean matrices. This paves the way for presenting the cost of simplifying and resolving IFA constraints in terms of the Boolean Matrix Product (BMP).

Let $G = (V, E)$ represent a DAG where V is the set of vertices in G and E is the set of directed edges in G. Let $G^* = (V, E^*)$ represent the transitive closure of G. Here, E^* represents paths between all pairs of vertices that are connected in G through a sequence of one or more edges. Assuming the vertices of the DAG are topologically sorted, the adjacency matrix (when the vertices are written out in topological order) is an upper triangular matrix. Such an adjacency matrix is represented in figure 3.13a for a DAG G. Here, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are submatrices of the adjacency matrix for G while $\mathbf{0}$ is a submatrix whose elements are zeroes. $\mathbf{A}$ and $\mathbf{B}$ are themselves upper triangular matrices. A clearer understanding of the significance of the submatrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ comes from figure 3.13c where the vertices are written out left to right in topological order. If we have an edge connecting vertices in the first half of the topological order, we capture the adjacency information in the submatrix $\mathbf{A}$. Any edge connecting vertices in the second half of the order is captured in submatrix $\mathbf{B}$. On top of this, edges going across the two halves are captures in submatrix $\mathbf{C}$.

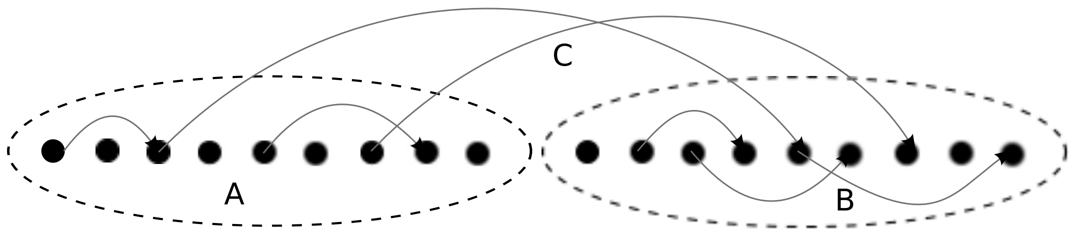
The next objective is to compute the reachability matrix for graph G. The reachability matrix is shown in figure 3.13b. Transitive closure for vertices in the first half of the topological order can be computed independent to those in the second half if we ignore any interconnections across the two sets. Therefore, submatrices $\mathbf{A}^*$ and $\mathbf{B}^*$ (which represent transitive closures of $\mathbf{A}$ and $\mathbf{B}$ respectively) in figure 3.13b can be computed by performing a transitive closure operation on submatrices $\mathbf{A}$ and $\mathbf{B}$. Transitive closure across the two halves in the topological ordering can be obtained by performing a boolean matrix product of $\mathbf{A}^*$, $\mathbf{C}$ and $\mathbf{B}^*$. If we assume that the $T(n)$ is the time necessary to compute G^* from G where n is the number of vertices in G, then computing $\mathbf{A}^*$ and $\mathbf{B}^*$ takes $T(n/2)$ time. Computing $\mathbf{A}^*\mathbf{C}\mathbf{B}^*$ can be done in $O((n/2)^\omega)$ where ω ($\omega \sim 2.3$) is the exponent of the fastest matrix multiplication algorithm [33, 115, 34]. Thus, we have the relation $T(n) = T(n/2) + O((n/2)^\omega)$. Using the master theorem for complexity of algorithms, the

A	**C**
0	**B**

(a) G

$\mathbf{A}^*$	$\mathbf{A}^*\mathbf{C}\mathbf{B}^*$
$\mathbf{0}$	$\mathbf{B}^*$

(b) G^*



(c) Vertices in topological order

Figure 3.13: Transitive Closure of DAGs

right hand side of the relation simplifies to $O(n^\omega)$. Thus, transitive closure of DAG with n vertices can be reduced to the problem of computing boolean matrix product on two $n \times n$ matrices.

3.7.3 BMP as a basis function

In section 3.6, we identified two bottlenecks in type-based IFA that uses TC for DAGs. These were identified to be simplification of schemes for label-polymorphic expressions using polarised garbage collection, and pre-processing lattices for answering lattice queries. Having shown that computing TC for DAGs can be reduced to the problem of multiplying boolean matrices, we now present the computational costs for type-based IFA using BMP as a basis function.

Simplification of label-polymorphic expression using polarised garbage collection involves compacting the atomic constraints graph to preserve only those subtyping relationships that exists between input and output variables in the expression. This involves computing a transitive closure of the atomic constraints graph and can be achieved in a time of $O(n^\omega)$.

Pre-processing lattices to answer lattice queries for IFA is a little more involved. A subtyping query on the lattice can easily be answered in constant time after performing a transitive closure of the lattice. If we assume that the lattice has k elements, this can be done in $O(k^\omega)$. However, answering $\sqcup$ and $\sqcap$ queries required us to pick one element from the common ancestors/descendants of the query arguments. Since $\sqcup$ is the dual of $\sqcap$ we discuss the cost for $\sqcup$ only. Once we have a reachability matrix through the transitive closure of the lattice, we can tell all ancestors (and descendants) for all vertices in the DAG representing the lattice.

For a pair of vertices, if we need to calculate $\sqcup$ we need to identify the common ancestors first and then pick one with the highest topological number. An elegant solution for pre-computing $\sqcup$ for all pair of vertices in a DAG has been discussed in [36]. It was shown in [36] that picking $\sqcup$ for all pairs of vertices in a DAG can be reduced to picking maximal witnesses in the boolean matrix product of the reachability matrix of the DAG with itself. The cost for doing this was shown to be $O(k^{(2+\theta)})$ where θ satisfies relation $\omega_{1,\theta,1} = 1 + 2\theta$. Here $\omega_{1,\theta,1}$ is the exponent of k in the operation cost for the multiplication of a $k \times k^{\theta}$ matrix with a $k^{\theta} \times k$ matrix. After obtaining a cost for $\omega_{1,\theta,1}$ in terms of ω (~ 2.3) and θ and solving for θ, it was shown that picking maximal witnesses and hence, identifying $\sqcup$ for all pairs of vertices in a DAG has a cost of $O(k^{2.575})$ [36].

3.8 Summary

In this chapter, we discussed a framework for type-based information flow analysis. We showed how constraints can be used for both type inference and simultaneously information flow analysis. We discussed techniques for solving information flow constraints carried by types, and also discussed strategies for the simplification of schemes for label-polymorphic expressions where the security annotation for the type is unknown. We derived the computational costs for both simplification and resolution of constraints, and showed that for both these cases transitive closure of DAGs is the primary bottleneck when it comes to computational costs. We then discussed how transitive closure of DAGs can be reduced to the problem of matrix multiplication, and showed how the computational costs for simplification and resolution of information flow constraints can be expressed using boolean matrix product as a basis function.

We use the discussion of the bottlenecks in IFA in section 3.7 to motivate the content of chapters 4, 5 and 6 in this dissertation. In these chapters, we design and evaluate structure-sensitive algorithms for answering lattice queries in constant time and simplifying label-polymorphic constraint graphs. The structure-sensitive algorithms proposed in these chapters adapt themselves to the structure of the DAG that is supplied to them, and seamlessly interpolate the performance graph between the best reported algorithms for trees and the best reported algorithms for dense DAGs, depending on the incidence of non-tree edges in the DAG. Thus, they are able to achieve the best performance for every point in the structure-spectrum of lattices and label-constraint graphs. This is especially useful for IFA which often has to deal with lattices that are structurally very diverse.

Chapter 4

Adaptive pre-processing of security lattices

4.1 Introduction

In chapter 3, we discussed the importance of lattice pre-processing in efficiently solving atomic inclusion constraints arising in program analysis that is based on polymorphic subtyping and label constants that are ordered as a lattice. In this chapter, we discuss a novel graph theoretic approach for pre-processing a DAG for answering the LCA query in constant time. Since the lattice is normally represented as a DAG, the results obtained for LCA computation for vertex-pairs in DAGs can be directly applied to obtaining the $\sqcup$ of two elements in a lattice. Additionally, since $\sqcap$ is the dual of the $\sqcup$ query and $\leq$ query on two elements in a lattice is just a special case of the $\sqcup$ query where one of the query elements is the $\sqcup$ of the two query elements, the results obtained in this chapter extend directly to $\sqcap$ and $\leq$ queries as well.

Unlike existing techniques which pre-process the DAG at the level of individual vertices, our graph decomposition lets us pre-process the DAG at the level of sets of vertices without any loss of precision in answering the LCA query for a pair of vertices. We call such a set of vertices a *cluster* which is a set of adjacent vertices with a single point of entry. In addition to uncovering a layer of abstraction in the pre-processing, clusters make the proposed pre-processing algorithm highly adaptive - the computational costs are a direct function of the latent scope for decomposition of the DAG into clusters. Thus, the proposed algorithm is highly suitable for applications which have to deal with lattices that have a wide structural variety.

The rest of the chapter is organized as follows. We define the LCA query in detail in section 4.2 and discuss why pre-processing a DAG to answer LCA queries in constant time is an expensive operation. We give an overview of our approach in section 4.3 where we discuss that any vertex u could reach another v either through the spanning tree covering the DAG or through a combination of spanning tree edges and cross edges. In the former case, we call u a tree ancestor (T) of v and in the latter case, we call u a cross ancestor (C) of v. Thereby, we categorize potential LCAs into one of the four categories - TT-PLCA, CT-PLCA, TC-PLCA and CC-PLCA - corresponding to the type of ancestral relationship between the potential LCA and the query vertices. The vertex with the highest topological number amongst these four PLCAs is the LCA of the query pair. In section 4.3, we discuss that the TT-PLCA can be computed by using the RMQ query on the spanning tree and show that the CT-PLCA need not be computed for a DAG. In section 4.4, we discuss techniques to identify the TC-PLCA for a vertex pair. In section 4.5, we discuss techniques to identify the CC-PLCA and summarise the chapter in section 4.6.

4.2 Lowest Common Ancestor

In this section, we provide a formal definition of the LCA query for DAGs and also discuss the reason why obtaining the LCA of two vertices in a DAG is much more involved that obtaining the LCA for two nodes in a tree.

Definition 4. *The set of Lowest Common Ancestors (LCA) of two vertices* u *and* v *in a DAG is a set of vertices* $L = \{\mathtt{l}_1, \mathtt{l}_2 \cdots, \mathtt{l}_n\}$ *such that all vertices in L are common ancestors of* u *and* v *and no other descendant of the vertices in L is an ancestor of* u *and* v *[64].*

A tree is a special case of a DAG; there is a unique LCA for all vertex-pairs in a tree. But vertex pairs in arbitrary DAGs may have multiple LCAs. In such a setting, a *representative LCA* is typically selected from the set of vertices satisfying the LCA properties. The initial approach to picking a representative LCA in the literature was to use the notion of depth of a vertex in the DAG [19, 20]. The depth was defined to be the longest hop distance of a vertex from the source of the DAG. In such a setting, it was possible for multiple vertices to have the same depth and ties were resolved arbitrarily to ensure that no two vertices have the same depth. Thus, it was possible to obtain a unique representative LCA for all vertex pairs. In later approaches [63, 36], a simpler approach was used to assigning depth values to vertices through topological ordering. In these works, the reachability matrix for the DAG was sorted according to the topological numbers of vertices and the representative LCA was defined to be the maximal witness of the boolean matrix product of the reachability matrix and its transpose. In other words, the representative LCA for a vertex-pair was nothing but the vertex with the highest topological number amongst the common ancestors of the vertex-pair. Similar to [63, 36], we define the representative LCA to be the vertex that has the highest topological number in the set of common ancestors of the vertex-pair in this chapter.

The regular structure of trees and the unique LCA of vertex-pairs in trees make computation of LCA in trees relatively easier as compared to other kinds of graphs. LCA computation in trees can be computed in linear time using the *range minimum query* (RMQ) technique [46]. On the other hand, a rooted DAG contains an overlay of forward and cross edges on top of the edges of a spanning tree that covers the DAG. This additional layer of complexity inhibits the applicability of the simple and elegant RMQ technique to the case of DAGs. As a result, all of the reported techniques in the literature have resorted to computing the transitive closure of the entire DAG as a first step towards computing the LCA. By computing the closure, it is possible to easily identify the ancestors of vertices and consequently, the respresentative LCA for any given vertex-pair.

Computing the closure of a DAG is a computationally expensive operation. The fastest known algorithm for computing the closure relies on matrix multiplication and can be achieved in $O(n^{\omega})$ where $\omega(\sim 2.3)$ is the exponent of the fastest matrix multiplication algorithm reported in the literature [33, 34, 115]. The additional drawback of this approach is that for sparse DAGs, where the structure is very similar to a tree, one is still forced to put up with computing the closure of the entire DAG. Ideally, one would hope for a technique that exploits the decomposition of the DAG into a spanning tree and a set of additional edges. LCA computation can then proceed by considering reachability information over the spanning tree and over the rest of the DAG separately. This would let us achieve a fast algorithm for computing pairwise LCAs in a sparse DAG.

4.3 Identifying potential LCAs for a vertex pair

In this section, we give an overview of our approach to computing the LCA of a vertex pair in constant time after polynominal preprocessing. For the subsequent discussions, we assume that the DAG under consideration is rooted and static. If there are multiple parentless vertices in the DAG, we can always introduce a single parent for the parentless vertices to make the DAG rooted.

We perform a depth first walk of the DAG and classify the edges as tree, forward and cross edges [35]. This can be achieved by preordering and postordering the vertices in a DAG and has the same computational costs as a depth first traversal of the rooted DAG. For computing the LCA, the set of forward edges can be safely ignored. These edges introduce a redundant order between two vertices that are already connected.

4.3.1 Overview of our approach to computing representative LCAs

With forward edges eliminated from the DAG, we are now left to deal with tree edges and cross edges. Subsequently, whenever we refer to a DAG, we assume that the DAG only contains tree and cross edges. For any vertex, we now have two kinds of ancestors. One kind, which we call *tree ancestors* reach the vertex through the spanning tree. The other kind, which we call *cross ancestors*, are all ancestors that are not tree ancestors.

We now give a brief overview of our approach to computing the respresentative LCA for two vertices x and y. Equations 4.1 and 4.2 show how the set of ancestors $A_{\mathtt{x}}$ and $A_{\mathtt{y}}$ for vertices x and y respectively are composed of tree ancestors (denoted by $A^t_{\mathtt{x}}$ and $A^t_{\mathtt{y}}$) and cross ancestors (denoted by $A^c_{\mathtt{x}}$ and $A^c_{\mathtt{y}}$) for the vertices.

$$\begin{aligned}
A_{\mathtt{x}} &= A^t_{\mathtt{x}} \cup A^c_{\mathtt{x}} && (4.1)\\
A_{\mathtt{y}} &= A^t_{\mathtt{y}} \cup A^c_{\mathtt{y}} && (4.2)\\
LCA(x,y) &= \max_{topo}[A_{\mathtt{x}} \cap A_{\mathtt{y}}] && (4.3)\\
&= \max_{topo}[\,\{\max_{topo}(A^t_{\mathtt{x}} \cap A^t_{\mathtt{y}}),\ \max_{topo}(A^t_{\mathtt{x}} \cap A^c_{\mathtt{y}}),\\
&\quad \max_{topo}(A^c_{\mathtt{x}} \cap A^t_{\mathtt{y}}),\ \max_{topo}(A^c_{\mathtt{x}} \cap A^c_{\mathtt{y}})\,\}\,]\\
&= \max_{topo}\{\text{TT-PLCA, TC-PLCA, CT-PLCA, CC-PLCA}\} && (4.4)
\end{aligned}$$

Similar to other reported techniques for LCA computation in DAGs , we assume the LCA for vertices x and y to be the vertex with the maximum topological number amongst the ancestors common to x and y [20] [36] [18]. This is used in equation 4.3. Equation 4.4 expands on equation 4.3 and shows how we can identify the representative LCA by shortlisting 4 potential LCAs (TT-PLCA, TC-PLCA, CT-PLCA, CC-PLCA) and then picking the one with the highest topological number.

We now show that if we rearrange the arguments of the LCA query such that the postorder number of the first argument is always greater than the postorder number of the second argument then we don't need to calculate the TC-PLCA in order to compute the LCA.

Definition 5. *For a vertex* `v` *in a DAG,* `pre(v)` *and* `post(v)` *denote the pre-order and post-order numbers for* `v` *in the spanning tree that covers the DAG.*

Definition 6. *A query of the form LCA(*`x`*,*`y`*) is considered argument-arranged if* `post(x)` $>$ `post(y)`*.*

Lemma 4.3.1. *If* `pre(x)` $>$ `pre(y)` *and* `post(x)` $>$ `post(y)` *and* `x` *reaches* `y` *then* `x` *is a cross ancestor of* `y`*.*

Proof Straightforward. The proof follows directly from the manner in which preorder and postorder numbers are allocated during a depth first walk. □

Lemma 4.3.2. *It is not necessary to compute the* CT-PLCA *for an LCA query if the query is argument-arranged.*

Proof For an *argument-arranged* query LCA(`x`, `y`), there can be two cases.

- `pre(x)` $<$ `pre(y)`: Since `post(x)` $>$ `post(y)` by the virtue of argument-arrangement, it immediately follows that there is a path in the spanning tree from `x` to `y`. In this case, the LCA of `x` and `y` is `x`. Thus, the LCA of `x` and `y` can be easily computed during the computation of TT-PLCA and we do not need to consider the CT-PLCA for this case.

- `pre(x)` $>$ `pre(y)`: Let `p` be an arbitrary cross ancestor of `x`. Then, `pre(p)` $>$ `pre(x)` $>$ `pre(y)` and `post(p)` $>$ `post(x)` $>$ `post(y)`. From lemma 4.3.1, it follows that if `p` reaches `y` then `p` is a cross-ancestor of `y` as well. It follows that $(A^c_{\mathtt{x}} \cap A^t_{\mathtt{y}}) = \phi$. Hence, we do not need to compute the CT-PLCA in this case as well.

Therefore, a simple arrangement of the arguments to the LCA query eliminates the need for computing the CT-PLCA. For the identifying TT-PLCA, the application of the RMQ technique to the spanning tree suffices. However, computing the TC-PLCA and the CC-PLCA is more involved and techniques to compute these are described in sections 4.4 and 4.5 respectively.

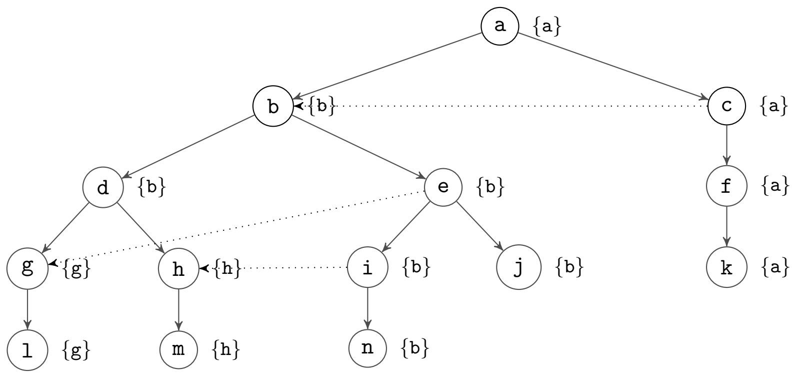


Figure 4.1: A directed acyclic graph with all vertices annotated with the corresponding clusterhead

4.3.2 Decomposing a DAG into clusters

There are two kinds of vertices in the DAG; one kind has an incoming spanning tree edge and the other kind has incoming cross-edges in addition to the tree edge. We denote the set of vertices of the former kind as $\downarrow$ and the set of vertices of the latter kind as $\downarrow^{+c}$. For the vertices in $\downarrow^{+c}$, if we ignore the incoming edges to these vertices, the DAG can be seen as a composition of trees. The only way to reach a vertex in these trees from a vertex external to it is by passing through its root - a vertex that belongs to $\downarrow^{+c}$. We call these component trees of the DAG as *clusters* and the root of the cluster as the *clusterhead*.

Definition 7. *Clusters are component trees of a DAG obtained by discarding all incoming edges to vertices that have both incoming spanning tree edges and cross edges.*

Fig. 4.1 shows the vertices of an example DAG annotated with clusterheads for the cluster to which they belong. After edge classification, cluster identification can be performed by a simple traversal of the spanning tree in $O(n)$ time where n is the number of vertices in the DAG.

If we are testing reachability from vertex `x` to vertex `y` and they belong to the same cluster, we only need to consider the edges of the spanning tree that covers the DAG. Otherwise, we have to additionally check for reachability from `x` to the clusterhead for `y` through a combination of tree and cross edges. In this context, the advantage that clusters offer is that we do not need to compute the transitive closure at the level of vertices but at the level of clusters; an approach that is significantly faster for sparse graphs [113].

Since the first step in computing the LCA is identifying common ancestors for the query vertices, reachability analysis has a direct bearing on the computation of the LCA. Due to the formulation of clusters, the computation of TC-PLCA and the CC-PLCA can be based on a combination of vertex labelling and small matrix lookups using the annotated labels in a manner similar to [113]. These small matrices are derived from a single matrix that captures the transitive reachability from cross edge sources to clusterheads. In the rest of this chapter, for an argument-arranged LCA query, the annotation at the first argument is used to index the rows of these small matrices and the annotation at the second argument is used to index the columns of the small matrices.

4.4 Identifying the TC-PLCA

To compute TC-PLCA(`x`, `y`) one does not need to consider all ancestors of `x` in the spanning tree. Instead, it is sufficient to pick just 2 cross-edge sources (denoted as $\mathtt{s}_<$ and $\mathtt{s}_>$) which we call *proximals.*

Definition 8. *For the query TC-PLCA(*`x`*,* `y`*), the proximals are defined as the cross-edge sources that immediately precede and succeed* `x` *in the pre-order sequence of vertices and reach the clusterhead for* `y` *(hence, reach* `y` *itself). Evaluation of reachability between proximals and clusterheads considers both tree and cross edges in the DAG.*

4.4.1 Picking appropriate proximals for a vertex

Let the TC-PLCA of two vertices `x` and `y` be denoted as `l`. `l` is `y`'s cross ancestor and reaches `y` through a combination of tree and cross edges. Until we reach a cross-edge source in the path from `l` to `y`, the path is composed of tree-edges entirely. Therefore, if we compute the TT-PLCA of `x` with every cross-edge source that reaches `y` and pick the vertex with the maximum depth in the spanning tree amongst the computed TT-PLCAs, we obtain `l`. However, with the aid of lemma 4.4.1, we will show that it is not necessary to consider all cross-edge sources that reach `y`. Instead, it is sufficient to pick the proximals only.

Lemma 4.4.1. *Let* [0,r] *be the range of preorder numbers of vertices in a DAG. For a given vertex* `x`*, the depth of* TT-PLCA(`x`,`y`) *in the spanning tree monotonically increases in the interval* [0, `pre(x)`] *and monotonically decreases thereafter.*

Proof The TC-PLCA(`x`, `y`) in our case needs to be the lowest vertex in the spanning tree that reaches both `x` and a cross-edge source that reaches `y`. The TC-PLCA reaches every cross-edge source in the sub-tree rooted at the TC-PLCA. Therefore, it is easy to see that for the TC-PLCA to reach a cross-edge source outside the sub-tree, it is imperative for the TC-PLCA to be higher up in the tree.

The proof follows directly from RMQ techniques to compute LCA in trees. Lets consider the range [0, `pre(x)`] first. If the depth is non-increasing then it means that there are two vertices `i` and `j` such that `pre(i)` $<$ `pre(j)` $<$ `pre(x)` and depth of the vertex returned by `RMQ(x,i)` is greater than that returned by `RMQ(x,j)`. The RMQ techniques relies on Euler tour of the graph which includes all subranges in identifying a vertex with the greatest depth. In the above case, in order to identify the LCA for `x` and `i`, all vertices having preorder numbers in [`pre(i)`, `pre(x)`] are also checked since the set of such vertices is a subset of vertices having preorder numbers in [`pre(j)`, `pre(x)`]. Therefore, the above result is a contradiction. Similarly, we can prove the lemma for (`pre(x)`,r]. □

Let $\mathbb{S}_<$ be the set of cross-edge sources having a pre-order number less than `x` and reaching the clusterhead for `y`. The first proximal, which we denote as $\mathtt{s}_<$, is the vertex with the highest pre-order number in $\mathbb{S}_<$. Similarly, let $\mathbb{S}_>$ be the set of cross-edge sources having a pre-order number greater than `x` and reaching the clusterhead for `y`. The second proximal, which we denote as $\mathtt{s}_>$, is the one with the lowest pre-order number in $\mathbb{S}_>$.

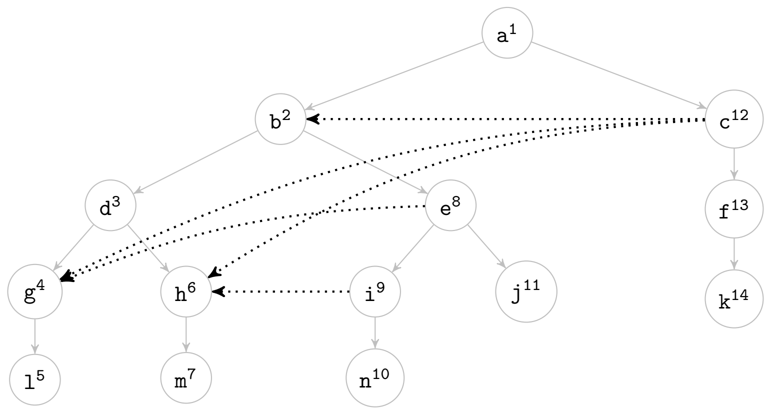
Identification of proximals simplifies the reachability information that needs to be captured for the TC-PLCA computation. Instead of considering reachability from one vertex to another, it is now sufficient to capture the transitive reachability information between cross-edge sources and clusterheads for the computing the TC-PLCA. This reduces the size of the reachability matrix that we need from a naive $O(n^2)$ to $O(c^2)$ where $c = max(N_s, N_t)$, N_s and N_t being the number of cross-edge sources and cross-edge targets (clusterheads) respectively.

4.4.2 Variations in proximals

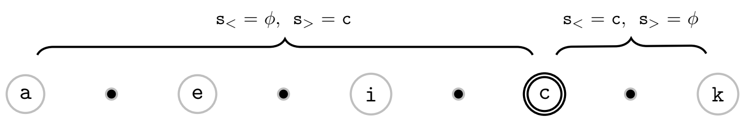
Answering arbitrary TC-PLCA queries requires us to annotate each vertex with proximals for each of the clusterheads. This is expensive and our next objective is to reduce the annotation overhead.

For a given vertex `y`, let all possible values of `x` in TC-PLCA(x,y) be written out in pre-order sequence. For all `x`'s in the pre-order sequence, the proximals change only when a cross edge source is encountered in the sequence. This is due to the fact that the proximals themselves are nothing but cross-edge sources.

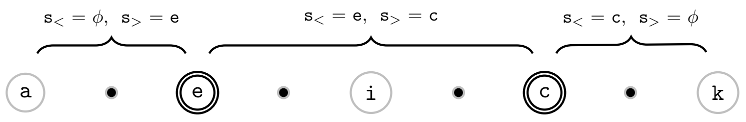
This point is further illustrated in Fig. 4.2 which shows the variations in proximals for all vertices for all clusterheads. The solid dots in Fig. 4.2 represent intermediate vertices in the pre-order sequence. Subfig. 4.2a shows the reachability between cross-edge sources and cross-edge targets for our example graph and aids the understanding of Subfig. 4.2b,



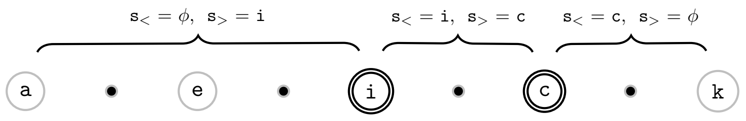
(a) Reachability between cross-edge sources and targets



(b) Proximals for all vertices for clusterhead b



(c) Proximals for all vertices for clusterhead g



(d) Proximals for all vertices for clusterhead h

Figure 4.2: Identifying proximals for all vertices for all clusterheads. Vertices are annotated with their pre-order numbers.

4.2c and 4.2d. In Subfig. 4.2b, 4.2c and 4.2d vertices are written out in pre-order sequence and cross-edge sources reaching clusterheads are marked with concentric circles. For each of the clusterheads, we also show how the values for proximals change as we run through the vertices written out in pre-order sequence.

Let us consider the cross-edge sources reaching clusterhead h in Subfig. 4.2d. i and c are the two cross-edge sources reaching the clusterhead h. For the pre-order range [0, pre(i)], the first proximal $s_<$ is undefined (denoted in the subfigure as ϕ). However, the second proximal $s_>$ is defined as i. In the next sub-range (pre(i), pre(c)] $s_<$ is i and $s_>$ is c. Finally, in the range (pre(c), pre(k)] $s_<$ is c and $s_>$ is undefined.

The proximals for vertices in pre-order sequence vary only when a cross-edge source is encountered. This subtle observation enables us to deploy a labeling and indexing scheme for identifying the proximals for any vertex. Therefore, we can annotate each vertex `x` with an index that points to a cross-edge source which has the lowest pre-order number amongst cross-edge sources having pre-order numbers higher than `x`. Let the identified cross-edge source be denoted as `u`. Since there are no other cross-edge sources in the interval (`pre(x)`+1, `pre(u)`), the proximals are same for both `x` and `u`. Subsequently, we can get the proximals for `x` by looking up the proximals for `u`.

4.4.3 Building and indexing the TC-matrix

In order to be able to deploy a labeling and indexing scheme for identification of proximals, we first build a matrix called the TC-matrix which holds the proximal information for cross-edge sources. The rows of the TC-matrix are indexed by clusterheads and its columns are indexed by cross-edge sources. The TC-matrix for our example graph is shown in table 4.1. In this subsection, we first discuss techniques for constructing the TC-matrix. Subsequently, we also discuss techniques to annotate vertices with labels to index the TC-matrix.

The first step in computing the TC-matrix is to compute the transitive closure for the rechability information between cross edge sources and clusterheads. We multiply an adjacency matrix based on the cross-edges with a second matrix that captures reachability from clusterheads to cross-edge sources (through the spanning tree) to obtain an intermediate matrix γ. The result of the closure over γ shows reachability from one cross-edge source to another through a combination of cross and tree edges. We may need to further amend γ because some cross-edge sources may be reachable from another solely through the spanning tree. It is well known the transitive closure of an adjacency matrix has the same computational complexity as a matrix multiplication. Hence, obtaining the transitive closure of γ has the same computational complexity as a matrix multiplication. Creating a reachability matrix between cross-edge sources and clusterheads from γ is straightforward and can be obtained by observing the cross-edges. Let this reachability matrix be denoted as $\mathcal{M}$.

Definition 9. *$\mathcal{M}$ is a sub-matrix of the transitive closure matrix for the DAG that captures the reachability information between cross-edge sources and clusterheads.*

The overall complexity of this reachability computation step can be limited to $O(c^{\omega})$ where ω is the exponent of the fastest matrix multiplication algorithm [34, 115].

Algorithm 4.1 TC-matrix computation

```
1: procedure ComputeTCPLCA($\mathcal{M}$)
2:     $prev\_s_< \leftarrow \phi$
3:     $Stack \leftarrow \phi$
4:     for each clusterhead $t$ do
5:         for each cross edge source $s$ do
6:             $s.s_< \leftarrow prev\_s_<$
7:             $Stack.push(s)$
8:             if $s \rightsquigarrow t$ then
9:                 while Stack is not empty do
10:                    $v \leftarrow Stack.pop()$
11:                    $v.s_> \leftarrow s$
12:                end while
13:                $prev\_s_< \leftarrow s$
14:            end if
15:        end for
16:        while Stack is not empty do
17:            $v \leftarrow Stack.pop()$
18:            $v.s_> \leftarrow \phi$
19:        end while
20:    end for
21: end procedure
```

	e	i	c
b	{ϕ,c}	{ϕ,c}	{ϕ,c}
g	{ϕ,e}	{e,c}	{e,c}
h	{ϕ,i}	{ϕ,i}	{i,c}

Table 4.1: TC-matrix

Upon obtaining $\mathcal{M}$, we use algorithm 4.1 to obtain the TC-matrix. We scan through the list of all cross-edge sources reaching each clusterhead (cf. 4-5) and push the cross-edge source onto a stack after updating the value for $s_<$ for it (cf. 6-7). The value of $s_<$ for a cross-edge source is set to be the same as the $s_<$ for the cross-edge source encountered immediately before it (denoted by $prev\_s_<$). Upon encountering a cross-edge source s that reaches the clusterhead, we pop all cross-edge sources on the stack to update the $s_>$ values for them (cf. 8-14). Additionally, we also update the value for $prev\_s_<$ to s. This process continues until we reach the cross-edge source with the highest pre-order number. At this stage, if there are any additional cross-edge sources on the stack, we set the $s_>$ value for these sources to be ϕ (cf. 16-19). Table 4.1 shows the TC-matrix for the DAG in Fig. 4.1 using this algorithm.

Algorithm 4.2 Labeling all vertices for indexing TC-matrix

```
 1: $Stack \leftarrow \phi$
 2: procedure LabelVerticesForTCPLCA(G)
 3:     LabelVertex(root(G), $\phi$, $\phi$)
 4:     while $Stack$ is not empty do
 5:         $v \leftarrow Stack.pop()$
 6:         $v.colIdx \leftarrow \phi$
 7:     end while
 8: end procedure
 9: procedure LabelVertex($n$, $rowIdx$, $colIdx$)
10:     Stack.push($n$)
11:     if n is a cross-edge source then
12:         if $colIdx$ is $\phi$ then
13:             $colIdx \leftarrow 0$
14:         else
15:             $colIdx \leftarrow colIdx + 1$
16:         end if
17:         while Stack is not empty do
18:             $v \leftarrow Stack.pop()$
19:             $v.colIdx \leftarrow colIdx$
20:         end while
21:     end if
22:     if $n$ is a clusterhead then
23:         if $rowIdx$ is $\phi$ then
24:             $rowIdx \leftarrow 0$
25:         else
26:             $rowIdx \leftarrow rowIdx + 1$
27:         end if
28:     end if
29:     $n.rowIdx \leftarrow rowIdx$
30:     for each $child$ of $n$ in the spanning tree do
31:         LabelVertex($child$, $rowIdx$, $colIdx$)
32:     end for
33: end procedure
```

In order to index the rows of the TC-matrix, we annotate vertices with a label for their clusterhead. In order to index the columns, we annotate each vertex with a second label that is based on proximal information for the vertex. The vertices can be labeled in $O(n + m)$ using algorithm 4.2 where m is the number of edges in the DAG. We trigger this algorithm using the root of the spanning tree that covers the DAG (cf ll 3). The function LabelVertex is responsible for annotating the row and column indices at every vertex for accessing the TC-matrix. We annotate the row label (cf ll 10-21) and column label (cf ll 22-29) with the aid of the variables rowIdx and colIdx. Similar to algorithm 4.1, while annotating column labels, we keep pushing vertices onto the stack until a cross-edge source is encountered. Upon encountering a cross-edge source, we pop all vertices on the stack and label the vertices with a column index that corresponds to the encountered cross-edge source. Our example DAG with annotated with the column and row indices (in that order) is shown in Fig. 4.3.

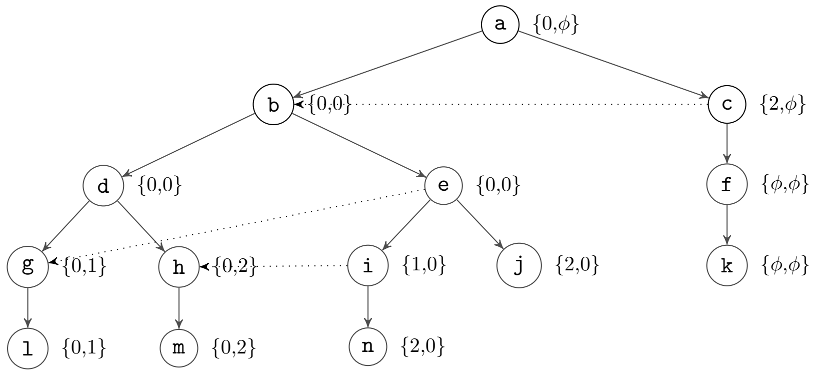


Figure 4.3: DAG vertices annotated with TC-matrix indices

4.5 Identifying the CC-PLCA

The CC-PLCA of a vertex pair has the highest topological number amongst the common cross ancestors that reach the pair. Computation of the CC-PLCA is done in three steps which are described below.

- Step 1 - We try to find out if any of cross-edge sources reach the both vertices in the query pair. If this is true, then the cross-edge source itself could be the CC-PLCA. For each query pair, we identify all cross-edge sources reaching both vertices and then choose one that has the highest topological number. We denote this vertex as τ. It may be the case that τ does not exist as there no cross-edge source that reaches both vertices. Therefore, we also need to consider LCAs of the cross-edge sources reaching the vertices as detailed in the step 2.

- Step 2 - For a vertex pair {x,y} let the distinct cross edge sources $\mathtt{c_x}$ and $\mathtt{c_y}$ reach x and y respectively. The LCA of $\mathtt{c_x}$ and $\mathtt{c_y}$ could potentially be a CC-PLCA for the vertex pair. Let the candidate CC-PLCA identified in this manner be denoted as $\overline{\tau}$. If S_x and S_y denote the set of all cross edge sources reaching x and y respectively, $\overline{\tau}$ can be identified in two stages. In the first stage, we create a shortlist of vertices by taking one vertex each from S_x and S_y and computing their LCA. Let this shortlisted set of vertices be denoted as $S_{\overline{\tau}}$. In the second stage, we choose the vertex with the highest topological amongst the vertices in $S_{\overline{\tau}}$.

- Step 3 - We choose the vertex that has the higher topological number between τ and $\overline{\tau}$ which gives us the CC-PLCA for the query pair.

4.5.1 A simplified approach to computing $\overline{\tau}$

Instead of computing the pairwise LCAs as detailed in step 2 above, we can obtain $\overline{\tau}$ by computing the pairwise TT-PLCA.

Lemma 4.5.1. *For an LCA query* LCA(x,y)*, let S_x and S_y be the set of cross-edge sources reaching the clusterheads of* x *and* y*. Let, $S^t_{\overline{\tau}}$ denote the set of vertices obtained by computing* TT-PLCA *of all pairs of vertices* c_x *and* c_y *such that* $c_x \in S_x$ *and* $c_y \in S_y$ *and* $c_x \neq c_y$*. $\overline{\tau}$ can be obtained by the picking the vertex with the highest topological number in $S^t_{\overline{\tau}}$.*

Proof Vertices in S_x and S_y form a partial order due to the fact that the set of vertices in S_x and S_y is transitively closed. Let us consider two cross edge sources c_x and c_y from the sets S_x and S_y respectively. During the LCA computation of c_x and c_y, we do not need to consider any cross ancestors of c_x and c_y towards identification of $\overline{\tau}$; they will be considered anyway when we consider other vertices in S_x and S_y and obtain their LCA. Therefore, it suffices to just compute the TT-PLCA of c_x and c_y. This discussion can be inductively extended to all pair-wise combinations of a vertex each from S_x and S_y. Therefore, if $S^t_{\overline{\tau}}$ denote the set of vertices obtained by computing TT-PLCA of all pairs of vertices c_x and c_y such that $c_x \in S_x$ and $c_y \in S_y$ and $c_x \neq c_y$. $\overline{\tau}$ can be obtained by the picking the vertex with the highest topological number in $S^t_{\overline{\tau}}$. □

For the remainders of this discussion, we refer to vertices that are TT-PLCAs of cross edge sources as *extras*.

4.5.2 CC-PLCA computation for all pairs of clusterheads

So far, we have discussed the CC-PLCA computation for a given vertex pair. It is important to reiterate two aspects of the CC-PLCA problem at this stage. Firstly, we are interested in the CC-PLCA computation of all vertex-pairs instead of any given pair. Secondly, since any cross ancestor reaches a vertex through its clusterhead, it would be sufficient to compute the CC-PLCA of all pairs of clusterheads. In order to compute τ for all pairs of clusterheads, we need reachability information from cross-edge sources to clusterheads. Let us denote this matrix by $\mathcal{M}$. In order to compute $\overline{\tau}$, we need reachability information between *extras* and clusterheads. Let this information be encoded in another reachability matrix which we denote as $\mathcal{M}_x$.

Definition 10. *$\mathcal{M}_x$ is a sub-matrix of the transitive closure matrix for the DAG that captures the reachability information between extras and clusterheads.*

The process of computing τ and $\overline{\tau}$ for all pairs of clusterheads from $\mathcal{M}$ and $\mathcal{M}_x$ respectively is straightforward. The details of computing the LCA from a reachability matrix can be found in [36]. The process is known as identification of the maximal witness in a boolean matrix product and has a best know runtime complexity of $O(c^{2.575})$ [36].

We have already computed $\mathcal{M}$ in section 4.4. We now discuss how $\mathcal{M}_x$ can be computed using $\mathcal{M}$ as an input. The initial step in the computation of $\mathcal{M}_x$ is to identify all *extras*. Simultaneously, we also need to keep track of which clusterheads are reached by which *extras*. One can naively enumerate the *extras* by obtaining pairwise TT-PLCA of all clusterheads. Since there are c cross-edge sources, the naive approach would entail $O(c^2)$ operations just to compute all TT-PLCAs. In addition for each of the TT-PLCA computation we have to keep track of clusterheads that the *extras* reach through reachability-set union operation. This would increase the worst-case complexity to $O(c^3)$. However, we will shortly show with the aid of a few lemmas that the algorithm can be simplified from a worst case complexity of $O(c^3)$ to $O(c^2 \log c)$. We first show through lemma 4.5.2 that it is not necessary to obtain pairwise TT-PLCA of all clusterheads.

Lemma 4.5.2. *Let $\mathcal{T}$ be a tree and the sequence $\mathcal{S} = \mathtt{v_1} \ldots \mathtt{v_p}$ be any p vertices from the tree written in post-order. Let $\mathtt{l}$ be the LCA of the nodes $\mathtt{v_1}$ and $\mathtt{v_2}$ and $\mathtt{v_k}$ be a vertex in $\mathcal{S}$ with the highest post-order number less than or equal to post($\mathtt{l}$). Then,* $\mathrm{LCA}(\mathtt{v_1}, \mathtt{v_i}) = \mathtt{l}$ *if* $2 \le i \le k$ *and* $\mathrm{LCA}(\mathtt{v_1}, \mathtt{v_i}) = \mathrm{LCA}(\mathtt{l}, \mathtt{v_i})$ *if* $k < i \le p$.

Proof Recall from the theory of post-order numbering for vertices in a tree that a vertex is numbered after numbering all its descendants. Therefore, if a vertex x is the ancestor of another vertex y then x is the ancestor of all other vertices that have post-order numbers in the range [post(y),post(x)].

Case 1. $2 \le i \le k$: Let $\mathtt{l_i} = \mathrm{LCA}(\mathtt{v_1}, \mathtt{v_i})$. In a tree, there is only one path from the root to every vertex which passes through all ancestors of vertex and there exists a total order amongst the ancestors of the vertex. We know that both $\mathtt{l_i}$ and $\mathtt{l}$ are ancestors of $\mathtt{v_1}$. So, there exists an order between $\mathtt{l_i}$ and $\mathtt{l}$. There are two cases possible, either $\mathtt{l_i}$ is an ancestor of $\mathtt{l}$ or $\mathtt{l_i}$ is a descendant of $\mathtt{l}$. We show by contradiction that neither is possible.

Since $\mathtt{post(v_1)} < \mathtt{post(v_i)} \leq \mathtt{post(v_k)} \leq \mathtt{post(l)}$ and $\mathtt{l}$ is an ancestor of $\mathtt{v_1}$, $\mathtt{l}$ is an ancestor of $\mathtt{v_i}$ as well. If $\mathtt{l}$ is a descendant of $\mathtt{l_i}$, then LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = $\mathtt{l}$. This is a contradiction since we know that LCA($\mathtt{v_1}$, $\mathtt{v_i}$) = $\mathtt{l_i}$. Also, since $\mathtt{post(v_1)} < \mathtt{post(v_2)} < \mathtt{post(v_i)}$ and $\mathtt{l_i}$ is the ancestor of $\mathtt{v_1}$, $\mathtt{l_i}$ is an ancestor of $\mathtt{v_2}$ as well. If $\mathtt{l_i}$ is a descendant of $\mathtt{l}$, then LCA($\mathtt{v_1}$, $\mathtt{v_2}$) = $\mathtt{l_i}$. A contradiction again since we know that LCA($\mathtt{v_1}$, $\mathtt{v_2}$) = $\mathtt{l}$. Therefore, $\mathtt{l_i}$=$\mathtt{l}$.

Case 2. $k < i \leq p$: Once again, let $\mathtt{l_i}$ = LCA($\mathtt{v_1}$, $\mathtt{v_i}$). For this case, we have $\mathtt{post(l_i)} > \mathtt{post(v_i)} > \mathtt{post(l)} \geq \mathtt{post(v_k)} > \mathtt{post(v_1)}$. Also, we know that both $\mathtt{l_i}$ and $\mathtt{l}$ are ancestors of $\mathtt{v_1}$. There is a total order between $\mathtt{l_i}$ and $\mathtt{l}$. Since $\mathtt{post(l_i)} > \mathtt{post(l)}$, $\mathtt{l_i}$ must be an ancestor of $\mathtt{l}$ in the tree and all paths from $\mathtt{l_i}$ to $\mathtt{v_1}$ pass through $\mathtt{l}$. Thus, LCA($\mathtt{v_1}$, $\mathtt{v_i}$) can be rewritten as LCA($\mathtt{l}$, $\mathtt{v_i}$) for this case. □

Lemma 4.5.2 shows that if we have a sequence of vertices $\mathcal{S} = \mathtt{v_1} \ldots \mathtt{v_p}$ in post-order sequence, the list of unique TT-PLCAs for all vertex pairs can be obtained by a recursive operator. This operator computes the TT-PLCA of the first two vertices in the sequence, adds the TT-PLCA back into the sequence (according to its post-order number) and drops the first vertex from the sequence. Assuming that the operator terminates, it continues to run until it exhausts $\mathcal{S}$. Based on this observation, we formulate a recursive operator Λ to identify the set of *extras* and clusterheads reachable from these extras. We first give a formal presentation of Λ and then discuss its correctness and termination properties.

Definition 11. Λ *is an operation on the set of cross-edge sources* $\mathcal{S}$ *sorted in ascending order of their post-order numbers such that:*

1. *It calculates the TT-PLCA* $\mathtt{l}$ *of first two vertices in* $\mathcal{S}$
2. *It updates the clusterheads reachable from* $\mathtt{l}$ *with those reachable from the first two vertices in* $\mathcal{S}$
3. *It inserts* $\mathtt{l}$ *back into* $\mathcal{S}$ *while maintaining the vertex ordering in* $\mathcal{S}$
4. *It drops the first vertex in* $\mathcal{S}$
5. *If* $\mathcal{S}$ *has at least 2 elements,* Λ *calls itself with* $\mathcal{S}$ *as an argument otherwise* Λ *terminates*

Lemma 4.5.3. *Λ correctly identifies all extras and all clusterheads reachable from these extras.*

Proof The TT-PLCA of $\mathtt{v_1}$ and $\mathtt{v_i}$ where $2 < i \leq k$ is $\mathtt{l}$. If $\mathtt{l}$ is not in $\mathcal{S}$ yet, we insert it in $\mathcal{S}$. We update the clusterheads reached by $\mathtt{l}$ with the clusterheads reached by $\mathtt{v_1}$ and $\mathtt{v_2}$. As Λ operates on $\mathcal{S}$, the pairwise TT-PLCA of $\mathtt{l}$ with vertices $\mathtt{v_{k+1}} \ldots \mathtt{v_p}$ will gives us the other *extras* that may arise due to $\mathtt{v_1}$. Therefore, we do not need $\mathtt{v_1}$ anymore and it can be dropped. Thus, Λ preserves the information about *extras*. If we assume termination of Λ (which will be proved later), at some stage $\mathtt{l}$ will become the first vertex in the sequence. If we find that $\mathtt{l}$ is not a cross-egde source, we add it to the set of *extras*.

Apart from reaching clusterheads directly, *extras* can also reach clusterheads transitively through other cross-edge sources reachable from them in the spanning tree. We need to show that when *extras* are dropped from $\mathcal{S}$, the identified set of clusterheads reachable from it is complete. Let $\mathtt{l_i}$ TT-PLCA of two vertices $\mathtt{v_i}$ and $\mathtt{v_{i+1}}$ such that $\mathtt{post(v_1)} < \mathtt{post(v_i)} < \mathtt{post(v_{i+1})} < \mathtt{post(l)}$. According to lemma 4.5.2, $\mathtt{l}$ also reaches $\mathtt{v_i}$ and $\mathtt{v_{i+1}}$. Therefore, $\mathtt{l_i}$ could be either $\mathtt{l}$ or one of its descendants in the spanning tree. If $\mathtt{l_i}$ is $\mathtt{l}$, reachable clusters from $\mathtt{l}$ are updated with those reachable from $\mathtt{v_i}$. Otherwise, clusterheads reachable from $\mathtt{l_i}$ are updated and $\mathtt{l_i}$ is inserted in $\mathcal{S}$ in a position between $\mathtt{v_{i+1}}$ and $\mathtt{l}$. $\mathtt{l_i}$ continues to remain in the sequence until it comes to the beginning of the sequence. Then, the information about clusterheads reachable from it is added to one its ancestors (which could be $\mathtt{l}$ or one of its descendants) and so on.

The discussion reveals a powerful property of Λ - no vertex is dropped without handing over clusterhead reachability information to a tree ancestor of the vertex that is already in $\mathcal{S}$. More importantly, it is not possible for $\mathtt{l}$ to come to the head of the sequence until all the vertices that have post-order numbers in the range $[\mathtt{post(v_1)}, \mathtt{post(l)}]$ have been dealt with. As a result, we will ultimately reach a stage in Λ where all reachable clusterheads from $\mathtt{l}$ have been correctly identified. $\square$

Lemma 4.5.4. *Λ terminates in $O(c)$ iterations.*

Proof The proof follows straightforwardly from the observation that for a set of nodes in a tree, the number of unique LCAs generated through pairwise LCA operations on nodes in the set is no larger than the cardinality of the set.

Let the $i_1, i_2 \cdots i_n$ be the indices of the first occurrence of $\mathtt{v_1}$, $\mathtt{v_2}$ $\cdots$ $\mathtt{v_n}$ in the Euler tour of the DAG. From RMQ discussions in [20], [43] and section 4.1, it is known that the index of the LCA for any vertex-pair $\mathtt{v_k}$ and $\mathtt{v_{k+1}}$ must lie in the range $[i_k, i_{k+1})$. Now consider the tuple $\{\mathtt{v_{k-1}},\mathtt{v_k},\mathtt{v_{k+1}}\}$. We prove that the number of unique LCAs for this small subset is no larger than two in order for the lemma to be true.

Given index($\mathtt{LCA(v_{k-1},v_k)}$) $\in [i_{k-1}, i_k)$ and index($\mathtt{LCA(v_k,v_{k+1})}$) $\in [i_k, i_{k+1})$, let the index of the result of the only other possible LCA query - $\mathtt{LCA(v_{k-1},v_{k+1})}$ - be i. If $i \in [i_{k-1}, i_k)$ and $i \neq$ index($\mathtt{LCA(v_{k-1},v_k)}$) then there is another vertex in $[i_{k-1}, i_k)$ which has a lower value than $\mathtt{LCA(v_{k-1},v_k)}$. This is a contradiction. Similarly, we can prove that $i \notin [i_k, i_{k+1})$. Generalizing this argument, we can prove that the cardinality of the sequence $\mathcal{S}$ decreases monotonically and the recursive refinement terminates in $O(c)$ steps. □

4.5.3 Algorithmic details

Having discussed the technical details of the process for computing the CC-PLCA, we are now ready to discuss the algorithmic details. In this subsection we present the algorithm that computes $\mathcal{M}_x$ from $\mathcal{M}$. The rest of the process for computing the CC-PLCA relies on computation of maximal witnesses in a boolean matrix product and can be found in the literature [36]. The algorithm discussed in the sub-section closely follows the theoretical discussions on CC-PLCA computation.

Algorithm 4.3 Finding clusters reachable through CC-PLCAs that are not cross edge sources

```
1: for $v \in C_s$ do                          ▷ $C_s$ is the set of cross-edge sources
2:     $\mathcal{S}.Enqueue(v)$                          ▷ $\mathcal{S}$ is a priority queue
3: end for
4: $\mathcal{M}_x \leftarrow \phi$
5: while !$\mathcal{S}.empty()$ do
6:     $v_1 \leftarrow Q.dequeue()$
7:     if !$v_1.isCrossSource()$ then
8:         $\mathcal{M}_x \leftarrow \mathcal{M}_x \cup v_1$
9:     end if
10:    if !$\mathcal{S}.empty()$ then
11:        $v_2 = \mathcal{S}.head()$
12:        $l = TT\_PLCA(v_1, v_2)$
13:        $l.clusters \leftarrow l.clusters \cup getReachable(\mathcal{M}, v_1)$
14:        if $v_2 != l$ then
15:            $\mathcal{S}.enqueue(l)$
16:        end if
17:    end if
18: end while
```

Algorithm 4.3 uses the reachability matrix $\mathcal{M}$ and the set of cross-edge sources C_s as input. It uses a priority queue as the basic data structure. The priority queue uses the

	b	g	h
b	b		
g	c	g	
h	c	e	h

Table 4.2: CC-PLCAs

post-order number as the ranking criteria. In the algorithm, we first initialize the priority queue with the set of cross-edge sources (cf ll 1-3). We dequeue a vertex $\mathtt{v_1}$ and check whether it is one of *extras*. If it is, we add it to $\mathcal{M}_x$ (cf ll 6-9). Then we compute the TT-PLCA of $\mathtt{v_1}$ and the head of the sequence $\mathcal{S}$. The TT-PLCA is denoted as `l`. We update the clusters reachable from the `l` as well (cf ll 12-13). Finally, we put back `l` in the sequence $\mathcal{S}$ (cf ll 14-18).

We know from lemma 4.5.4 that the outer loop in algorithm 4.3 runs $O(c)$ times. For each of the iterations, we insert a vertex in the sequence which takes $O(\log c)$ time because $\mathcal{S}$ is already sorted and we update the reachable clusterheads which takes another $O(c)$ time. So, the worst case time complexity for obtaining $\mathcal{M}_x$ from $\mathcal{M}$ is $O(c^2)$. At the same time, it is clearly evident that we do not need more space than $O(c^2)$. After obtaining $\mathcal{M}$ and $\mathcal{M}_x$, we just need to do 2 maximal witness of boolean matrix product operations and a comparison of two $c \times c$ matrices. Given that the maximal witness operation has a time and space complexity of $O(c^{2.575})$ and $O(c^2)$ respectively, we can conclude that the CC-PLCA computation has a time and space complexity of $O(c^{2.575})$ and $O(c^2)$ respectively.

Table 4.2 shows the CC-PLCAs obtained for all pairs of clusterheads for our example graph. Indexing this matrix requires no further labelling since we have already annotated each vertex its corresponding clusterhead during the TC-PLCA computation. These clusterhead labels can be used to index table 4.2 as well.

The final step in computing the LCA of any two arbitary vertices is to return the vertex that has the highest topological number amongst the TT-PLCA, TC-PLCA, CC-PLCA. This operation can be achieved in constant time.

Theorem 4.5.5. *The representative least common ancestor of a vertex-pair in a DAG can be answered in constant time after $O(n + c^{2.575})$ preprocessing requiring $O(n + c^2)$ space.*

Discussion. In order to answer TC-PLCA and CC-PLCA queries, we use a combination of vertex labeling and small-matrix look-ups. Labeling of vertices for indexing TC-matrix and the CC-PLCAs matrix relies on a depth-first traversal of the DAG and can easily be integrated with the initial traversal of the DAG for edge classification. Similar to the depth-first traversal of a DAG, labeling has a time and space cost of $O(n+m)$ and $O(n)$ respectively.

The pre-processing phase of our algorithm computes the TC-matrix and the CC-PLCA matrix for answering TC-PLCA and CC-PLCA queries efficiently. The TC-matrix is computed from $\mathcal{M}$ with a time and space cost of $O(c^2)$. Computing $\mathcal{M}$ from the cross-edge information derived from the initial traversal of the DAG can be done using a sequence of matrix multiplications. This has a time and space cost of $O(c^\omega)$ and $O(c^2)$ respectively. Thus, the TC-matrix can be computed with an overall time and space cost of $O(c^\omega)$ and $O(c^2)$ respectively.

For the computation of the CC-PLCA matrix we need to perform an element-wise comparison of matrices that hold τ and $\overline{\tau}$ for all combinations of clusterheads. This operation can be done with a time and space cost of $O(c^2)$. The matrices that hold τ and $\overline{\tau}$ can be respectively obtained from $\mathcal{M}$ and $\mathcal{M}_x$ through the maximal witness of the boolean matrix product operation. This has a time and space cost of $O(c^{2.575})$ and $O(c^2)$ respectively. It has been further shown that $\mathcal{M}_x$ itself can be obtained from $\mathcal{M}$ in $O(c^2 \log c)$ time and $O(c^2)$ space. Thus, the overall time and space cost of obtaining the CC-PLCA matrix is $O(c^{2.575})$ and $O(c^2)$ respectively.

Finally, answering TT-PLCA queries in constant time requires us to pre-process the spanning tree that covers the DAG. This is achieved in $O(n)$ time and space using existing techniques for LCA computation in trees.

4.6 Summary

In this chapter, we have proposed a fast and scalable technique to identify representative LCAs in a DAG. The computational requirement of our technqiues scales itself based on the number of vertices in the DAG with incoming or outgoing cross edges. We achieved this by taking the spanning tree of the DAG as the base structure for our analysis and computing the transitive closure of the additional reachability information in the graph. Then, we identified potential LCAs depending on all possible types of paths that may exist between the potential LCA and the query vertex. The vertex with the maximum topological number amongst the PLCAs was identified as the representative LCA. The reported techniques provide best of both worlds in terms of computational requirements: LCA computation using our algorithm proceeds on trees and dense DAGs in the most efficient techniques reported currently for these structures in the literature. The computational requirements of our algorithm interpolate seamlessly for anything in-between these two categories. Also, unlike exisiting algorithms that compute the transitive closure of the entire DAG, we compute the closure for only cross-edge source and targets which renders our algorithms more efficient that those reported in the literature.

Chapter 5

Evaluation of adaptive pre-processing for security lattices

5.1 Introduction

In the previous chapter we discussed the theoretical foundations of an adaptive approach to pre-processing DAGs for answering lowest common ancestor queries in constant time. However, the dicussion was limited to theoretical costs of pre-processing the DAG. In this chapter, we discuss the application of the algorithm to lattice queries normally encountered in information flow analysis of programs. We experimentally evaluate the algorithm presented by testing it against a range of security lattices with widely varying structures and show that given for the wide spectrum of security lattices encountered in practice, the adaptive algorithm presented in chapter 4 provides a seamless means of pre-processing the lattice in order to answer join ($\sqcup$), meet ($\sqcap$) and ordering ($\leq$) queries in constant time. Since join is the dual of the meet query and ordering queries are a special case of join queries ($\mathtt{a} \leq \mathtt{a}'$ if $\mathtt{a} \sqcup \mathtt{a}' = \mathtt{a}'$), we limit our experiments to the join query only.

The main contributions of this chapter are as follows.

- We discuss the wide spectrum of the structure of security lattices encountered in practice when it comes to lattices governing flow of information in programs. We show that the policy lattices that govern information flow analysis in real-world programs could range from trees with a bottom ($\perp$) element all the way to dense DAGs. This serves as the motivation behind using the adaptive pre-processing technique described in chapter 4.
- We compare and contrast the adaptive approach with existing techniques for pre-

processing lattices for answering lowest common ancestor queries. We discuss the relative advantages and disadvantages of a cluster based approach as advocated in the adaptive approach compared with techniques presented in the literature for trees and DAGs. As well as the pre-processing times, we also discuss the cost for querying after pre-processing the lattice. This is important because despite its pre-processing advantages, the adaptive approach uses a potentially more involved querying approach.

- We demonstrate experimentally the superiority of the adaptive cluster-based algorithm. We show that not only does it lends a level of abstraction to pre-processing lattices but its computational costs are a direct function of the lattice's latent scope for decomposition into clusters. This makes the adaptive approach a preferred option in applications like language-based security which have to deal with lattices that are structurally diverse. With regards to the query times, our experiments show lattices pre-processed by the adaptive approach take longer to query than existing techniques. However, this difference in answering time per query is miniscule when compared to the gains in pre-processing times.

The rest of the chapter is organised as follows. In section 5.2, we present the extremities of the structural spectrum of security lattices encountered in practice. We discuss the algorithmic options for pre-processing these lattices in section 5.3 and showcase the advantages of a cluster-based approach to pre-processing these lattices in section 5.4. We further discuss why, despite the advantages it offers in terms of pre-processing, the query time for a cluster-based approach can be expected to be more expensive than exisiting techniques. We demonstrate the advantages of structure-sensitive algorithms experimentally in section 5.5. Finally, we summarise the main results of this chapter in section 5.6.

5.2 Structure-spectrum of security lattices

There are a wide variety of security lattices that govern how information should flow through programs. On one hand, we have type-lattices which are similar in structure to a tree with a few additional cross-edges. On the other hand we have powerset lattices which are akin to dense DAGs with each vertex in the DAG having multiple incoming edges. In this section, we demonstrate the structural contrast in such lattices. In section 5.2.1, we discuss type-lattice directed language security where the type lattice is similar in structure to a tree. In in section 5.2.2, we discuss powerset lattices used in mashup security the structure of which is similar to a dense DAGs.

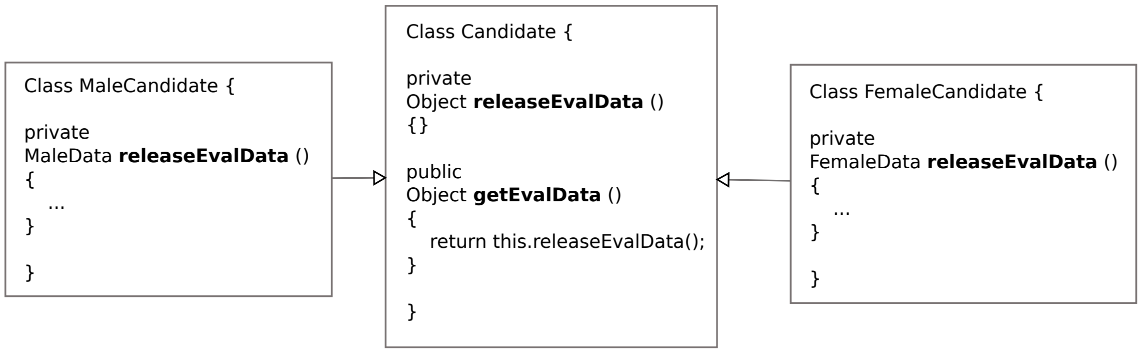
5.2.1 Class-level non-interference

For a secure computation, it is necessary that the low-security part of the output should not depend on the high-security part of the input. This is commonly known as non-interference. In chapter 3, we have described a model of information flow that was too fine grained; each value and expression has its individual security level. In reality, however, sometimes it is useful to relax such a strict criteria for non-interference. Abstract non-interference [47] is a more relaxed form of non-interference where the observational power of attackers are limited; it deals with attackers that observe only properties of data rather than exact values.

In [117], a model of information flow was presented with the class representing a collection of objects with the same structure as an abstract property. Thus, from the point of view of lattice-directed information flow control, classes would represent elements in the security lattice and the subclass relation would represent the ordering between the elements.

Class-level non-interference mandates that a class is secure if observing the output of any of its public methods does not reveal any type information regarding its inputs. Consider a scenario where the evaluator in a test is biased against candidates based on their gender. It is imperative that when the evaluator requests the data for evaluating a candidate (which could be answers in a test), no information is released about the gender of the candidate. However, the system may have been designed as shown in figure 5.1 due to historical reasons; this could be to monitor equal opportunities for applicants regardless of their gender. In this case, we have a `Candidate` class which has two subclasses `MaleCandidate` and `FemaleCandidate` for male and female candidates respectively. Such an implementation, however, is flawed from the perspective of protecting the gender data from the examiner because observing the return type of `getEvalData()` using the `instanceof` method (from Java) will reveal whether the evaluation data belongs to a male candidate or a female one.

In class-level non-interference, each variable as well as each static or instance field of a class is mapped to an abstract value. An abstract value is a set of classes each annotated with a security flag. For example, in the figure 5.1, the object `this` could map to any of the three classes: it could be either `Candidate` or any of its subclasses (`MaleCandidate` or `FemaleCandidate`). In program analysis for class-level non-interference, the abstract



```
Class MaleCandidate {

private
MaleData releaseEvalData ()
{
   ...
}

}
```

```
Class Candidate {

private
Object releaseEvalData ()
{}

public
Object getEvalData ()
{
   return this.releaseEvalData();
}

}
```

```
Class FemaleCandidate {

private
FemaleData releaseEvalData ()
{
   ...
}

}
```

Figure 5.1: An implementation that leaks sensitive information

value to which `this` would map would be $\texttt{Candidate}^{\texttt{H}}_{\downarrow}$. Here, the $\downarrow$ represents that `this` could map to either the `Candidate` class or any of its sub-classes. The annotation `H`, which stands for *high*, denotes that since the object maps to multiple classes, it is vulnerable to security attacks that expose class-level data. Hence, augmenting the class information for `this` with a *high* flag enables the verifier to check whether class information for objects that map to a secure abstract value (denoted by `H`) flow to those that map to an insecure abstract value (annotated by a privilege level `L` which is less than `H`).

In class-level non-interference, ad-hoc security modifiers such as `L` and `H` are added at the level of classes to denote whether a field should to be regarded as insecure or secure. The security policy requires that there be no flows from `H` fields to those marked with `L`; such a flow constitutes an illicit flow. However, this is contingent upon verifying whether a flow is legal at the level of classes in the first instance. If the flow is legal at the level of classes, only an additional trivial check is necessary to establish class-level non-interference. This check verifies the absence of flows from fields marked as `H` to those marked as `L`. Thus, the class hierarchy serves as the broad-based policy that determines what constitutes a secure flow of class-level information. If we have a large class hierarchy then it is important to answer queries (such as $\leq$, $\sqcup$ and $\sqcap$) of the class-level non-interference verifier in constant time. In this chapter, we investigate the structure of class-hierarchies of some commonly used open source software to determine suitable algorithms that would pre-process the hierachy to answer queries of the verifier in constant time.

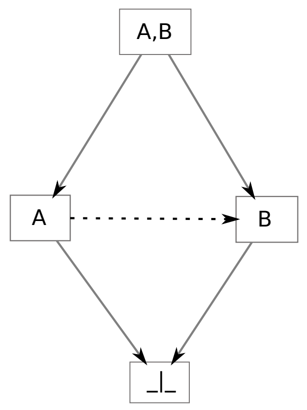
5.2.2 Mashup security

Web mashups are growing ever-popular because integration of services from multiple providers into a single hosting page provides unprecedented functionalities. The hosting page for the mashup, called the *integrator*, is often a hotbed of interaction between various components in it. The components are typically loaded from multiple origins which have varying levels of trust and therefore, the issue of securing information flow between components in the mashup is an important issue.

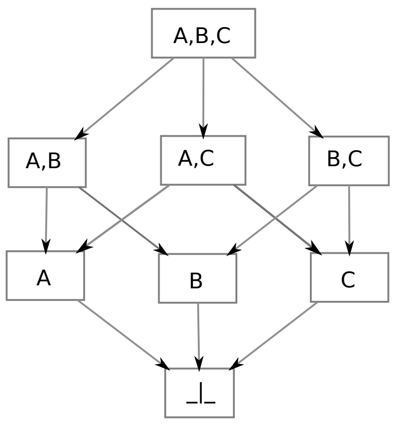
Recently, it was shown that a lattice based approach to mashup security is suitably able to deal with multi-origin trust issues [71] and provides an effective basis for secure information interchange between mashup components. Such a lattice for controlling information flow in a mashup is constructed from the mashup itself. It is the powerset lattice of the set of origins from where content is sourced for the mashup. The powerset lattice constructed from the set of origins in the mashup forms the set of permissible pathways in which information can flow in the mashup. Beyond the permissible pathways, if further declassification [94] amongst lattice elements is desired, escape-hatches [95] can be described on a per origin basis.

When the mashup is loaded, nodes in the DOM-tree are labelled with the origin of the nodes. After this, validity of information flow between the nodes is decided based on the subset relation between sets of labels on the nodes. Disregarding any escape-hatches, information can flow a source node to a sink node if the privilege level of the sink is at least as restrictive as that of the source. This is typically decided by checking whether the label of the source is a subset of the level of the sink. In the lattice world of things, this translates into checking whether the label representing the sink node is ordered higher than the label representing the source node in the lattice. This is how the powerset lattice derived from the set of all origins in the mashup forms a basis for governing flow of information within the mashup.

Figure 5.2 shows powerset lattices for mashups with information sourced from multiple origins. In figure 5.2a, we have the bottom (denoted by $\perp$) element of the lattice; this represents the lowest privilege level in the security lattice. Nodes in the DOM-tree sourced from origin `A` are marked with the lattice level `A`. The arrow from `A` to $\perp$ denotes that `A` is at a higher privilege level compared to $\perp$ and in the confidentiality world of things, flow of information from `A` to $\perp$ is not permitted. Similarly, content that is sourced from both `A` and `B` is annotated with the lattice level `A,B` and as shown in the powerset lattice in figure 5.2a, anything annotated with `A,B` is at a higher privilege level that either `A` or `B`. So, flow of information from `A` and `B` to `A,B` is secure but not the other way round. Finally, the dotted line in figure 5.2a represents an escape-hatch which enables information to flow from nodes whose content labeled `B` to nodes labeled with `A`. Even though there is pre-existing privilege ordering between `A` and `B`, the escape-hatch renders `A` at a higher privilege level



(a) Powerset lattice of 2 origins with declassification



(b) Powerset lattice of 3 origins

Figure 5.2: Powerset lattices for mashup security

than `B` (escape-hatches are declared on a per-origin basis). Figure 5.2b represents the powerset lattice for a mashup with 3 origins and without any declassification.

Without loss of generality, we ignore the presence of escape-hatches in this work. The reasoning to support this decision stems from the fact that escape-hatches are defined on an ad-hoc and per-origin basis and declassification through them is treated orthogonally to the flow through the powerset lattice. To quote a definition of a valid flow from [71]:

Definition 12. *For an expression e of level l_{source}, a legal flow for e to the target level l_{target} is allowed if $l_{source} \sqsubseteq l_{target} \sqcap \mathbf{D}$ where $\mathbf{D}$ is the set of origins to which the expression can be explicitly declassified through definition of escape-hatches on a per-origin basis.*

It can be observed from the definition, there are two parts to checking a valid flow: through the powerset lattice (by checking $l_{source} \sqsubseteq l_{target}$) and through declassification (by enumerating the elements of $\mathbf{D}$). The two are orthogonal to each other. Since, we are solely focused on lattices in this work, we turn our attention to speeding up the former computation. However, it must be noted that if the escape-hatches are modelled as edges in the DAG representation of the lattice and subsequently, enumerate further channels for valid information flow, any lattice pre-processing algorithm can still deal with them. We present experimental results related to random DAGs in section 5.5 that supports this point.

5.3 Algorithmic options for pre-processing lattices

In section 5.2, we discussed the two emerging examples of security lattices that occur in practice. However, the two examples showed the wide structural variety in the lattices that information flow analysis has to deal with. In this section, we discuss the algorithmic options available to pre-processing lattices in order to answer queries like $\sqcup$, $\sqcap$ and $\leq$ on two elements in the lattice in constant time. As discussed in chapter 3, such queries are necessary for information flow analysis. The objective of this section is to identify pre-processing algorithms that can deal with such a wide structural spectrum of lattices.

5.3.1 Tree algorithms for pre-processing lattices

Lattices and trees are structurally different entities; even though a tree can be considered an upper semilattice, there is no vertex in a tree that represents the join of two vertices in the tree. However, for the sake of studying algorithmic approaches for answering $\sqcup$ $\sqcap$ and $\leq$ queries on class hierarchies, it is imperative to consider approaches to answer such queries for trees. Multiple inheritance in class hierarchies is quite rare. In the absence of multiple inheritance, the class hierarchy based policy lattice as described for class-level non-interference in section 5.2 is nothing but a tree with a bottom ($\perp$) element. Hence, pre-processing algorithms for trees to answer $\leq$ and $\sqcup$ can often be straightforwardly applied to security lattices. In such cases, the $\sqcup$ ($\leq$) query for two elements in the lattice is obtained by performing a lowest common ancestor (reachability) query for the corresponding vertices in the tree that is obtained by ignoring the bottom element. On the other hand, the $\sqcap$ of two elements in a lattice derived from a class hierarchy is the $\perp$ element unless one is less than the other (assuming the absence of multiple inheritance).

If the ordering among the elements of the lattice is structurally similar to a tree, as is the case with class hierarchies, then pre-processing the lattice to answer subtyping queries is fairly straightforward. Interval-range labeling, where each node is annotated with a range representing the subtree rooted at that node [46], enables answering $\leq$ queries on the lattice in constant time after O(n) pre-processing for annotating the tree. Answering the $\sqcup$ and its dual the $\sqcap$ queries requires additional pre-processing but at the same asymptotic cost. Pre-processing the lattice in order to answer these queries consists of first performing an Euler tour of the graph [35]. The $\sqcup$ of two elements in the lattice lies within the first occurrence of the elements in the Euler tour of the tree representing the lattice. The identification of the exact element that is the $\sqcup$ is achieved by matrix lookups which are themselves pre-computed after the Euler tour has been completed. Both the Euler tour and pre-computation of the matrices can be completed in O(n) time and the $\sqcup$ query can be answered in constant time.

5.3.2 DAGs algorithms for pre-processing lattices

While the pre-processing algorithm for answering queries on tree-like lattices is fast and elegant, it is too restrictive to be extended to lattices in general. A lattice in general is typically represented as a DAG; DAGs allow arbitrary ordering amongst elements of the lattice and are a super-type of lattices. Answering $\leq$ $\sqcup$ and $\sqcap$ queries for lattices has, respectively, the same costs as answering reachability, lowest common ancestor (LCA) and highest common descendant (HCD) queries on a DAG representing the lattice. Therefore, in this section, we discuss the computational costs for answering these queries in DAGs. In the subsequent part of this section, we say lattice for the DAG representation of the lattice and $\leq$, $\sqcup$ and $\sqcap$ for the reachability, LCA, HCD queries on the DAG representation of the lattice, respectively. Consequently, elements in a lattice now correspond to vertices in the DAG representation of the lattice.

For a lattice, a $\leq$ query on the lattice can easily be answered in constant time after performing a transitive closure of the lattice. If we assume that the lattice has k elements, this can be done in $O(k^{\omega})$ where ω is the exponent of the fastest matrix multiplication algorithm [33, 115]. However, answering $\sqcup$ and $\sqcap$ queries required us to pick one element from the common ancestors/descendants of the query arguments. Since, $\sqcup$ is the dual of $\sqcap$ we discuss the cost for $\sqcup$ only. Once we have a reachability matrix through the transitive closure of the lattice, we can tell all ancestors (and descendants) for all elements in the lattice.

For a pair of elements, if we need to calculate the $\sqcup$ we need to identify the common ancestors first and then pick one (which is unique in case of lattices) with the highest topological number. An elegant solution for pre-computing $\sqcup$ for all pair of vertices in a DAG has been discussed in [36]. It was shown in [36] that picking $\sqcup$ for all pairs of vertices in a DAG can be reduced to picking maximal witnesses in the boolean matrix product of the reachability matrix of the DAG with itself. The cost for doing this was shown to be $O(k^{(2+\theta)})$ where θ satisfies relation $\omega_{1,\theta,1} = 1 + 2\theta$. Here, $\omega_{1,\theta,1}$ is the exponent of the multiplication of a $k \times k^{\theta}$ matrix with a $k^{\theta} \times k$ matrix. After obtaining a cost for $\omega_{1,\theta,1}$ in terms of ω and θ, it was shown that picking maximal witnesses and hence, identifying $\sqcup$ for all pairs of vertices in a DAG has a pre-processing cost of $O(k^{2.575})$ [36]. Thus, for a lattice with k elements, it is possible to answer $\sqcup$ queries for all pairs in constant time after $O(k^{2.575})$ preprocessing.

5.4 An adaptive framework for pre-processing lattices

While representing lattices as DAGs is general enough to be extended to any lattice, it is not particularly efficient for sparse lattices which are similar in structure to a tree with a few non-tree edges thrown in. Indeed for such lattices one would hope to be able to use tree algorithms for most of the lattice queries with the non-tree edegs accounted for separately while computing joins, meets and ordering queries. Such an adaptive approach to pre-processing of rooted DAGs has been proposed in chapter 4. The novelty of the approach lies in decomposing a DAG into clusters which are parts of the DAG connected using only tree edges. The transitive closure for the adaptive approach is computed at the level of clusters rather than at the level of individual vertices in the DAG. This significantly reduces the computational costs for preprocessing the DAG in order to answer the representative lowest common ancestor of a pair of vertices in the DAG in constant time. An overview of the cluster-based approach to pre-processing the DAG is shown in figure 5.3. The figure shows a DAG decomposed into clusters; the clusters have been demarcated using dotted lines. Edges for the spanning tree covering the DAG have been shown using solid lines while cross edges are denoted using dashed lines.

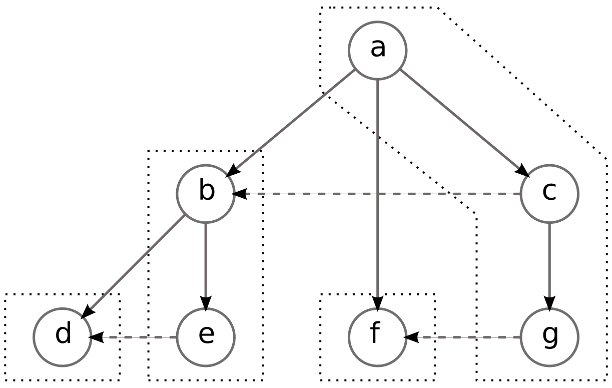


Figure 5.3: Overview of cluster-based preprocessing for DAGs

Since the representative lowest common ancestor (join or $\sqcup$ in case of a lattice) of two vertices in a DAG is a common ancestor of both such that none of its descendants is a common ancestor of the two vertices, conventional techniques for pre-processing a DAG rely on the reachability matrix of the DAG as a starting point for pre-processing the DAG. Therefore, the manner in which the transitive closure of the DAG is encoded has a direct bearing on the runtime of the pre-processing algorithm. Table 5.1a shows the reachability matrix for the DAG shown in figure 5.3. The adaptive approach to pre-processing the DAG takes a departure from such an encoding of the transitive closure of the DAG. Instead, it only stores the reachability information from one cluster to another via cross edges as shown in table 5.1b.

	a	b	c	d	e	f	g
a	1						
b	1	1	1				
c	1		1				
d	1	1	1	1	1		
e	1	1	1		1		
f	1		1			1	1
g	1		1				1

(a) Full closure

	c	e	g
b	1		
d	1	1	
f			1

(b) Cluster-based closure

Table 5.1: A compact approach to encoding transitive closure of DAGs

If two vertices belong to the same cluster, then their lowest common ancestor (LCA) is found using a pre-processing algorithm catered for trees. This is called the TT-PLCA : a candidate/potential LCA (PLCA) which reaches the both the vertices purely through edges of the spanning tree. If the vertices belong to two different clusters, then two other PLCAs are identified: the TC-PLCA/CT-PLCA and the CC-PLCA. The TC-PLCA/CT-PLCA is a candidate LCA such that the LCA reaches one of the vertices through tree edges only and the other through a combination of tree and cross edges. CC-PLCA, on the other hand, is a candidate LCA that reaches both the query vertices through a combination of tree and cross edges. The vertex that has the lowest topological number amongst the candidate LCAs is chosen as the result of the LCA query.

The algorithmic details for computing the LCAs of vertices is described in chapter 4. The costs for pre-processing the DAG for obtaining the TT-PLCA are based on standard approaches to computing the LCA of vertices in a tree. The TT-PLCA can be computed in constant time after a $O(n)$ preprocessing. On the other hand, the cross-edge based reachability matrix that is necessary for computing the TC-PLCA and CC-PLCA is itself constructed by computing the transitive closure of the cross-edge based adjacency matrix. This takes $O(c^{\omega})$ where ω is as decribed in section 5.3 and $c = max\{n_s, n_t\}$ where n_s and n_t are the number of vertices in the DAG with incoming or outgoing cross-edges respectively. On top of computing the transitive closure of the cross-edge adjacency matrix, it is necessary to pick CC-PLCAs, which requires performing a maximal witness of boolean matrix product on the cross-edge based reachability matrix. This operation is analogous to that described it section 5.3 and the cost for it is $O(c^{2.575})$.

A clear advantage of the cluster-based approach to pre-processing the lattice is that the algorithm seamlessly adapts to the incidence of cross-edges in the DAG/lattice. However, a subtlety lies in the query times for pre-processing based on a full closure as shown in table 5.1a and a cluster based approach as shown in 5.1b. The full-closure approach can answer lowest common ancestors queries by performing a simple table lookup in memory. On the other hand, the cluster-based approach needs to computer three PLCAs and identify the one with the lowest topological number amongst them. Even identifying the PLCAs is not a simple lookup operation. As shown in chapter 4, one needs to obtain indices for indexing the TT-PLCA and CC-PLCA matrices first. Furthermore, the TT-PLCA is identified by performing subsequent LCA operations on the spanning tree. These additional steps make the query more expensive compared to a full-closure based approach. In section **??**, we take a closer look at the pre-processing and query times for both the full closure based pre-processing algorithm and the cluster-based pre-processing algorithm, in order to establish the relative merits of the cluster-based approach.

5.5 Experiments

We ran experiments for measuring the preprocessing times and query times for both class hierarchies of large-scale commonly-used softwares and powerset lattices for both the full closure approach to pre-processing lattices and the cluster based approach. Additionally we also tested these algorithms against custom generated DAGs. These tests show the seamless adaptability of the cluster-based approach. We describe the experimental setup and the yardsticks we use to compare the cluster based algorithm to the full closure algorithm in section 5.5.1. Then, we describe our choices of benchmarks for the experiment in section 5.5.2. The results for class hierarchies as well as powerset lattices are discussed in section 5.5.3. After showing the suitability of the cluster-based $\sqcup$ pre-processing algorithm for class hierarchies and powerset lattices, we also show how the algortihm adapts seamlessly to the full spectrum of lattices (based on the incidence of non-tree edges) in section 5.5.3.

5.5.1 Setup

All experiments were conducted on a 2.26 GHz Intel Core 2 Duo processor with a 3MB shared cache and 8GB of memory. The system was running OS X Mavericks (10.9.3) operating system. All reported timing measurements are average wall-clock times. For each of the test cases, we measured the following:

- The total number of vertices in the lattice. In table 5.3, this is denoted as `total` for class hierarchies and 2^n for powerset lattices.
- The fraction of vertices with incoming or outgoing cross edges which we denote as `c_rat`. Note that as described in section 5.4, we take the larger of n_s (number of vertices with outgoing cross edges) or n_t (number of vertices with incoming cross edges) to derive `c_rat`. In particular, `c_rat` $= max\{n_s, n_t\} \div n$ where n is the total number of elements in the lattice.
- The time (in milliseconds) for preprocessing the lattice in order to answer $\sqcup$ queries in constant time for a pair of elements. This is denoted as `ppt_cc` for the cluster based closure algorithm and `ppt_fc` for the pre-processing algorithm based on full closure of the lattice.
- The time (in microseconds) taken to query the lattice for obtaining $\sqcup$ of two elements. This is denoted as `qt_cc` and `qt_fc` in table for the cluster-based algorithm and the algorithm which relies on a full closure of the lattice.
- Lastly, we also report two ratios which are indicative of the comparative performance of the algorithms. These ratios called `ppt_rat` and `qt_rat` are, respectively, the ratios of the pre-processing and query times of the cluster-based approach to the full closure approach.

5.5.2 Benchmarks

For testing the suitability of the cluster-based algorithm, we extracted the class hierarchies of six large, commonly-used programs. These programs are listed in table 5.2. They are also a part of the commonly used DaCapo benchmark suite used for Java benchmarking by the programming language community [22]. However, it must be clarified that we have tried to use newer, full-fledged versions of the software as opposed to the abridged ones that come with the DaCapo benchmark suite.

For testing out mashup security which is driven by powerset lattices, we generated powerset lattices based on the number of origins in a mashup. The powerset lattice is obtained by obtaining the powerset of all origins in the mashup and ordering the individual sets by the inclusion relation. As shown in table 5.3b, we have varied the number of components of the mashup (denoted by n) from 2 to 8. This gives us powersets with cardinality (denoted by 2^n) ranging from 4 to 256.

Program	Version	Description
Tomcat	5.5	web server and servlet container
Batik	1.7	manipulation of images in SVG format
Eclipse	Indigo	Integrated Development Environment
Jython	2.5.1	Python interpreter in Java
Pmd	5.1.0	Java source code analyser
Xalan	2.7.1	XSLT processor for XML documents

Table 5.2: Benchmark descriptions for class hierarchy testing

Finally, we demonstrate seamless adaptability of the proposed cluster-based pre-processing algorithm with randomly generated DAGs. In order to do so, we first generate binary trees with a specified depth and then add additional cross-edges to the tree to obtain DAGs with varying fraction of vertices that have an incoming or outgoing cross edge. As shown in figure 5.5a, we generate binary trees that range in depth from 4 levels to 8 levels. From each of these trees, we generate random DAGs where the fraction of vertices with incoming or outgoing cross edges varies from 0.1 to 0.5.

5.5.3 Results

Pre-processing and query times for the class hierarchies is shown in table 5.3a. Here, `total` refers to the number of classes in the hierarchy. It can be observed that the cluster based approach to pre-processing the class hierarchy is significantly better than an approach based on a full closure. This is because the full closure approach relies on maximal witness of a boolean matrix product to precompute the $\sqcup$ for all vertex pairs. This can be an expensive operation if the adjacency matrix for the lattice is large. Also, it can be observed from table 5.3a that the query times for both the cluster based approach and the full closure approach are pretty much constant for all the test cases. However, the query time for the cluster based approach is slightly more expensive. This is because the query for the full closure based approach does a simple matrix lookup to obtain the $\sqcup$ of two elements. In case of the cluster-based approach, additional processing needs to be done in order to obtain candidate $\sqcup$s. Then amongst the candidate $\sqcup$s, one with the highest topological number needs to be selected as described in section 5.4. Even though the cluster based approach is slower when it comes to query times, the gains during the pre-processing phase far outweigh the increased query time. In other words, one can query the lattice for $\sqcup$ many more times in the time saved during pre-processing. The number of additonal queries can be done from the gains in pre-processing time realised from choosing an adaptive pre-

Software	vertices	c_rat	ppt_cc(ms)	ppt_fc(ms)	ppt_rat	qt_cc(μs)	qt_fc(μs)	qt_rat
Tomcat	636	0.00	1.48	14480.35	1E-4	0.98	0.68	1.44
Batik	232	0.00	0.66	653.36	1E-3	0.96	0.66	1.44
Eclipse	2200	0.00	5.08	644337.95	8E-6	0.97	0.67	1.45
Jython	344	0.00	1.00	2235.09	4E-4	0.96	0.67	1.44
Pmd	1053	0.00	2.67	66656.00	4E-5	0.96	0.67	1.44
Xalan	821	0.00	2.07	31441.23	7E-5	0.97	0.66	1.46

(a) Class Hierarchies

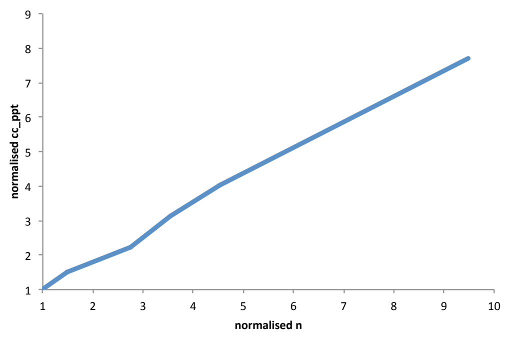
n	2^n	c_rat	ppt_cc(ms)	ppt_fc(ms)	ppt_rat	qt_cc(μs)	qt_fc(μs)	qt_rat
2	4	0.25	0.06	0.02	3.39	1.63	0.62	2.60
3	8	0.50	0.14	0.09	1.68	1.48	0.70	2.11
4	16	0.69	0.65	0.37	1.73	1.48	0.65	2.27
5	32	0.81	3.21	2.22	1.45	1.56	0.72	2.18
6	64	0.89	17.86	14.24	1.25	1.56	0.69	2.26
7	128	0.94	93.64	106.81	0.88	1.57	0.67	2.33
8	256	0.96	586.85	968.18	0.61	1.62	0.69	2.36

(b) Powerset lattices

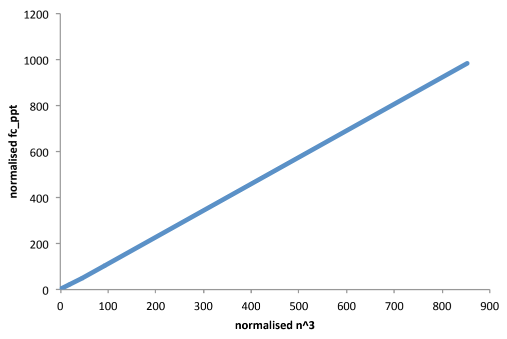
Table 5.3: Preprocessing and query times for powerset lattices and class hierarchies

processing approach is entirely dependent on the size of the lattice under consideration and the incidence of cross edges. For example, for a small powerset lattice ($n \leq 4$), one is no better off by using the adaptive approach. On the other end of the spectrum, one can fit in over 10^6 queries for large class hierarchies such as the one for Eclipse.

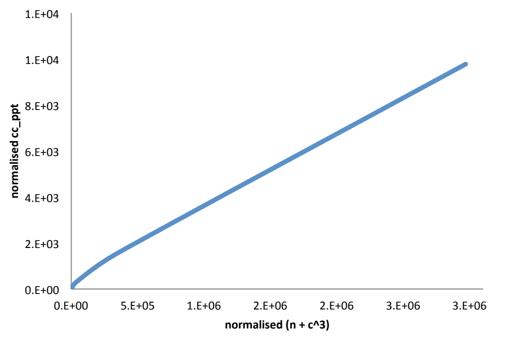
Pre-processing times and query times for the powerset lattices is presented in table 5.3b. Here, n is the number of components in the mashup and hence, 2^n is the number of vertices in the powerset lattice for controlling information flow in the mashup. The fraction of vertices with incoming/outgoing cross edges increases steadily with the increase in the number of components in the mashup. This is reflected in the `c_rat` values. The choice of the number of components for our experiments covers a large range of `c_rat` values starting from 0.25 for mashups with 2 components to 0.96 for mashups with 8 components. It can be observed from table 5.3b that for smaller mashups, the cluster based preprocessing algorithm is slower than the full closure based preprocessing algorithm. This can be explained through the observation that the fixed costs of preprocessing the lattice (like identifying clusters, computing their closure and labeling the lattice to index this closure) outweigh the benefits of a cluster-based approach for small lattices. Indeed, since the full closure based algorithm merely computes a few matrix multiplications to preprocess the lattice, it runs faster than the cluster-based approach for small lattices. However, as the size of the powerset lattice increases, matrix multiplications start getting expensive and



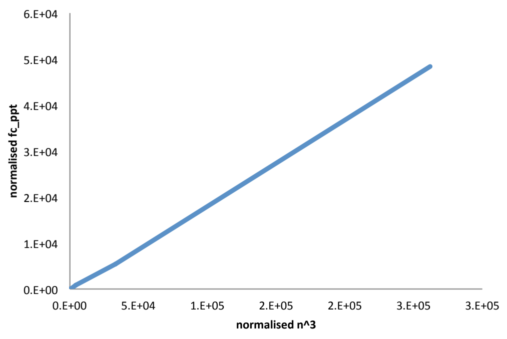
(a) Class hierarchy: partial closure



(b) Class hierarchy: full closure



(c) Powerset: partial closure



(d) Powerset: full closure

Figure 5.4: Plots showing the correlation between theoretical complexity and practical computational costs

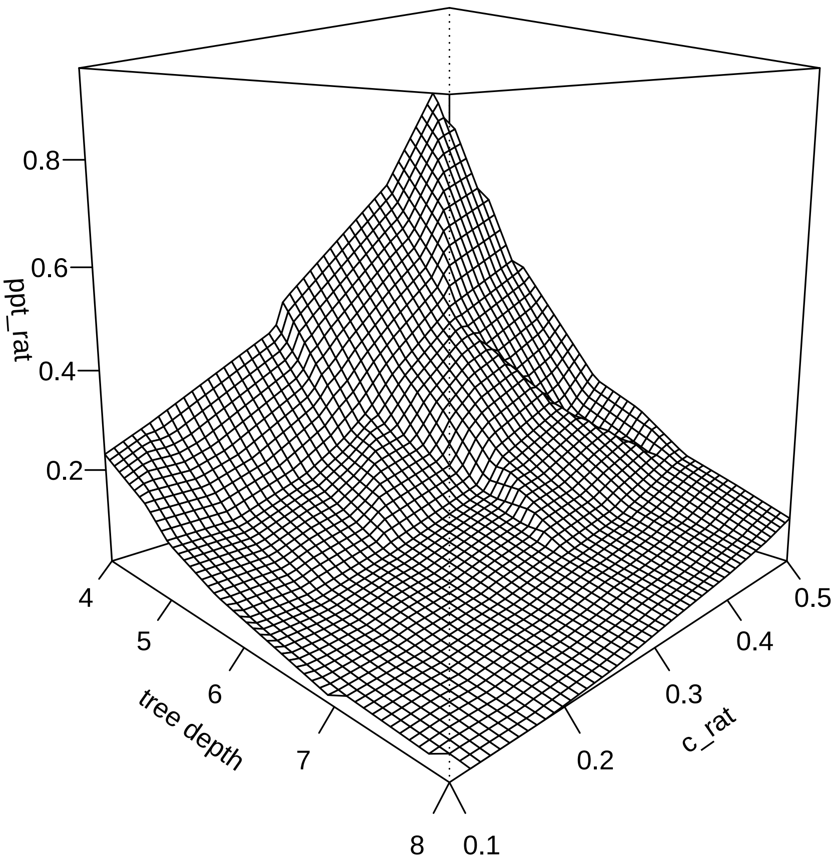
the full closure based approach starts to trail behind the cluster-based approach. The query times in the case of powerset lattices also follow a similar trend to class hierarchies with the query time for cluster-based approach being more expensive than the query time for full-closure approach. However, it is noteworthy that the query times for class hierarchies is lesser than the query times for powersets in the cluster based approach. This can be explained by observing that in class hierarchies, there are no cluster closures to enumerate; all the class hierarchies are trees. This significantly reduced the overhead to compute candidate $\sqcup$s arising from cluster closures and reduces the total query time significantly.

There exists a strong correlation between the theoretical and practical costs for the results presented in this section. In figure 5.4, we have plotted the growth in the actual computational costs with the growth in the theoretical computational costs as the graphs under consideration grow larger. It is to be noted here that we use a plain vanilla matrix-multiplication/transitive-closure based algorithm for our implementation which gives us a cost of $O(n^3)$. It is to be noted that there is a faster algorithm which has a theoretical cost of $O(n^{2.575})$ for pre-computing all-pair lowest common ancestors using an $n \times n$ adjacency matrix for a DAG of size n. However, we choose a simpler implementation for our purpose which has a slightly higher asymptotic cost.

There exists a linear relationship between the normalised theoretical costs and the normalised experimental costs for both the class hierarchies and powerset lattices. Figures 5.4a and 5.4b, denote this correlation for the partial closure (cluster-based approach) and the full closure respectively. Since the class hierarchy does not contain any cross edges, we normalise the experimental costs for $O(n)$ instead of the actual theoretical cost of $(O(n + c^3))$ for the cluster-based approach. For the algorithm based on full closure we show the growth in experimental costs compared to the growth in experimental costs for an $O(n^3)$ algorithm. Similarly, for the powerset lattices, the costs for the cluster-based algorithm and the full-closure based algorithm have a linear relationship with $O(n + c^3)$ and $O(n^3)$ respectively as shown in figures 5.4c and 5.4d. These results demonstrate that our theoretical assessments of computational costs go hand-in-hand with the actual experimental measurements.

The potential benefits from the cluster-based approach shines through when the `c_rat` is well below 1.0. As `c_rat` approaches 1.0, the runtimes for the cluster-based approach will start to approach that of the full closure based algforithm. However, the true benefit of the cluster-based algorithm lies in the fact that the pre-processing time of the algorithm seamlessly adjusts itself based on the incidence of vertices with incoming/outgoing cross edges; but, it is seldom worse off than the full-closure based algorithm. This is evidenced on our experiments with random DAGs as shown in figure 5.5. In figure 5.5a, the axes of tree depth and `c_rat` together define the structure of the DAG under consideration. For example, a DAG with a tree depth of 5 and a `c_rat` of 0.3 means that the DAG is actually a binary tree of depth 5 (63 vertices) superimposed with cross-edges such that $max\{n_s, n_t\} \div n = 0.3$. More information about n_s, n_t and n for randomly generated DAGs can be found in section 5.5.1. Some sample datapoints related to figure 5.5a that reinforce the subsequent discussion of the results are given in table 5.5b.

Two major trends emerge from figure 5.5a. First, it can be seen that for a given tree depth, the higher the `c_rat` values, the lesser the gain in pre-processing time using the cluster based approach. This can be explained by considering the asymptotic costs of the cluster-based approach which is $O(n + c^{2.575})$ and that of the full closure approach which is $O(n^{2.575})$ where n is the total number of vertices in the DAG and c is the total



(a) Pre-processing speedups for full spectrum of lattices

vertices	c_rat	ppt_cc(ms)	ppt_fc(ms)	ppt_rat	qt_cc(μs)	qt_fc(μs)	qt_rat
15	0.07	0.06	0.32	0.20	1.49	0.66	2.26
15	0.48	0.37	0.35	1.07	1.49	0.69	2.23
63	0.11	0.43	12.81	0.05	1.60	0.69	2.39
63	0.52	4.92	14.89	0.34	1.52	0.68	2.30
255	0.11	4.13	909.55	0.01	1.60	0.69	2.40
255	0.52	112.03	1050.08	0.11	1.56	0.70	2.31

(b) Sample datapoints

Figure 5.5: Experimental results for random DAGs

number of vertices with incoming or outgoing cross edges. As can be expected, as the `c_rat` increases, the number of vertices with incoming or outgoing cross edges increases as well. Therefore, the costs of the cluster-based algorithm tends to approach that of the full closure algorithm.

The second trend that is noteworthy in figure 5.5a is that for a given `c_rat` value, the smaller DAGs (i.e. the ones which have a lower depth) have a lesser reduction in the pre-processing costs using the cluster-based approach. This is to be expected because the full closure approach uses boolean matrix product for pre-processing the DAGs. So, for smaller DAGs, the fixed costs for identifying clusters combined with the ease of doing small matrix multiplications (as necessitated by the full closure approach) dampen the gains that could arise from a cluster-based approach. Thus, the maximum gains from a cluster-based approach can be realised when the size of the DAG is large and the proportion of vertices with incoming or outgoing cross edges is small. These observations are corroborated by the results reported in table 5.3. In table 5.3b, the preprocessing time for Eclipse is much faster using a cluster based approach as opposed to a full closure based approach. This is because the class hierarchy is both large and has no cross edges in it. On the other end of the spectrum, for small powerset lattices with a high `c_rat` ratio as shown in table 5.3b, there is little to gain from a cluster based approach. However, it must be noted that the cluster-based approach is seldom significantly worse than the full closure approach.

5.6 Summary

In this chapter we discussed a wide spectrum of security lattices that govern information flow in programs. We showed that the structure of these lattices range from trees in the case of class hierarchies all the way to dense DAGs in the case of powerset lattices. In view of this, we discussed why an adaptive algorithm which is sensitive to the structure of the lattice under considerations is a superior approach to pre-processing these lattices. We demonstrated how clusters can be useful in adapting an algorithm to the structure of the lattice under consideration. While it was initially thought that querying the pre-processed lattice in the cluster-based approach was slower compared to a full-closure based approach, we experimentally showed that the slowdown per query was much smaller when compared to the gains in pre-processing times when a clusters-based approach was adopted. We ran extensive experiments on a wide range of security lattices to demonstrate the viability of a cluster-based approach to lattice pre-processing and showed how the algorithm scales seamlessly across a large variety of lattices encountered in practice. While this work has been developed with information flow in mind, it can be easily extended to other areas of program analysis which use subtyping in conjunction with a partial order that dictates the subtyping policy.

Chapter 6

Adaptive simplification of polymorphic flow constraints

6.1 Introduction

Annotated type systems enforce a pre-defined partial order amongst the annotated labels, and aid in a variety of program analysis paradigms like alias analysis [41, 16], security [117, 87, 78] and resource reasoning [56]. This approach to program analysis is commonly known in the literature [91] as polymorphic subtyping on labels . In such systems, the programmer annotates only parts of the program with labels, and for all unannotated terms the label is represented as a variable. Then, a most general value for the label variables is inferred, while respecting the programmer-specified partial ordering amongst labels which needs to be enforced as well as the flow in the program. This process is known as label inference. Constraints on label variables in polymorphic subtyping systems are of two kinds: they can be an ordering relation with another label variable derived from the program flow, or they can be an ordering with a concrete programmer-annotated label. The presence of programmer-annotated labels aids in deriving concrete bounds for label variables and helps in determining their most general value.

There are scenarios, however, when the programmer annotation is completely missing for a term. Such terms are called label-polymorphic. Values for label variables in label-polymorphic terms are context-sensitive and can only be derived if a the term is put in context with other programmer-annotated terms. In the face of label-polymorphism, it is necessary that any constraint that is derived for label variables from the data and/or control flow in an expression is represented in as compact a manner as possible. This avoids duplication of unnecessary constraints when the term is put in multiple contexts and also makes long sets of constraints easier to read.

The first step in compaction is to represent the set of label-variables and the constraints on them as a directed graph. It is noteworthy that in the absence of programmer annotations, constraints on label variables are only due to the program flow, and a directed graph is a natural means to representing inter-relationships between label variables. This is analogous to data flow graphs in other forms of program analysis. In the second stage of compaction, only direct or transitive relationships between input and output variables are preserved as valid constraints. This is sufficient to check whether the label-polymorphic term violates the pre-defined ordering on labels when it is put in context. The difficulty in such an approach is that size of the label-constraints graph is dependent on the term under consideration. For large expressions, the number of label variables (vertices) in the term (label-constraint graph) and the number of constraints (edges) on these variables could be non-trivial.

In this chapter, we revisit the issue of label-constraint simplification from a graph theoretic perspective. Our approach is inspired by the recent advances in the theory of transitive closure for directed graphs, and we discuss a novel algorithm for compaction based on decomposition of the label-constraints graph. In a significant departure from pre-existing work, the runtime of the proposed algorithm is directly related to the stucture of the label-constraints graph, and the algorithm adapts to the structure of the graph. The decomposition-based approach provides a level of abstraction, and the asymptotic costs of our algorithm are a function of the latent scope for decomposition in the graph. If the scope for decomposition is good, there is a potential for significant speedup. On the flip side, we establish through experimental evaluation that even if there is little scope for decomposition, the proposed algorithm is no worse off that a baseline algorithm for compaction. The experiments are performed by applying the proposed algorithm to label-constraint graphs encountered in language-based security which tend to have a poor scope for decomposition.

The rest of the chapter is organised as follows. We give an example of a label-polymorphic expression in section 6.2. We discuss how label relationships in a label-polymorphic expression are suitably represented as a directed acyclic graph (DAG). We also present a baseline algorithm for compacting the label-relationship graph in this section. We present an overview of the approach in section 6.3 and present the algorithm for a decomposition-based approach as well as an analysis of its computational costs. In section 6.4, we stress test the proposed algorithm by applying it to graphs encountered in language based security, to demonstrate that even in the face of graphs that have little scope for decomposition, the algorithm has comparable performance to a baseline algorithm. Finally, we summarise the contributions of this chapter in section 6.5.

6.2 Compaction of label-polymorphic expressions

In this section, we discuss how to compact the constraints on labels in label-polymorphic expressions. We discuss how label-constraint graphs arise from a label-polymorphic expression in section 6.2.1. We then discuss a closure based algorithm to compact label-constraint graphs in section 6.2.2.

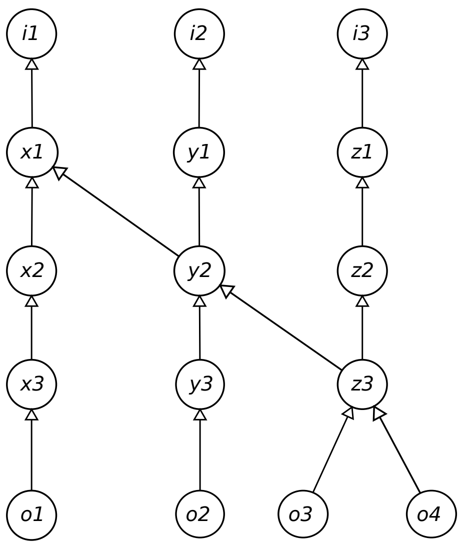
6.2.1 From expressions to DAGs

We first describe how constraints on labels in label-polymorphic expressions can be represented as a graph. Figure 6.1a shows an expression `foo` which takes a 3-tuple as an argument (denoted by $(\mathtt{i1}, \mathtt{i2}, \mathtt{i3})$ in ll. 1) and produces a 4-tuple (denoted by $(\mathtt{o1}, \mathtt{o2}, \mathtt{o3}, \mathtt{o4})$ in ll.15). $(\mathtt{o1}, \mathtt{o2}, \mathtt{o3}, \mathtt{o4})$ is produced from $(\mathtt{i1}, \mathtt{i2}, \mathtt{i3})$ in three steps (ll. 13-14). In the first step, $\mathtt{f}_{\sim\mathtt{12}}$ is applied to $(\mathtt{i1}, \mathtt{i2}, \mathtt{i3})$. Then, $\mathtt{f}_{\sim\mathtt{23}}$ is applied to the results of the first step. In the final step, $\mathtt{f}_{\mathtt{dup}}$ is applied to the results of step 2. $\mathtt{f}_{\sim\mathtt{12}}$ (ll. 7-8) and $\mathtt{f}_{\sim\mathtt{23}}$ (ll. 9-10) take in a 3-tuple argument and apply the function $\sim$ (ll. 2-6) to the first and second elements of the argument, and second and third elements of the argument, respectively. Here, π_i is a projection function for the i^{th} element of the tuple and $\sim$ is a trivial function that takes two arguments and produces a 2-tuple where the first element is the first argument to $\sim$ and the second element is the larger of the two arguments. $\mathtt{f}_{\mathtt{dup}}$ takes a 3-tuple and duplicates the third element of the tuple and produces a 4-tuple which is used to initialise $(\mathtt{o1}, \mathtt{o2}, \mathtt{o3}, \mathtt{o4})$.

The corresponding label constraint graph for `foo` is shown in figure 6.1b. The vertices in figure 6.1b denote label variables and the edges between the vertices denote constraints on the label variables derived from the program flow. Each label variable has the same name as the corresponding program variable except that it is written in italics. A constraint of the form $p \leq q$ is denoted in the graph with an arrow from q to p. We present the label-constraints graph in a concise form here for the sake of simplicity. In particular, we have ignored any intermediate constraints that might arise due to the projection, type constructors or other function definitions like $<$. However, inclusion of these constraints does not affect our discussion in any way. Inclusion of these constraints just introduces additional vertices and edges in the flow constraints graph, and techniques proposed in

```
let foo (i1, i2, i3) =
let $\sim$ x y =
   if x < y then
      (x, x)
   else
      (x, y)
and $f_{\sim 12}$ (x1, y1, z1) =
   ($\pi_1$($\sim$ x1 y1), $\pi_2$($\sim$ x1 y1), z1)
and $f_{\sim 23}$ (x2, y2, z2) =
   (x2, $\pi_1$($\sim$ y2 z2), $\pi_2$($\sim$ y2 z2))
and $f_{dup}$ (x3, y3, z3) =
   (x3, y3, z3, z3)
and (o1, o2, o3, o4) =
   $f_{dup}$ $f_{\sim 23}$ $f_{\sim 12}$ (i1, i2, i3)
in (o1, o2, o3, o4)
```

(a) Label-polymorphic expression



(b) Label constraints graph

Figure 6.1: A label-polymorphic expression and it's label constraints graph

this chapter will still be applicable notwithstanding. We just focus on constraints on labels introduced during function applications for a precise presentation of our approach.

Another important observation in the mapping from label-polymorphic expressions to label-constraint graphs is the absence of cycles in the graphs. Normally, in the simple case of a loop or a recursion, one would expect backward flow in the label-constraint graph, and an edge from a vertex to its ancestor would be sufficient to represent it. However, such a loop in the label-constraint graph renders the constraint unsolvable because of cyclical dependency between the label variables. In such a case, the common approach in most solvers is to fuse the labels that are part of the loop into a single vertex. The parents (children) of the fused vertex in the label-constraint graph are set as the union of the parents (children) of each vertex in the cycle. Thus, the label-constraints graph is ultimately represented as a directed acyclic graph (DAG).

6.2.2 A baseline algorithm

The traditional approach to compaction is shown in algorithm 6.1. The function COMPACT (ll. 11) takes a polymorphic expression e and its corresponding label-constraint graph g as parameters, and derives transitive flow relationships between input and output variables for the expression. The function first identifies all vertices in g corresponding to

Algorithm 6.1 Polarized garbage collection

```
1: function TRAVERSE($v$)
2:     if IS_OUTPUT($v$) then                      ▷ Check if $v$ is an output variable
3:         $v.o \leftarrow v.o \cup v$
4:     end if
5:     $\bar{c} \leftarrow v.children$
6:     for all $c \in \bar{c}$ do
7:         if NOT_TRAVERSED($c$) then              ▷ Traverse down if $c$ hasn't been visited
8:             TRAVERSE($c$)
9:         end if
10:        $v.o \leftarrow v.o \cup c.o$
11:    end for
12: end function
13: function COMPACT($e$, $g$)
14:     $\bar{i} \leftarrow$ VERTICES($e.inputs$, $g$)
15:     for all $i \in \bar{i}$ do
16:         TRAVERSE($i$)
17:     end for
18: end function
```

the input variables for the expression (ll. 14) and for each of the input variables, it calls the recursive function TRAVERSE which walks down the label-constraint graph starting at the argument passed to TRAVERSE (which we call v as shown in ll.1) and collects all the output variables reached by v. If the argument to TRAVERSE is itself an output variable, it is added to the list of output variables reachable from v (ll. 2-4). Additionally, for each of the children of v, the output variables reachable from the child are also added to the list of output variables reachable from v (ll.10). A predicate (ll. 7-9) ensures that TRAVERSE visits each vertex in the label-constraint graph only once; it checks whether the child c of a vertex has been traversed already before initiating the traversal at c. The overall complexity of this algorithm is $O(v_o(n+m))$ where v_o is the number of output variables in the polymorphic expression, n is the number of vertices in the label-constraint graph and m is the number of edges in the graph.

6.3 Simplification through decomposition

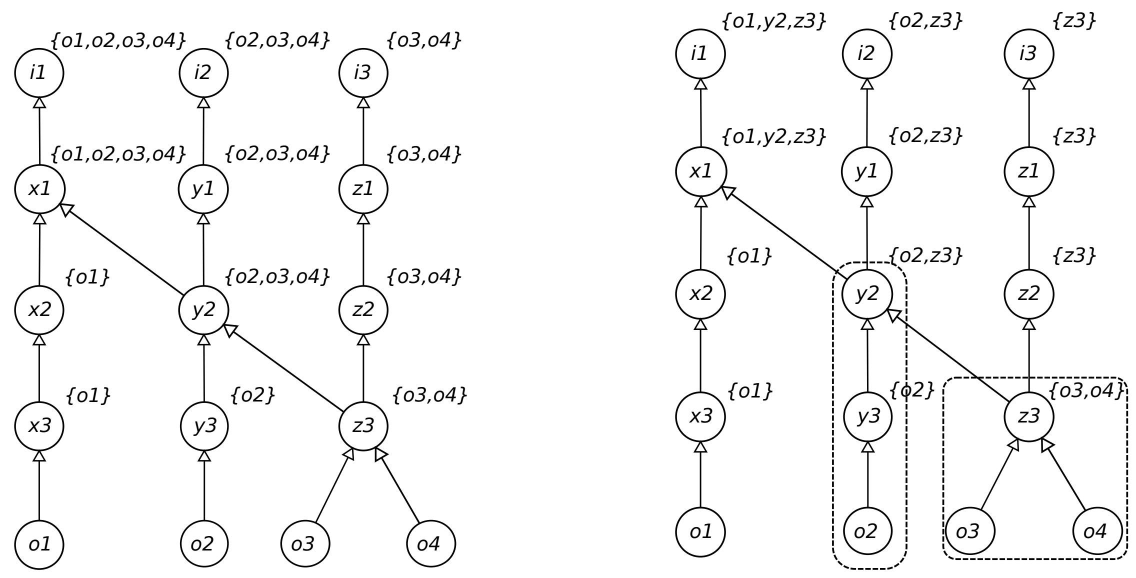
In this section, we discuss a cluster-based approach to performing the compaction of label constraints graph for label-polymorphic expressions. The advantage of doing so lies in the level of abstraction that we gain while reasoning about compaction. In section 6.3.1, we discuss an overview of our approach, and we discuss the algorithm for cluster-based formulation in section 6.3.2.

6.3.1 A *cluster* based approach

As noted previously in section 6.2.1, the label-constraint graph is a DAG, due to elimination of all loops (formally known as strongly connected components or SCCs in short) in the initial label-constraint graph obtained directly from the program flow. Loop elimination (by fusing vertices in the SCCs) is done by classifying edges in the label-constraint graph and identifying any edges from a vertex to its ancestor in the spanning tree covering the DAG [106]. This requires making the initial DAG rooted, by adding a root and making it the parent of all parentless vertices, and constructing a spanning tree for the resultant DAG. Therefore, a legacy of the loop elimination algorithm is a rooted DAG with a spanning tree, and all edges (not just those belonging to the spanning tree) that have already been classified. It is important to observe that loop elimination does not alter the classification of edges outside the SCCs; loop elimination preserves characteristics of all edges to (from) vertices external from (to) the SCC. The only difference is after elimination the SCC is represented as an aggregate vertex which is obtained after fusing all vertices in the SCC into a single vertex.

The proposed compaction algorithm builds on the existing edge classification performed by the loop elimination operation. For the subsequent discussions, we will assume that the DAG that is input to our algorithm is static and rooted, with edges that have already been classified into one of the three categories: tree edges, forward edges and cross edges [35]. Similar to chapter 4, we ignore the set of forward edges as they introduce a redundant connection between vertices that are already connected through tree edges and decompose the DAG into clusters and note the clusterheads for those clusters.

Due to the operations that are performed during the fusion of SCCs, we are able to identify clusters and associated clusterheads for free. The two advantages that clusters offer over a baseline algorithm are abstraction and constant-time union operation. These two advantages become obvious when we compare the baseline-algorithm with a cluster-based approach as shown in figure 6.2. The graph in figure 6.2a shows a traversal that collects output variables reached by any given vertex in the label-constraints graph. The output variables reached by a vertex are annotated alongside the vertex. On the other hand, figure 6.2b shows the collection strategy using clusters. The label-constraint graph contains two vertices that have an incoming cross-edge in addition to a tree edge; they are $y2$ and $z3$. Thus, these two vertices form clusterheads and the corresponding clusters are demarcated using a dashed line.



(a) Baseline collection (b) Collection using clusters

Figure 6.2: A comparison of the baseline algorithm vs cluster based approach

The first advantage of clusters is abstraction from the level of individual vertices to the level of disjoint sets of vertices (clusters). The only way to reach a vertex within a cluster is to go through its clusterhead. Therefore, if we just record the clusterheads that contain output variables (we call then *polar clusters*), we can always identify the list of reachable output variables from the set of reachable polar clusters. In figure 6.2b, since the clusters for clusterheads $y2$ and $z3$ both contain output variables, both of these clusters are polar clusters. Hence, we annotate $y2$ and $z3$ in addition to $o1$ as the set of output variables and polar clusters reachable from $x1$. As opposed to figure 6.2a, it can be observed from the annotations at each vertex in the cluster-based approach that the payload of the collection algorithm is reduced. This is due to the abstraction introduced through a cluster-based formulation for compaction.

A cluster-based approach seems similar to a collection algorithm that is based just on (overlapping) sets of vertices. However, the second advantage of clusters, namely constant time union operation makes it an attractive proposition when compared to the simple sets of vertices. Consider the case we could represent the entire set $\{y2, y3, o2, z3, o3, o4\}$ just with $y2$ as a representative since all of these vertices can only be reached through $y2$. However, in such a case, the union of sets of vertices does not remain as elegant as that for a cluster based formulation. Consider the hypothetical case where we have an additional edge (not shown) in the label-constraint graph of figure 6.2 with $z2$ reaching $y2$. In the (overlapping) set-based approach, we have $z2$ reaching the representative $y2$ (which stands for $\{y2, y3, o2, z3, o3, o4\}$) and the representative $z3$ (which stands for $\{z3, o3, o4\}$). The set of reachable vertices from $z3$ is now the union of sets reached by the two representatives. This is not a constant time operation. On the other hand, if we use a cluster based approach, we have $z3$ reaching two non-overlapping clusters headed by $y2$ and $z3$, the union of which is a constant time operation because clusters are standalone trees and disjoint with each other.

6.3.2 Algorithm and computational costs

Having described an overview of our approach, we now describe an algorithm for cluster-based compaction of the label-constraint graph for label-polymorphic expressions. The technique is described in algorithm 6.2. Like algorithm 6.1, the function COMPACT takes in a label-polymorphic expression and its associated label-constraint graph as inputs, and compacts the expression by deriving reachability information between the inputs and outputs of the expression. Unlike algorithm 6.1, it only collects polar clusters as it traverses down the label relationship graph.

Collection of output variables is done internally to the cluster; if the child c of a vertex v in the label relationship graph is a clusterhead, then only the polar clusters $c.clus$ which are reachable from c are added to the list of clusters reachable from v (ll. 5-9). Otherwise, both the reachable polar cluster and the reachable output variables that are reachable from c are added to $v.clus$ and $v.o$ respectively (ll. 15-20). Only polar clusters are copied over, to prevent unnecessary increase in the payload of the algorithm. A final difference compared to algorithm 6.1 lies at the end of the function COMPACT (ll. 27-29) where we collate all output variables for all polar clusters reachable from the input variable.

The computational cost of the algorithm is a function of the structure of the graph that is passed to it. This is evident from a representation of the computational costs as a function of the structure as shown in figure 6.3. If we assume that the number of output variables per cluster is p and the number of clusters with at least one output variable in it is Ω_p then the algorithm has an asymptotic cost of $O((\Omega_p + p) \times (n + m))$. Here, n is the number of label variables in the label-constraint graph and m is the number of edges in the label-constraint graph. If the number of cross edges is small i.e. the constraints graph is structurally similar to a tree, then Ω_p tends to approach zero and p tends to approach the total number of output variables (represented by Ω). In such a case, the computational cost of the cluster-based approach tends to match that of the baseline algorithm which is $O(\Omega \times (n + m))$. On the other hand, if there are too many cross edges,

Algorithm 6.2 Polarized garbage collection using clusters

```
function TRAVERSE($v$)
    if IS_OUTPUT(v) then                          ▷ Check if $v$ is an output variable
        $v.o \leftarrow v.o \cup v$
    end if
    if IS_CTAR($v$) then                          ▷ Check if $v$ is a cross-edge target
        if $v.o \neq \phi$ then                   ▷ Check if $v$ is a polar cluster
            $v.clus \leftarrow v.clus \cup v$
        end if
    end if
    $\overline{c} \leftarrow v.children$
    for all $c \in \overline{c}$ do
        if NOT_TRAVERSED($c$) then                ▷ Traverse down if $c$ hasn't been visited
            TRAVERSE($c$)
        end if
        if IS_CTAR($c$) then
            $v.clus \leftarrow v.clus \cup c.clus$          ▷ Collect polar clusters only
        else
            $v.clus \leftarrow v.clus \cup c.clus$          ▷ Collect polar clusters
            $v.o \leftarrow v.o \cup c.o$                   ▷ Collect output vars of current cluster
        end if
    end for
end function
function COMPACT($e$, $g$)
    $\overline{i} \leftarrow$ VERTICES($e.inputs$, $g$)
    for all $i \in \overline{i}$ do
        TRAVERSE($i$)
        for all $clus \in i.clus$ do
            $i.o \leftarrow i.o \cup clus.o$        ▷ Collect output vars from reachable polar clusters
        end for
    end for
end function
```

the level of abstraction uncovered by a cluster based approach is reduced and in this case, p approaches Ω and once again the computational cost of the algorithm approaches that of the baseline algorithm.

6.4 Stress testing the cluster-based approach

In this section, we test the cluster-based compaction algorithm on the standard library of the FlowCaml programming language [100]. FlowCaml is an extension of the ML

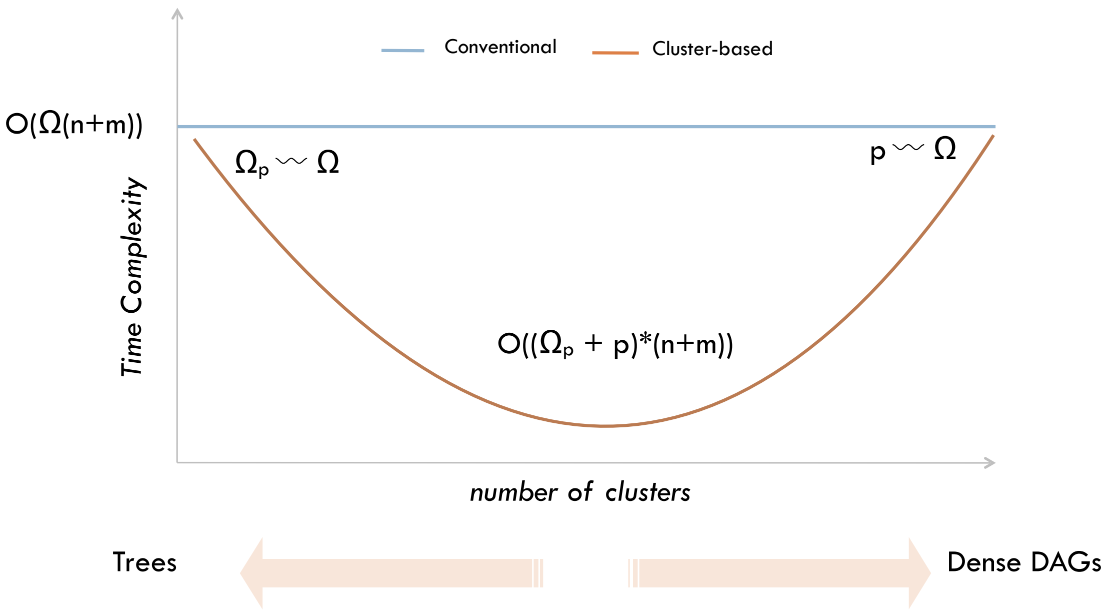


Figure 6.3: Computational costs as a function of the ratio of output variables to polar clusters

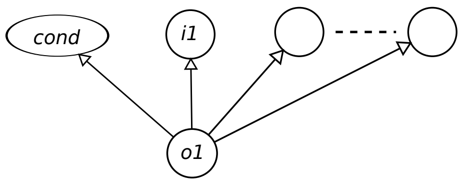
programming language with a type system that analyses information flow through the program. In FlowCaml, standard ML types can be annotated with security levels which describe the amount of information that the expression associated with the type holds. Through type inference, the type system of FlowCaml automatically infers security levels for unannotated expressions and checks whether the program obeys the security policy intended by the programmer. The security policy itself is typically described as a lattice of privilege levels that governs the flow of information.

6.4.1 Nature of constraints in information flow analysis

In section 6.2.1, we discussed how constraints from label polymorphic expressions are normally represented as DAGs. In type-based IFA, one needs to keep tabs on the flow of information through the control flow in addition to the data flow. As we will demonstrate shortly with an example, this introduces additional cross-edges in the label-constraint graph. As a consequence, the label constraint graphs fall in the right half of structure spectrum shown in figure 6.3. This is further corroborated in our experimental results presented in this section. Since performance of a cluster based approach tends to degrade in face of a large number of cross-edges, type-based IFA is a suitable benchmark to stress test the cluster based approach.

```
1 let foo i1 =
2    let o1 =
3       if (cond) then i1 else ...
4 in o1
```

(a) Label-polymorphic expression



(b) Label constraints graph

Figure 6.4: A label-polymorphic expression and its label constraints graph

We now show why type-based IFA introduces additional cross edges in the label-constraints graph of a label polymorphic expression. Figure 6.4 shows a simple expression which needs to be analysed to check flow of information flow through it. Type-based IFA is typically done to prohibit privileged data from being inadvertently declassified and written to a less secure location where it can be read by an unintended user. This involves getting insights into the flow of information through the control flow in addition to the data flow. For example, it is possible to guess the value of predicate in a conditional by observing the output of the conditional expression. Consider the expression in figure 6.4a. From the result of the conditional expression in ll.3, if we get to know only that the *true* branch is being taken or that the *false* branch is being taken, we can immediately guess the value of `cond` even though it does not contribute directly towards the computation of the result value. To account for such forms of information leakage, the usual practice is to maintain an aggregate of the privilege levels for all points in the control flow leading up to the current point in the program. This is in addition to the rudimentary constraints derived from the data flow as described in figure 6.4. Therefore, not only do we have a constraint on the label for `o1` from the label for `i1` as shown in figure 6.4b, we also have a constraint on the label for `o1` from the label for the predicate `cond`. Additionally, there could be more constraints on the label for $o2$ due to the *false* branch of the if statement which is shown by empty circles in figure 6.4b.

It should be noted how the introduction of an additional edge due to control flow has forced $o1$ to have multiple parents. The result of this is a cross-edge to $o1$ in addition to what would have been only a tree-edge to $o1$ had we considered only the data flow. Therefore, the label-constraints graphs in type-based IFA tend to contain a large number of clusters relative to the number of vertices in the label-constraint graph, and relatively little abstraction that can be achieved through clusters. Consequently, they form a good test-case for stress-testing the cluster-based formulation for compacting label-constraint graphs.

6.4.2 Quantitative aspects of label-constraint graphs in type-based IFA

Table 6.1 details information about the structure of the label-constraint graphs for the standard library of the FlowCaml programming language. We present statistics for six files that contain library functions to create and manipulate arrays, hash tables, lists, queues, sets and stacks. These files are named array.fml, hashtbl.fml, list.fml, queue.fml, set.fml and stack.fml respectively. Each file constains multiple label-contraint graphs because there are multiple label-polymorphic expressions per file. We present two sets of characteristics per file to capture aggregate and individual statistics for label-constraint graphs. One set shows the total number of output variables (ov_{tot}), polar clusters (pc_{tot}), clusters (c_{tot}) and vertices (n_{tot}), by adding up these values for all label-constraint graphs in the file. We also present a second set of statistics to get an insight into the structure of individual label-constraint graphs. The second set of statistics shows the following: the average number of output variables (ov_{avg}), polar clusters (pc_{avg}), clusters (c_{avg}) and vertices (n_{avg}) for the label-constraint graphs for individual label-polymorphic expression in a file.

Filename	ov_{tot}	pc_{tot}	c_{tot}	n_{tot}	ov_{avg}	pc_{avg}	c_{avg}	n_{avg}
array.fml	1046	977	2653	2978	7.6	7.1	19.4	21.7
hashtbl.fml	2490	2259	5071	5745	8.1	7.4	16.6	18.8
list.fml	1876	1782	4022	4411	7.6	7.2	16.3	17.9
queue.fml	370	349	724	788	5.4	5.1	10.6	11.6
set.fml	1639	1562	4133	4608	5.1	4.9	12.9	14.4
stack.fml	337	315	551	614	5.4	5.1	8.9	9.9

Table 6.1: Cumulative and average statistics for label-constraint graphs

Functions in the standard library of a programming language that manipulate data structures are typically small and succinct. This is evidenced in the average statistics for label-constraint graphs in table 6.1. The average size of constraints graphs is small but it can be seen by comparing n_tot and n_avg values that there are hundreds of such small label-constraints graphs per file. Another interesting observation from table 6.1 which reinforces the discussions in section 6.4.1 is the incidence of a large number cross edges in the label-constraint graph; it can be seen that over 90% of the vertices are clusterheads. The level of abstraction that can be reached using the proposed algorithm is highly limited because the number of polar clusterheads is about the same as the number of output variables. The lack of a means to abstract away from vertices to sets of vertices makes type-based IFA an effective means to stress-test the proposed algorithm. Despite having to deal with intractable graphs in type-based IFA, we will experimentally show in the next section that the performance of the proposed algorithm is comparable to a baseline algorithm. The results make a strong case for cluster-based compaction, because if there is scope for decomposition one stands to gain from using a cluster-based approach, but even if there is none one is seldom very much worse off by adopting a cluster-based approach to compaction.

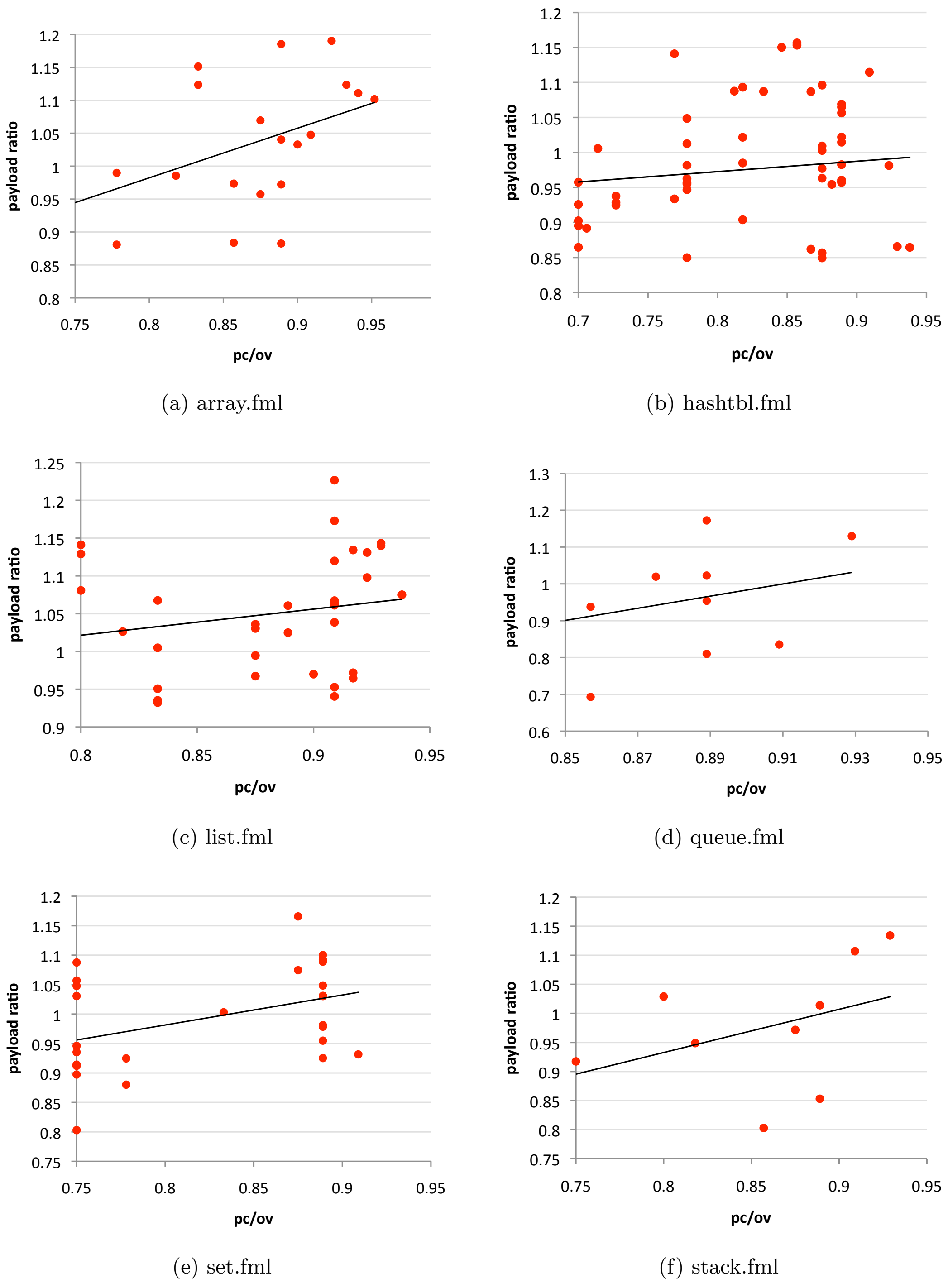
6.4.3 Performance in face of intractable graphs

Figure 6.5 shows scatterplots with best fit regression lines for the testcases described in section 6.4.2 and compares a cluster-based approach to a baseline algorithm described in section 6.3.2. For each file, each point in the scatterplot corresponds to a label-constraint graph in that file. The range on x-axis is the ratio of polar clusters to output variables in the label-constraints graph. The y-axis is the ratio of the time taken to collect output variables using the clusters based approach described in section 6.2.2 to the baseline algorithm. To ensure a realistic comparison of the algorithms, we only consider the ratios of collection times i.e. we subtract the time taken to traverse the label constraint graph from the total time taken by the algorithm. The traversal of the graph is the same operation in both the algorithms and getting rid of it gives us a better picture of the relative advantages of each algorithm. We have also chosen to ignore label-constraint graphs that have a large proportion of unconnected stand-alone vertices. Such vertices push up the pc/ov ratio to 1 and skew the analysis.

There are two important trends to observe in figure 6.5. Firstly, as the ratio of polar clusters to output variables increases, the baseline algorithm starts to perform better than the cluster-based approach; this is evidenced from the best fit lines in figure 6.5. This is because no latent abstraction can potentially be exploited using a cluster-based approach if every output variable is located in a separate cluster. In such a case, the cluster-based approach will introduce computational overheads. However, as evidenced from figure 6.5, this overhead is typically modest (mostly between 5% and 15%) for even the most intractable of graphs which have high pc/ov values. Secondly, it worth noting that the performance of the cluster-based approach is still comparable to the baseline algorithm for most of cases. Even in face of such intractable graphs (with $pc/ov > 0.7$), the cluster-based approach tends to perform better for a significant number of label-constraint graphs. For lower values of the pc/ov ratio, the cluster-based approach performs better than the baseline algorithm. For higher pc/ov values, the cluster-based approach is inhibited by the additional processing costs. It is anticipated the for pc/ov values of below 0.7, we can get a higher level of abstraction using clusters leading to significant gains in processing times. However, this is a case for future work and we intend to investigate this further.

6.5 Summary

In this chapter we discussed an algorithm for compacting label-constraint graphs of label-polymorphic expressions using graph decomposition. We first discussed a baseline algorithm for compaction and showed how graph decomposition can benefit the baseline algorithm. We presented our algorithm for compaction using graph decomposition and also analysed its computational costs. We showed how decomposition of the label-constraints graph into clusters of vertices lends a level of abstraction to the compaction operation, and showed theoretically how this abstraction leads to a lower computational cost. Since the level of abstraction that can be achieved is dependent on the structure of the label-constraint graph, we stress-tested the proposed approach against intractable graphs that do not lend themselves well to the decomposition proposed in this chapter. We showed that deployment of the cluster-based algorithm does not majorly impede performance even in the face of intractable graphs. This builds a strong case for the adoption of our algorithm because one stands to gain if there is any latent potential for abstraction using clusters; the upsides to our algorithm are promising with little downsides even in the face of intractable graphs. Adoption of the proposed algorithm enables a structure sensitive approach to compaction, where the runtime of the algorithm is a function of the latent scope for abstraction using clusters.



(a) array.fml

(b) hashtbl.fml

(c) list.fml

(d) queue.fml

(e) set.fml

(f) stack.fml

Figure 6.5: Ratio of payload collection times as a function of polar clusterhead to output variables ratio

Chapter 7

Conclusion and Future Work

In this thesis, we proposed techniques to efficiently simplify and solve constraints arising from type-based flow analysis. We used information flow analysis (IFA) as an application of our work. We derived a complexity bound on atomic constraint solving for type-based IFA of programs. Our results contrast with previous complexity assessments, which took into account only computations done for solving constraints on label variables while trivialising lattice pre-processing costs. Our assessment, on the other hand, took into account the costs for both constraint solving and the lattice pre-processing necessary for the solver. We noted that both the label-constraint graph and the security lattices in type-based IFA are typically represented as directed graphs by the constraint solver, and discussed how the efficiency of type-based IFA can gain from graph decomposition. This set the scene for proposing novel techniques for atomic constraint simplification and solving using graph decomposition which is the key contribution of this thesis.

We showed how lattices can be partitioned into non-overlapping trees called clusters. Then we showed how clusters can be exploited to pre-process the lattice to answer $\leq$, $\sqcup$ and $\sqcap$ queries in constant time. Partitioning the DAG representing the lattice into clusters introduced a level of abstraction in the pre-processing algorithm; it enabled us to reason at the level of sets of vertices rather than individual vertices, which made the pre-processing algorithm efficient. It also enabled the pre-processing algorithm to be described in such a manner that its computational cost is dependant on the latent scope for decomposition in the DAG. Thus, the proposed algorithm became highly adaptive in nature. It ran in the same time as the best reported algorithms for trees if the structure of the lattice is similar to a tree, and in the same time as the best reported algorithms for DAGs if the structure is similar to a dense DAG.

We demonstrated the suitability of the proposed lattice pre-processing algorithms by testing them out with real-world and random security lattices. We showed experimentally that using clusters as a building block makes the pre-processing algorithm compact and sensitive to the structure of the lattice under consideration. We discussed a wide spectrum of real-world security lattices that govern information flow in programs whose structure ranged from trees in the case of class hierarchies, all the way to dense DAGs in the case of powerset lattices. In view of this, we showed how an adaptive algorithm which is sensitive to the structure of the lattice under consideration is a superior approach to pre-processing these lattices.

Having made novel contributions to the solver using a decomposed approach to lattice pre-processing, we applied the concept of clusters to the simplification constraints on labels for label-polymorphic expressions. Such expressions lack any form of annotation, which renders the constraints on label variables unsolvable. For such expressions we showed how a cluster-based approach can help in representing the label constraints in a compact manner. We tested our algorithm by applying it to the label-constraints encountered in the standard library of FlowCaml - a full-fledged programming language that supports information flow analysis. Due to the additional edges in the label-constraints graph introduced through implicit flows in the code, these graphs had limited scope for decomposition. We showed that even in the face of such intractable graphs, the cluster-based algorithm for constraint graph compaction has comparable performance with a standard baseline algorithm for this case.

Our results for pre-processing lattices and compacting constraint graphs for label-polymorphic expressions underlines the key advantage of a cluster-based approach. If there is latent scope for decomposition, the cluster-based approach is superior to existing algorithms. Even if there is little scope for decomposition, one is never significantly worse off by adopting a cluster-based approach. This makes the cluster-based algorithms a desirable enabler for efficient bound constraint solvers like those studied in this thesis. The techniques proposed in this thesis have been shown to be sensitive to the structure of the label-constraint graphs, as well as to the policy lattice that governs the flow of information through programs. Thereby, we have experimentally demonstrated a notion of adaptability in computational costs based on the latent scope of decomposition in the constraint graph and the lattice. Without loss of generality, the techniques proposed in this research can be easily extended to other forms of type-based flow analysis, and can be used to design efficient constraint solvers for other problems involving atomic bound constraints.

7.1 Future Work

In this thesis we proposed an adaptive approach to solving atomic bound constraints. While the proposed techniques were shown to be applicable to a wide variety of issues in bound constraint solving, there are still avenues for improving the proposed framework and applying its core concepts to numerous other areas of application. In this section, we highlight avenues for future work.

7.1.1 Enrichment of the existing framework

Choice of the spanning tree: In this thesis, we didn't study an optimal method for traversing a DAG in order to obtain its spanning tree. In reality, however, the spanning tree of the DAG is not unique. The order of traversing vertices in the graph has a bearing on edge classification. For example, consider a forward edge in the DAG which directly connects two vertices `v1` and `v2`. The presence of the forward edge means that `v1` and `v2` are also transitively connected in the DAG through edges of spanning tree `T1` covering the DAG. Since the spanning tree is non-unique, it is also possible to construct a different spanning tree `T2` which traverses the forward edge in `T1` first followed by the transitive links between `v1` and `v2`. In the case of `T2`, the order of traversal will now create a cross edge coming into vertex `v2` from its set of parents sans `v1`. This discussion shows that a poor choice of the spanning tree may inadvertently introduce additional cross-edges and inhibit the performance of the adaptive techniques proposed in this thesis. Therefore, a further area of work for improving the proposed framework is to identify schemes for constructing and optimal or near-optimal spanning tree.

Dynamic Graphs: The cluster based decomposition of DAGs proposed in this thesis assumed a static DAG. While this is a good starting point to explore efficient means of pre-processing DAGs encountered in program analysis, it is by no means exhaustive. In a significant number of program optimisations, the DAG is dynamic; it keeps evolving as the program is compiled. Take for example, the case of program specialisation. When a generic function is specialised to a specific instances, new types are introduced the existing type lattice. This makes function dispatch a tricky problem. One needs a suitable means of introducing new types into the existing type lattice and yet be able to answer lattice queries efficiently to perform function dispatch. However, applying the techniques proposed in this thesis to lattices that evolve throughout the compilation process is a non-trivial problem. This is because once a new element is introduced in the existing lattice, the spanning tree needs to be reconstructed. A new spanning tree gives rise to a new set of cross edges and the entire process of computing closure over clusters has to be repeated. This can be a very expensive operation. Therefore, further work needs to be done to extend cluster-based closure techniques to lattices and DAGs that are dynamic.

7.1.2 Additional areas of application

In this dissertation, we discussed a novel way of computing transitive closure of graphs where the closure operation is over sets of vertices rather than individual vertices. We applied the proposed techniques to solve atomic constraints as well as answering queries in lattices. Both atomic constraints and lattice operations have widespread usage in programming languages and program analysis. A natural next step for this research would be to extend the techniques proposed in this dissertation to related areas where an efficient closure operation plays a central role. We highlight some of these areas below.

Type Inference: Much like IFA, where annotating each term with a label is cumbersome, it is undesirable to annotate types for every term in programs. Many modern programming languages, therefore, support some form of type inference where types for terms are represented as variables. Similar to IFA, the type inference engine tries to deduce the most general substitution for these variables by inspecting the data flow. In section 2.2.3, we presented HM(X) which is a generalisation of such inference engines that support Hindley-Milner polymorphism. Here, X was a parameter to the inference engine which defined the semantics of the relationship between type variables. For example, X could be specified to mean a unification-based system which equates two variables if one flows to another, or X could mean an inclusion-based system where a flow from a variable P to Q implies that P has at least as many attributes as Q, and hence we have have an inclusion constraint $P \leq Q$. It is important to note, however, that the $\leq$ relation has different meanings in different typing disciplines. For example, in structural subtyping which was discussed in this dissertation, the relation means P and Q have the same structure - hence the decomposition of this relation into atomic constraints is straightforward. In other subtyping disciplines, such as nominal subtyping or non-structural subtyping, we have a programmer supplied lattice over classes or type constructors, respectively. Therefore, it needs to be investigated how the techniques proposed in this dissertation could be directly extended to such systems. A case in point would be to use our approach for inference of the most general type in Java generics. There, the class hierarchy plays an important role in deciding the most general type for type parameters. We have already demonstrated the suitability of our approach to processing class hierarchies in chapter 5, in order to answer ordering queries in constant time. Therefore, we envisage that the adaptive inference techniques discussed in this dissertation can be extended to inference of generic types too. Reduction of different subtyping disciplines to an atomic subtyping problem entails different computational costs, and therefore such extensions of the core atomic constraint simplification and resolution algorithm discussed in this dissertation to richer subtyping systems need to be carefully evaluated.

Ownership Inference: As discussed in section 2.4, inferring ownership of objects requires approximating the object graph at compile time and identifying dominance boundaries of vertices in the graph. The inferred ownership properties are then fed back to the type system, and it is verified whether the inferred properties obey the properties of the Ownership Type system. To identify the dominance boundaries of vertices in the object graph, one needs to perform closure of the object graph to identify which vertices are dominated by any given vertex. In this regard, any form of abstraction and adaptability that can be introduced by techniques proposed in this thesis would aid the analysis in face of large object graphs. However, extension of the current work to analysing object graphs is non-trivial. Object graphs contain both forward edges and strongly connected components in addition to the restricted cases of tree and cross edges considered in this thesis. While strongly connected components can be considered as a single vertex for the sake of the analysis, the presence of forward edges cannot be ignored. This is because forward edges directly influence the dominator relationships in graphs. Therefore, it remains to be investigated how the techniques proposed in this thesis can be extended to forward edges and strongly connected components and subsequently, to ownership inference as well.

7.1.3 Limitations

The techniques described in this dissertation can be used to speed up all classes of problems where flow is governed by a lattice. Information Flow Analysis is just an instance of such a problem. Another such instance is that of type inference in the presence of polymorphic subtyping. As far as the qualitative results are concerned, our techniques can speed-up the identification of all forms of confidentiality and integrity breaches that occur through data and control flow in programs. However, the proposed approach does not take into account leaks through side-channels such as those exploited in timing attacks on computer programs.

We have assumed a determinable data and control flow graph in our approach. However, in reality many applications have non-deterministic control flow. For example, an application running in a multi-tasking environment may be sporadically suspended or woken up from sleep. Other instances of such control flow are rampant in mainstream mobile operating systems such as Android. An Android application does not have a single point of entry. Instead, an Android app is composed of components which can be invoked in arbitrary orders depending on user interactions and system events. In such a case, it is difficult to reconstruct an exact control flow graph and the usual approach is to model the control flow in light of the app lifecycle as described in the Android framework. However, in this work, we have not considered modelling the control flow for applications running in multi-tasking systems or applications that have a prescribed life-cycle transition diagram as is the case with Android apps.

Finally, we have assumed a rather simplistic model for permissible pathways in which information can be declassified. We have assumed a lattice-based security model. In reality, however, security models can be arbitrarily complex. For example, it is not unusual for a principal hierarchy to evolve over time. In such a case, the policy becomes dynamic which requires our pre-processing algorithm to be reapplied in order to perform lattice lookups in constant time. It is also possible for the declassification pathways to be predicated. For example, declassification from one element in an order to another could depend on some condition on the state of the elements. Our techniques are not sophisticated enough to cover these cases. We would need to augment our algorithm with a notion of logical implication to work out information leakage in such a scenario.

Bibliography

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 31–41, 1993.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 311–330, 2002.

[3] Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. *Journal of Computer Security*, 17(5):549–597, 2009.

[4] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, 2002.

[5] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 91–102, New York, NY, USA, 2006. ACM.

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[7] Chris Andreae, Yvonne Coady, Celina Gibbs, James Noble, Jan Vitek, and Tian Zhao. Scoped types and aspects for real-time java memory management. *Real-Time Systems*, 37(1):1–44, 2007.

[8] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 53–60, 2007.

[9] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: Permissive yet secure error handling. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 45–57, 2009.

[10] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, 2009.

[11] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 165–178, 2012.

[12] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control [extended abstract]. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 166–177, 2002.

[13] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, CSFW '02, pages 253–267, 2002.

[14] Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005.

[15] Anindya Banerjee and David A. Naumann. State based ownership, reentrance, and encapsulation. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 387–411, 2005.

[16] Anindya Banerjee and David A. Naumann. Aliasing in object-oriented programming. chapter State Based Encapsulation for Modular Reasoning About Behavior-preserving Refactorings, pages 319–365. Springer-Verlag, 2013.

[17] Amnon B. Barak and Paul Erdös. On the maximal number of strongly independent vertices in a random acyclic directed graph. *SIAM Journal on Algebraic and Discrete Methods*, 5(4):508–514, 1984.

[18] Matthias Baumgart, Stefan Eckhardt, Jan Griebsch, Sven Kosub, and Johannes Nowak. All-pairs ancestor problems in weighted DAGs. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *Lecture Notes in Computer Science*, pages 282–293. 2007.

[19] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[20] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[21] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal of Computing*, 22(2):221–242, 1993.

[22] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[23] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 25–37, 1998.

[24] Jr. Bocchino, RobertL. and VikramS. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, volume 6813 of *Lecture Notes in Computer Science*, pages 306–332. Springer Berlin Heidelberg, 2011.

[25] Robert L. Bocchino. Aliasing in object-oriented programming. chapter Alias Control for Deterministic Parallelism, pages 156–195. Springer-Verlag, 2013.

[26] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, 2009.

[27] Robert L. Bocchino, Jr., Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 535–548, New York, NY, USA, 2011. ACM.

[28] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 180–196, 2006.

[29] Niklas Broberg and David Sands. Flow-sensitive semantics for dynamic information flow policies. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 101–112, 2009.

[30] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. *SIGPLAN Not.*, 45(1):431–444, 2010.

[31] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer Berlin Heidelberg, 2013.

[32] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: Deployment-time confinement checking. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 374–387, 2003.

[33] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, 1987.

[34] Don Coppersmith. Rectangular matrix multiplication revisited. *Journal of Complexity*, 13(1):42 – 49, 1997.

[35] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 3rd edition, 2009.

[36] Artur Czumaj, Miroslaw Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, 2007.

[37] Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin Pierce. Sensitivity analysis using type-based constraints. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages*, FPCDSL '13, pages 43–50, 2013.

[38] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[39] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 230–241, 1994.

[40] Werner Dietl, Michael D. Ernst, and Peter Müller. Tunable static inference for generic universe types. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 333–357, 2011.

[41] Werner Dietl and Peter Müller. Aliasing in object-oriented programming. chapter Object Ownership in Program Verification, pages 289–318. Springer-Verlag, 2013.

[42] Stefan Eckhardt, Andreas Mühling, and Johannes Nowak. Fast lowest common ancestor computations in DAGs. In *Algorithms ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 705–716. 2007.

[43] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In *17th Symposium on Combinatorial Pattern Matching (CPM), volume 4009 of LNCS*, pages 36–48. Springer, 2006.

[44] Alexandre Frey. Satisfying subtype inequalities in polynomial space. *Theoretical Computer Science*, 277(1-2):105–117, 2002.

[45] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[46] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 135–143, 1984.

[47] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, 2004.

[48] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[49] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[50] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Marktoberdorf Summer School*. IOS Press, 2011.

[51] Nevin Heintze. Control-flow analysis and type systems. In *Static Analysis Symposium*, SAS'95, pages 189–206, 1995.

[52] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 365–377, 1998.

[53] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Logic in Computer Science, 1997. LICS '97. Proceedings., 12th Annual IEEE Symposium on*, pages 352–361, 1997.

[54] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[55] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment. In KimG. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 616–627. Springer Berlin Heidelberg, 1998.

[56] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012.

[57] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ml. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 781–786, 2012.

[58] Wei Huang, Werner Dietl, Ana Milanova, and Michael D. Ernst. Inference and checking of object ownership. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 181–206, 2012.

[59] Sebastian Hunt and David Sands. On flow-sensitive security types. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, 2006.

[60] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 329–341, 1998.

[61] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 223–236, 2010.

[62] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15:290–311, 1993.

[63] Miroslaw Kowaluk and Andrzej Lingas. Unique lowest common ancestors in dags are almost as easy as matrix multiplication. In *Algorithms – ESA 2007*, volume 4698 of *Lecture Notes in Computer Science*, pages 265–274. Springer Berlin Heidelberg, 2007.

[64] Miroslaw Kowaluk, Andrzej Lingas, and Johannes Nowak. A path cover technique for LCAs in DAGs. In *SWAT '08: Proceedings of the 11th Scandinavian workshop on Algorithm Theory*, pages 222–233, 2008.

[65] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 671–690, 2010.

[66] V. Kuncak and M. Rinard. Structural subtyping of non-recursive types is decidable. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 96–107, 2003.

[67] Peng Li and S. Zdancewic. Encoding information flow in haskell. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 12–16, 2006.

[68] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. *SIGPLAN Not.*, 40(1):158–170, 2005.

[69] Yi Lu, John Potter, and Jingling Xue. Ownership downgrading for ownership types. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 144–160, 2009.

[70] Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for java. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 423–440, 2007.

[71] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. A lattice-based approach to mashup security. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 15–23, 2010.

[72] Ana Milanova and Yin Liu. Practical static ownership inference. RPI/DCS-09-04, Rensselaer Polytechnic Institute, 2009.

[73] Ana Milanova and Jan Vitek. Static dominance inference. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns*, TOOLS'11, pages 211–227, 2011.

[74] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1:245–285, 1991.

[75] Christian Mossin. *Flow analysis of typed higher-order programs.* PhD thesis, DIKU, University of Copenhagen, 1997.

[76] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[77] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, 1984.

[78] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 228–241, 1999.

[79] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, 1997.

[80] Joachim Niehren, Tim Priesnitz, and Zhendong Su. Complexity of subtype satisfiability over posets. In *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 357–373. Springer Berlin Heidelberg, 2005.

[81] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[82] Jens Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01*, pages 20–27, 2001.

[83] Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '07, pages 1–10, 2007.

[84] Benjamin Pierce. *Types and Programming Languages.* The MIT Press, 1 edition, 2002.

[85] François Pottier. Simplifying subtyping constraints: a theory. *Information and Computation*, 170:153–183, 2001.

[86] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 46–57, 2000.

[87] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.

[88] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 228–238, 1998.

[89] François Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, 2001.

[90] Francois Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[91] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to cfl-reachability. In *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 54–56, 2001.

[92] Jakob Rehof. Minimal typings in atomic subtyping. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 278–291, 1997.

[93] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2006.

[94] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 255–269, 2005.

[95] Andrei Sabelfeld and AndrewC. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin Heidelberg, 2004.

[96] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer Berlin Heidelberg, 2010.

[97] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[98] Sven-Bodo Scholz. Single assignment c: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[99] Vincent Simonet. An extension of HM(X) with bounded existential and universal data-types. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, pages 39–50, 2003.

[100] Vincent Simonet. The Flow Caml System: Documentation and user's manual. Technical Report RT-0282, INRIA, 2003.

[101] Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *APLAS*, pages 283–302, 2003.

[102] Christian Skalka and Scott F. Smith. Static use-based object confinement. *International Journal of Information Security*, 4(1-2):87–104, 2005.

[103] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 355–364, 1998.

[104] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 203–216, 2002.

[105] Daniel Tang, Ales Plsek, and Jan Vitek. Static checking of safety critical java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 148–154, 2010.

[106] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[107] J. Tiuryn. Subtype inequalities. In *Logic in Computer Science, Proceedings of the Seventh Annual IEEE Symposium on*, LICS '92, pages 308–315, 1992.

[108] Jerzy Tiuryn. Subtyping over a lattice. Technical report, Warsaw University, 1997.

[109] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '93, pages 686–701, 1993.

[110] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(34):1 – 34, 1990.

[111] Valery Trifonov and Scott Smith. Subtyping constrained types. In *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 349–365. Springer Berlin Heidelberg, 1996.

[112] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[113] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 75, 2006.

[114] J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1998.

[115] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th symposium on Theory of Computing*, STOC '12, pages 887–898, 2012.

[116] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[117] Damiano Zanardini. Class-level Non-Interference. *New Generation Computing*, 30(2-3):241–270, 2012.

[118] Tian Zhao, Jason Baker, James J. Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008.

[119] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, RTSS '04, pages 241–251, 2004.

[120] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.