Stony Brook University

From Rules to Efficient Algorithms for Cyber Trust Applications

A Dissertation Presented

by

Katia Hristova

to

The Graduate School

in Partial fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2007

Stony Brook University

The Graduate School

Katia Hristova

We, the dissertation committee for the above candidate for

the degree of Doctor of Philosophy

hereby recommend the acceptance of this dissertation

Professor Scott Stoller, (Chairman)

Computer Science Department

Professor Yanhong A. Liu, (Advisor)

Computer Science Department

Professor C. R. Ramakrishnan, (Committee Member)

Computer Science Department

Professor David Warren, (Committee Member)

Department of Computer Science

Dr. Vugranam Sreedhar, (External Committee Member)

IBM TJ Watson Research Center, Yorktown Heights, NY

This dissertation is accepted by the Graduate School.

Lawrence Martin

Dean of the Graduate School

Abstract of the Dissertation

From Rules to Efficient Algorithms for Cyber Trust Applications

by
Katia Hristova

Doctor of Philosophy
in
Computer Science
Stony Brook University
2007

Cyber trust applications require correct and efficient algorithms for solving complex analysis problems. We address this challenge by generating efficient algorithms and implementations from high-level specifications of these problems expressed using rules. We use extended Datalog rules to intuitively specify analysis problems in the areas of model checking, information flow analysis, and trust management, and then generate efficient algorithms and implementations systematically from the rules. Our work resulted in new and more efficient algorithms for some problems and new and improved time and space complexity analysis for others.

In the model checking area, we describe an efficient algorithm with improved complexity analysis for linear temporal logic model checking of pushdown systems. This model checking framework can express and check many practical properties of programs, including many dataflow properties and general correctness and security properties. For secure information flow analysis, we describe the development of the first linear-time algorithm for inferring information flow types of programs for a formal type system. We also extend the algorithm with informative error reporting to facilitate error detection and corrections. In the area of trust management, we describe efficient algorithms for analysis of trust management policies specified in SPKI/SDSI, a well-known trust management framework designed to facilitate the development of secure and scalable distributed computing systems.

Our approach of expressing policy analysis problems as rules is much simpler than previous techniques, in addition to deriving better, more precise time complexities. We show the efficiency of these algorithms by performing precise time and space complexity analysis and confirming them through experiments.

Lastly, we describe a method to generate efficient algorithms for answering rule-based queries. The method is based on the well-known magic set transformation. We apply the method to query problems for graph reachability, as well as in model checking, information flow analysis, and security policy frameworks.

To my family

Contents

List of Figures

Acknowledgments

This thesis is the product of several years of hard work, persistence, procrastination, hesitation, and a serious amount of external assistance. I believe the last item on the list is arguably the most significant.

First of all, my thanks go to my adviser Prof. Annie Liu, who stayed the course with me through repeated qualifying exams, paper drafts, project changes, title changes, a long job search, and a maternity leave. I cannot overstate the importance of her involvement in my graduate career. I thank her for her enthusiasm and inspiration, kind support, trenchant critiques, and most of all - her remarkable patience.

I am also thankful to my committee members Prof. Scott Stoller, Prof. C.R. Ramakrishnan, Prof. David Warren, and Dr. Vugranam Sreedhar for being on my committee and for providing me with their feedback. Prof. Stoller's help with his valuable insights and sound advice has been immense. I truly appreciate Prof. Warren's amazingly clear explanation of the Magic Set Transformation, and Prof. Ramakrishnan's help with my work on model checking.

I would also like to thank my labmates for many useful discussions and advice, and for providing me with a friendly and collaborative work environment. Special thanks go to Tom Rothamel for his help with our work on information flow and with numerous implementation issues. I would like to thank Tuncay Tekle for his insight, collaboration and proofreading of our work on trust management.

My warmest thanks goes to my family and my family in law. My grandmothers Katia and Stoina, my grandfather Dimitar, my mother Stoyanka, my father Petar, and my sister Desi, for their love and understanding, unconditional support and irreplaceable help. Much of my life I have been to them just like my thesis has been to me - at times troublesome, at times demanding, at times boring, at times overwhelming... I can now sympathize with them in my own way. :) I cannot forget to mention my cat Iovchev, who often stared at me through the webcam, undoubtedly wondering what is so important over there.

Countless thanks to the sweetest and most loving family in law I could possibly have! My mom-in-law Jamile, dad-in-law Mohamad, my brothers-in-law Ali, Hussein and Hadi, and my sister-in-law Dina. I am grateful for the love they have given me, their constant

support and their absolute confidence in me, and for being there for me even over the distance that separates us.

I would like to say an extra thank you to my mother and my brother-in-law Hussein for leaving behind their loved ones, their homes, and their jobs across the ocean, to be here with me and take care of my newborn son while I completed writing this thesis.

This work would not have been possible without my wonderful husband Firas, who has been next to me throughout the PhD process. His love and encouragement have been a guiding light for me through these years. His unflinching courage, persistence and strength will always be my inspiration! Special thanks to our sweet baby boy Stanislav Nader for his patience while I was finishing this thesis, and for his mesmerizing smiles that brightened even the hardest days!

I am also grateful to my best friend and labmate Rahul whose incurable enthusiasm for the work and experience with the process enabled him to both encourage me and commiserate with me appropriately. His timely comments and sound advice have been invaluable. Our coffee meetings, walks and chats were priceless rejuvenation in the busy days for me. Thank you, Rahul, for putting up with my endless questions, frantic emails, and long conversations!

Finally, thanks to all the wonderful friends I met in graduate school who have made Stony Brook a very special place for me and my times here unforgettable.

Chapter 1

Introduction

This thesis focuses on improving software productivity by formulating and using high-level problem specifications that are amenable for automated code generation and analysis. We design efficient algorithms and implementations for problems in the areas of cyber trust, ranging from verification to trust management and computer security and privacy. We express the problems in a high level language, and use tools that automatically generate efficient implementations from high-level specifications. Our work demonstrates that adopting such an approach significantly cuts down the cost and time for the software development process of the software lifecycle.

Cyber trust applications require correct and efficient algorithms for solving complex analysis problems. We address this challenge by generating efficient algorithms and implementations from high-level specifications of these problems expressed using rules. We use extended Datalog rules to intuitively specify analysis problems in the areas of model checking, information flow analysis, and trust management, and then generate efficient algorithms and implementations systematically from the rules. Our work resulted in new and more efficient algorithms for some problems and new and improved time and space complexity analysis for others.

LTL Model Checking of Pushdown Systems. Model checking is a widely used technique for verifying that a property holds for a system. Systems to be verified can be modeled accurately by pushdown systems (PDS). Properties can be modeled by linear temporal logic (LTL) formulas. LTL is a language commonly used to describe properties of systems and is sufficiently powerful to express many practical properties. Examples include many dataflow analysis problems and various safety and security problems for programs.

In our work the model checking problem is formulated in terms of evaluation of a Datalog program. The Büchi PDS, corresponding to the product of the PDS and the automaton representing the inverse of the property, is expressed in Datalog facts, and a reach graph — an abstract representation of the Büchi PDS, is formulated in rules. Efficient algorithms and data structures are generated automatically directly from the rules. We thus developed a model checker with improved time complexity guarantees and improved algorithm understanding.

The algorithm derived in this work is essentially the same as the one presented in [35]. What distinguishes our work is that we used a novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [13] and a systematic method for deriving efficient algorithms and data structures from the rules [57], and arrives at an improved complexity analysis. We show the effectiveness of our approach by using a precise time complexity analysis, along with experiments.

This work appeared in the 7th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI), 2006 [44].

Type Inference for Secure Information Flow. Protection of the confidentiality and privacy of data is becoming increasingly important. In addition to controlling direct access to information, it is also essential to control information flow, especially in untrusted code. Static analysis of information flow in programs allows for fine-grained control without runtime overhead. Denning [29, 30] proposed a lattice model that could be used to verify secure information flow in programs. In this model, security classes are ordered in a lattice, and program variables and data are each assigned a security class. The basic security requirement is absence of information flow from higher to lower security classes. Security classes can indicate both the level of secrecy and the level of integrity of data. Based on Denning's lattice model of information flow analysis, several type-based approaches have been developed in which the security properties are formulated as type systems — formal systems of typing rules used to reason about information flow properties of programs.

We described the design, analysis, and implementation of the first linear time algorithm for information flow analysis expressed using a type system. This work is based on the type system presented by Volpano et al. in [88], which formulates Denning's lattice model and is shown to be sound. Information flow is guaranteed to be secure for a program if the program type checks correctly.

Given a program and an environment of security classes for information accessed by the program, our algorithm checks whether the program is well typed, i.e., there is no information of higher security classes flowing into places of lower security classes, by inferring the highest or lowest security class as appropriate for each program node. We express the analysis as a set of extended Datalog rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the extended Datalog rules. Our extended Datalog rules are Datalog rules with negation and external functions. The method described in [57] is used to generate specialized algorithms and data structures and complexity formulas for the extended Datalog rules.

Given a program and an environment of security types, the algorithm infers minimum or maximum security types, as appropriate, for each program node, such that the program type checks correctly. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types.

The generated implementation uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The implementation employs an incremental approach that considers one program node at a time. The running time is optimal for the set of rules we use to specify type inference, in the sense that each combination of instantiations of hypotheses is considered once in `O(1)` time. We thus obtained an efficient type inference algorithm.

The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus a small overhead for preprocessing the lattice. This complexity is confirmed through our prototype implementation and experimental evaluation on code generated from high-level specifications for real systems.

An early version of this work appeared in the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), 2006, and a journal version of it appeared as a technical report [45].

Trust Management Policy Analysis. Trust management is a unified approach to specifying and enforcing security policies in distributed systems and has become increasingly important as systems become increasingly interconnected. We described a systematic method for deriving efficient algorithms and precise time complexities from extended Datalog rules as it is applied to the analysis of trust management policies specified in SPKI/SDSI, a well-known trust management framework designed to facilitate the development of secure and scalable distributed computing systems.

SPKI/SDSI [34] is based on public keys and incorporates *Simple Public Key Infrastructure* (SPKI) and *Simple Distributed Security Infrastructure* (SDSI). It provides fine-grained access control using local name spaces and a security policy model. The SPKI/SDSI framework uses name certificates to define names in principals' local name spaces as keys or other names, and uses authorization certificates to grant authorizations and to delegate the ability to grant authorizations. A principal is authorized to access a resource by an authorization certificate or by a chain of certificates involving naming and delegation. Designing efficient algorithms for inferring authorizations and answering related queries is essential for enforcing SPKI/SDSI policies.

We expressed policy analysis problems using extended Datalog, extended with list constructors and external functions. We represent certificates as facts, and describe rules and queries for computing the name-reduction closure, inferring authorizations, and solving other policy analysis problems for SPKI/SDSI. These other analysis problems include ones about the current state of the policy, as well as ones about changes in the state that would be caused by possible changes in the policy, such as expiration or addition of a set of certificates.

We systematically generated specialized algorithms and data structures, together with precise time complexity formulas, from the extended Datalog rules for computing name-reduction closure and inferring all authorizations. The generated algorithms consider one certificate or intermediate analysis fact at a time, and use a combination of linked and indexed data structures to represent different certificates and intermediate values.

Other policy analysis problems are specified as additional rules and queries, and use a method to systematically push given inputs for the analysis from queries into hypotheses of rules, yielding specialized and simplified rules for the given queries. This is similar to pushing demands by queries in magic set transformations [15], but instead of yielding more complicated rules with magic predicates, we obtain simplified, specialized rules that are much easier for generating efficient implementations and precise complexities.

Our approach of expressing policy analysis problems as extended Datalog rules is much simpler than previous techniques for analysis of SPKI/SDSI policies. Our method also derived better, more precise time complexities than before in addition to generating complete algorithms and data structures.

This work appeared in the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming (PPDP), 2007 [46].

Answering Rule-Based Queries Efficiently. In the last chapter of this thesis we describe a method to generate algorithms that perform on-demand computation, i.e., algorithms to answer rule-based queries efficiently. The described method combines a prominent bottom-up optimization called Magic Set Transformation (MST) [15] with a systematic method for deriving efficient algorithms and data structures from the rules [57]. The method focuses on Datalog, which is an important logic-based programming language and can be used to model a significant class of practical problems. We apply the method to graph reachability, trust management policy analysis, information flow analysis, and model checking problems.

Datalog. Datalog is a database query language based on the logic programming paradigm [22, 3]. A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, ..., x_{1a_1}) \ , ..., \ p_h(x_{h1}, ..., x_{ha_h}) \rightarrow q(x_1, ..., x_a)$$

where h is a natural number, each p_i (respectively q) is a relation of a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $q(x_1, ..., x_a)$ is called a *fact*. For the rest of the thesis, "rule" refers only to the case where $h \geq 1$, in which case each $p_i(x_{i1}, ..., x_{ia_i})$ is called a *hypothesis* of the rule, and $q(x_1, ..., x_a)$ is called the *conclusion* of the rule. The meaning of a set of rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules.

The extended Datalog we use is Datalog with negation and external functions.

The rest of this thesis is organized as follows. Chapter 2 describes the implementation and analysis for LTL model checking of pushdown systems. Chapter 3 describes the design, analysis, and implementation of an efficient algorithm for secure information flow analysis based on the type system. Chapter 4 describes efficient algoritms for analysis of trust management policies specified in the SPKI/SDSI framework. Chapter 5 describes a method for answering rule-based queries efficiently. Chapter 6 concludes.

Chapter 2

Improved Algorithm Complexity Analysis for LTL Model Checking of PDS

Model checking is a widely used technique for verifying that a property holds for a system. Systems to be verified can be modeled accurately by pushdown systems (PDS). Properties can be modeled by linear temporal logic (LTL) formulas. LTL is a language commonly used to describe properties of systems [26, 27, 67] and is sufficiently powerful to express many practical properties. Examples include many dataflow analysis problems and various safety and security problems for programs.

In order to solve the LTL model checking problem for PDSs, a Büchi automaton B, corresponding to the property to be checked for, is constructed. The automaton B and PDS, P, corresponding to the system to be model checked, are combined into one product Büchi PDS BP. BP accepts the unwanted behaviors of the system, thus checking correctness amounts to checking for emptiness of the automaton BP (i.e., showing the system has no unwanted behavior). If BP accepts the empty language only, then true is returned. Otherwise, an example of input that BP accepts is returned.

This chapter focuses on LTL model checking of PDS, specifically on the global model checking problem [35]. The model checking problem is formulated in terms of evaluation of a Datalog program [13]. The Büchi PDS, corresponding to the product of the PDS and the automaton representing the inverse of the property, is expressed in Datalog facts, and a reach graph — an abstract representation of the Büchi PDS, is formulated in rules. The method described in [57] generates specialized algorithms and data structures and complexity formulas for the rules. The generated algorithms and data structures are such that given

a set of facts, they compute all facts that can be inferred. The generated implementation employs an incremental approach that considers one fact at a time and uses a combination of linked and indexed data structures for facts. The running time is optimal, in the sense that each combination of instantiations of hypotheses is considered once in `O(1)` time.

Our main contributions are:

- A novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [13] and a systematic method for deriving efficient algorithms and data structures from the rules[57].
- A precise and automatic time complexity analysis of the model checking problem. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

We thus develop a model checker with improved time complexity guarantees and improved algorithm understanding.

The rest of this chapter is organized as follows. Section 2 defines LTL model checking of PDS. Section 3 expresses the model checking problem by use of Datalog rules. Section 4 describes the generation of a specialized algorithm and data structures from the rules and analyzes time complexity of the generated implementation. Section 5 discusses related work and concludes.

2.1 Linear Temporal Logic Model Checking of Pushdown Systems

This section defines the problem of model checking PDS against properties expressed using LTL formulas, as described in [35].

2.1.1 Pushdown systems

A *pushdown system (PDS)* [33] is a triple (C_P, S_P, T_P), where C_P is a set of control locations, S_P is a set of stack symbols and T_P is a set of transitions. A transition is of the form $(c, s) \rightarrow (c', w)$ where c and c' are control locations, s is a stack symbol, and w is a sequence of stack symbols; it denotes that if the PDS is in control location c and symbol s is on top of the stack, the control location changes to c', s is popped from the stack, and the symbols in w are pushed on the stack, one at a time, from left to right. A *configuration* of a PDS is a pair (c, w) where c is a control location and w is a sequence of symbols from the top of the stack. If $(c, s) \rightarrow (c', w) \in T_P$ then for all $v \in {S_P}^*$, configuration

(c, sv) is said to be an *immediate predecessor* of (c', wv). A *run* of a PDS is a sequence of configurations $conf_0, conf_1, ..., conf_n$ such that $conf_i$ is an immediate predecessor of $conf_{i+1}$, for $i = 0, \ldots, n-1$.

We only consider PDSs where each transition $(c, s) \rightarrow (c', w)$ satisfies $|w| \leq 2$. Any given PDS can be transformed to such a PDS. Any transition $(c, s) \rightarrow (c', w)$, such that $|w| > 2$, can be rewritten into $(c, s) \rightarrow (c', w_{hd}\ s')$ and $(c', s') \rightarrow (c, w_{tl})$, where w_{hd} is the first symbol in w, w_{tl} is w without its first symbol, and s' is a fresh symbol. This step can be repeated until all transitions have $|w| \leq 2$. This replaces each transition $(c, s) \rightarrow (c', w)$, where $|w| > 2$, with $|w| - 1$ transitions and introduces $|w| - 1$ fresh stack symbols.

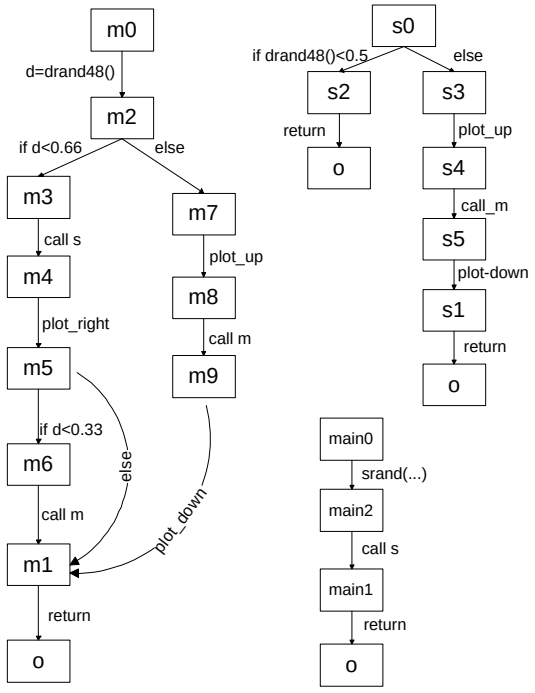
The procedure calls and returns in a program correspond to a PDS [36]. First, we construct a control flow graph (CFG) [4] of the program. Then, we set up one control location, say called c. Each CFG vertex is a stack symbol. Each CFG edge (s, s') corresponds to a transition (i) $(c, s) \rightarrow (c, \epsilon)$, where ϵ stands for the empty string, if (s, s') is labeled with a return statement; (ii) $(c, s) \rightarrow (c, s' f_0)$, if (s, s') is labeled with a call to procedure f, and f_0 is f's entry point; (iii) $(c, s) \rightarrow (c, s')$, otherwise. A run of the program corresponds to a PDS run.

```
void m()
  double d = drand48();
  if (d < 0.66):
    s(); plot_right();
    if (d < 0.33): m();
    else:
  else:
    plot_up(); m(); plot_down();

void s()
  if (drand48() < 0.5): return;
  else:
    plot_up(); m(); plot_down();

main()
  srand48(time(NULL)); s();
```

(a) Example program.



(b) Corresponding CFG.

Figure 2.1: Example program and corresponding CFG.

Figure 2.1 shows an example program and its CFG [35]. The program creates random bar graphs using the commands `plot_up`, `plot_right`, and `plot_down`. The corresponding PDS is:

$$
\begin{aligned}
C_P &= \{c\} \\
S_P &= \{m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, s0, s1, s2, s3, s4, s5, \\
&\quad main0, main1, main2\} \\
T_P &= \{(c, m3) \to (c, m4s0), (c, m6) \to (c, m1m0), (c, m8) \to (c, m9m0), \\
&\quad (c, m1) \to (c, \epsilon), (c, s2) \to (c, \epsilon), (c, s4) \to (c, s5m0), \\
&\quad (c, s1) \to (c, \epsilon), (c, main2) \to (c, main1s0), (c, main1) \to (c, \epsilon)\}
\end{aligned}
$$

2.1.2 Linear temporal logic formulas

Linear temporal logic (LTL) formulas [26, 27, 67] are evaluated over infinite sequences of symbols. The standard logic operators are available; if f and g are formulas, then so are $\neg f$, $f \wedge g$, $f \vee g$, $f \to g$. The following additional operators are available: $X\ f$: f is true in the next state; $F\ f$: f is true in some future state; $G\ f$: f is true globally, i.e., in all future states; $g\ U\ f$: g is true in all future states until f is true in some future state.

A LTL formula can be translated to a Büchi automaton, a finite state automaton over infinite words. The automaton accepts a word if on reading it a good state is entered infinitely many times. Formally, a *Büchi automaton* (BA) is a tuple $(C_B, L_B, T_B, C0_B, G_B)$ where C_B is a set of states, L_B is a set of transition labels, T_B is a set of transitions, $C0_B \subseteq C_B$ is a set of starting states, and $G_B \subseteq C_B$ is a set of good states. A transition is of the form (c, l, c'), where $c, c' \in C_B$ and $l \in L_B$. The label of a transition is a condition that must be met by the current symbol in the word being read, in order for the transition to be possible. A label _ denotes an unconditional transition. An *accepting run* of a Büchi automaton is an infinite sequence of transitions $(c_0, l_0, c_1), (c_1, l_1, c_2), \ldots, (c_{n-1}, l_{n-1}, c_n)$, where a state $c_i \in G_B$ appears infinitely many times.

To specify a program property using an LTL formula, the program's CFG edges are used as atomic propositions. LTL formulas are defined with respect to infinite runs of the program. The corresponding BA accepts an infinite sequence of CFG edges, if on reading it, the automaton enters a good state infinitely many times. For example, the property that plotting up is never immediately followed by plotting down is expressed by the LTL formula $F = G(plot\_up \to X(\neg plot\_down))$. The BA* corresponding to $\neg F$ is shown in Figure 2.2. In the diagram nodes correspond to states and edges correspond to transitions of the BA; double circles mark good states and a square marks the start state.

2.1.3 LTL model checking of PDS

Given a system expressed as a PDS P, and a LTL formula F, the formula F holds for P if it holds for every run of P. We check whether F holds for P as follows [35]. First, we construct B — the BA corresponding to $\neg F$. Second, we construct BP — a Büchi PDS that is the product of P and B, and make sure BP has no accepting run. A

*The Büchi automaton was generated with the tool LBT that translates LTL formulas to Büchi automata (http://www.tcs.hut.fi/Software/maria/tools/lbt/).

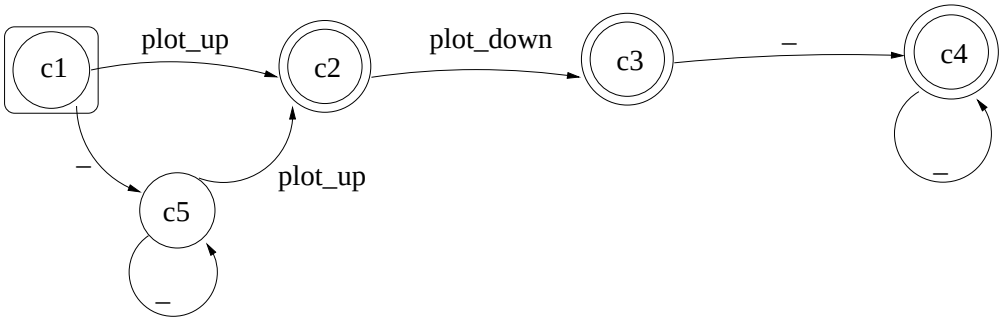


Figure 2.2: Büchi automaton corresponding to $\neg G(plot\_up \rightarrow X(\neg plot\_down))$.

Büchi PDS (BPDS) is a tuple (C, S, T, C_0, G), where C is a set of control locations, S is a set of stack symbols, T is a set of transitions, $C_0 \subseteq C$ is the set of starting control locations, $G \subseteq C$ is the set of good control locations. Transitions are of the form $((C * S) * (C * S^*))$. The concepts *configuration*, *predecessor*, and *run* of a BPDS are analogous to those of a PDS. An *accepting run* of the BPDS is an infinite sequence of configurations in which configurations with control locations in G appear infinitely many times. The product BPDS BP of $P = (C_P, S_P, T_P)$ and $B = (C_B, L_B, T_B, C0_B, G_B)$ is the five-tuple $((C_P * C_B), S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, where $(((c_P, c_B), s), ((c'_P, c'_B), w)) \in T_{BP}$ if $(c_P, s) \rightarrow (c'_P, w) \in T_P$, and there exists f such that $(c_B, f, c'_B) \in T_B$, and f is true at configuration $((c_P, c_B), s)$; $(c_P, c_B) \in C0_{BP}$ if $c_B \in C0_B$; $(c_P, c_B) \in G_{BP}$ if $c_B \in G_B$.

The *reach graph* is analogous to the *A_br* automaton as it is described in [37] and the *R graph* in [13].

Next we construct a *reach graph* — a finite graph that abstracts BP. The nodes of the graph are configurations of BP. An edge $((c, s), (c', s'))$ in the reach graph corresponds to a run that takes BP from configuration (c, s) to configuration (c', s'). If a good control location in BP is visited in the run corresponding to an edge, the edge is said to be *good*. A path in the reach graph is a sequence of edges. Cycles in the reach graph correspond to infinite runs of BP. Paths containing cycles with good edges in them correspond to accepting runs of BP and are said to be *good*. If the reach graph corresponding to BP has no good paths, BP has no accepting runs and F holds for P. Otherwise, the good paths in the reach graph are counterexamples showing that F does not hold for P.

2.2 Specifying the Reach Graph in Rules and Detecting Good Paths

This section expresses the reach graph using Datalog rules and and describes the generation of a specialized algorithm and data structures for detecting good paths in the reach graph.

Expressing the Büchi PDS. The BPDS is expressed by the relations `loc`, `trans0`, `trans1`, and `trans2`. The `loc` relation represents the control locations of the BPDS; its arguments are a control location and a boolean argument indicating whether the control location is good. One instance of the relation exists for each control location. The three relations `trans0`, `trans1`, and `trans2` express transitions. The facts `trans0(c1,s1,c2)`, `trans1(c1,s1,c2,s2)`, and `trans2(c1,s1,c2,s2,s3)`, where `ci`'s are control locations and `si`'s are stack symbols, denote transitions of the form $((c, s), (c, w))$ such that, $w \in S^*_{BP}$ and $|w| = 0$, $|w| = 1$, and $|w| = 2$, respectively. `or` is a relation with three boolean arguments; in the fact `or(x1,x2,r)`, the argument `r` is the value of the logical or of the arguments `x1` and `x2`.

Expressing the edges of the reach graph. The reach graph is expressed by relations `erase` and `edge`. The fact `erase(c1,s1,g,c2)` denotes a run of BP from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$. The third element in the tuple is a boolean value that indicates whether the corresponding run goes through a good control location. The `edge` relation represents the reach graph edges. `edge(c1,s1,g,c2,s2)` denotes an edge between nodes $(c1, s1)$ and $(c2, s2)$; `g` is a boolean argument indicating whether the edge is good. For a BPDS $(C_{BP}, S_{BP}, T_{BP}, C0_{BP}, G_{BP})$, `erase` and `edge` are the relation satisfying:

i. $(c1, s, g, c2) \in$`erase` if $(c1, s) \rightarrow (c2, \epsilon) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

ii. $(c1, s1, g1 \vee g2, c3) \in$`erase` if $(c1, s1) \rightarrow (c2, s2) \in T_{BP}$, and $(c2, s2, g2, c3) \in$ `erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise

iii. $(c1, s1, g1 \vee g2 \vee g3, c4) \in$`erase` if $(c1, s1) \rightarrow (c3, s2s3) \in T_{BP}$, $(c2, s2, g2, c3) \in$`erase`, and $(c3, s3, g3, c4) \in$`erase`, and $g1 = true$ if $c1 \in G_{BP}$ and $false$ otherwise

and

i. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \rightarrow (c2, s2) \in T_{BP}$, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

ii. $(c1, s1, g, c2, s2) \in$`edge` if $(c1, s1) \rightarrow (c2, s2s3) \in T_{BP}$, $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

iii. $(c1, s1, g1 \vee g2, c3, s3) \in$`edge` if $(c1, s1) \rightarrow (c2, s2s3) \in T_{BP}$, $(c2, s2, g2, c3) \in$ `erase`, and $g = true$ if $c1 \in G_{BP}$ and $false$ otherwise

In model checking of programs, the relation `erase` summarizes the effects of procedures. The three parts of the above definition correspond to the program execution exiting, proceeding within, or entering a procedure.

```
trans0(c1,s1,c2),loc(c1,g)→erase(c1,s1,g,c2)
trans1(c1,s1,c2,s2),erase(c2,s2,g2,c3),loc(c1,g1),
    or(g1,g2,g)→erase(c1,s1,g,c3)
trans2(c1,s1,c2,s2,s3),erase(c2,s2,g2,c3),
    erase(c3,s3,g3,c4),loc(c1,g1),or(g1,g2,g4),or(g4,g3,g)
    →erase(c1,s1,g,c4)
trans1(c1,s1,c2,s2),loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3),loc(c1,g)→edge(c1,s1,g,c2,s2)
trans2(c1,s1,c2,s2,s3),erase(c2,s2,g2,c3),loc(c1,g1),
    or(g1,g2,g)→edge(c1,s1,g,c3,s3)
```

Figure 2.3: Rules corresponding to the `erase` relation used to construct the reach graph, and the `edge` relation of the reach graph.

The definitions of the `erase` and `edge` relations can be readily written as rules. These rules are shown in Figure 2.3.

Detecting good paths. Checking that the BPDS accepts the empty language amounts to checking that the resulting reach graph has no good paths. To find good paths in the reach graph we use the algorithm presented in [13, Figure 4] but ignore consideration of resource labels by the algorithm. The algorithm uses depth first search and is linear in the number of edges in the reach graph.

2.3 Efficient Algorithm for Computing the Reach Graph

This section describes the generation of a specialized algorithm and datastructures for computing the reach graph from the rules shown in the previous section, as well as analyzing precisely the time complexity for computing the reach graph and expressing the complexity in terms of characterizations of the facts—the parameters characterizing the BPDS.

2.3.1 Generation of efficient algorithms and data structures

We transform the extended Datalog rules into an efficient implementation using the method in [57] for Datalog rules. The method has three steps.

- `Step 1:` transform the least fixed point (LFP) specification of the extended Datalog rules into a `while`-loop.
- `Step 2:` transform expensive set operations in the loop into incremental operations.

- `Step 3:` design appropriate data structures for each set, so that operations on it can be implemented efficiently.

These three steps correspond to dominated convergence [21], finite differencing [63], and real-time simulation [62], respectively, as studied by Paige et al.

Auxiliary relations. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. The transformation introduces auxiliary relations with necessary arguments to combine two hypotheses at a time. We repeatedly apply the following transformations to each rule with more than two hypotheses until only rules with at most two hypotheses are left. We replace any two hypotheses of the rule, say $P_i(X_{i1}, ..., X_{ia_i})$ and $P_j(X_{j1}, ..., X_{ja_j})$ by a new hypothesis, $Q(X_1, ..., X_a)$, where Q is a fresh relation, and X_k's are variables in the arguments of P_i or P_j that occur also in the arguments of other hypotheses or the conclusion of this rule. We add a new rule: $P_i(X_{i1}, ..., X_{ia_i}) \wedge P_j(X_{j1}, ..., X_{ja_j}) \rightarrow Q(X_1, ..., X_a)$.

```
1.   loc(c1,g)∧trans0(c1,s1,c2)→erase(c1,s1,g,c2)
2.   loc(c1,g1)∧trans1(c1,s1,c2,s2)→gtrans1(c1,g1,s1,c2,s2)
3.   gtrans1(c1,g1,s1,c2,s2)∧erase(c2,s2,g2,c3)
       →gtrans1e(c1,s1,c3,g1,g2)
4.   gtrans1e(c1,s1,c3,g1,g2)∧or(g1,g2,g)→erase(c1,s1,g,c3)
5.   loc(c1,g1)∧trans2(c1,s1,c2,s2,s3)
       →gtrans2(c1,g1,s1,c2,s2,s3)
6.   gtrans2(c1,g1,s1,c2,s2,s3)∧erase(c2,s2,g2,c3)
       →gtrans2e(c1,s1,s2,c3,g1,g2)
7.   gtrans2e(c1,s1,s2,c3,g1,g2)∧erase(c3,s2,g3,c4)
       →gtrans2ee(c1,s1,c4,g1,g2,g3)
8.   gtrans2ee(c1,s1,c4,g1,g2,g3)∧or(g1,g2,g4)
       →gtrans2ee_or(c1,s1,c4,g3,g4)
9.   gtrans2ee_or(c1,s1,c4,g3,g4)∧or(g4,g3,g)
       → erase(c1,s1,g,c4)
10.  gtrans1(c1,g,s1,c2,s2)→ edge(c1,s1,g,c2,s2)
11.  gtrans2(c1,g,s1,c2,s2,s3)→ edge(c1,s1,g,c2,s2)
12.  gtrans2e(c1,s1,s2,c2,g1,g2)∧or(g1,g2,g)
       →edge(c1,s1,g,c2,s2)
```

Figure 2.4: The reach graph expressed in rules with at most two hypotheses.

The resulting rule set for constructing the reach graph is shown in Figure 2.4. Several auxiliary relations have been introduced. The relations `gtrans1` and `gtrans2` represent transitions like `trans1` and `trans2` respectively, but an extra argument indicates whether the transitions start at a good control location. The relations `gtrans1e` and `gtrans2e`, represent runs of the BPDS starting with a transition `trans1` and `trans2`

respectively, followed by a run represented as a fact of the `erase` relation. The facts `gtrans1e(c1,s1,c2,g1,g2)` and `gtrans2e(c1,s1,s2,c2,g1,g2)` represent runs from configuration $(c1, s1)$ to configurations $(c2, \epsilon)$ and $(c2, s2)$ respectively, where `g1` and `g2` indicate, respectively, whether the first control location in the run is good and whether the rest of the run visits a good control location. The relation `gtrans2ee` represents runs consisting of one transition and two runs expressed as facts of the `erase` relation. The fact `gtrans2ee(c1,s1,c2,g1,g2,g3)` stands for a run from configuration $(c1, s1)$ to configuration $(c2, \epsilon)$; the arguments `g1`, `g2`, and `g3` are booleans indicating respectively, whether the first control location in the run is good, and whether the remaining two parts of the run visit a good control location. The relations `gtrans1ee_or` and `gtrans2ee_or` represents runs like `gtrans1ee` and `gtrans2ee`, except with two boolean arguments combined using logical or.

Fixed-point specification and `while`-loop. We represent a relation of the form `Q(a1, a2, ... , an)` using tuples of the form `[Q,a1,a2,...,an]`. S `with` X and S `less` X denote $S \cup \{X\}$ and $S - \{X\}$, respectively. We use the notation $\{X : Y_1$ `in` $S_1, \ldots, Y_n$ `in` $S_n | Z\}$ for set comprehension. Each Y_i enumerates elements of S_i; for each combination of values $Y_1, \ldots, Y_n$, if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If Z is omitted, it is implicitly the constant `true`.

$\{[X_1\ Y_1] \ldots [X_n\ Y_n]\}$ denotes a map that maps X_1 to Y_1, ..., X_n to Y_n. $dom(E)$ denotes the domain set of map E, i.e., $\{X : [X\ Y]$ `in` $E\}$. $E\{X\}$ denotes the *image set* of X under map E, i.e., $\{Y : [X\ Y]$ `in` $E\}$. $E\{X\} := S$ denotes setting the image set $E\{X\}$, of X under map E, to S. `LFP`(S_0, F) denotes the smallest set S that satisfies the conditions $S_0 \subseteq S$ and $F(S) = S$.

The algorithm is expressed using standard control constructs `while`, `for`, `if`, and `case`. Program block structure is indicated by indentation. We abbreviate $X := X$ `op` Y as X `op`$:= Y$.

The input to the algorithm is the given BPDS represented by a set `bpds` of facts. We define `rbpds` to be the set of facts in `bpds` represented as tuples as described above.

We use set `bpds` for the set of facts representing the BPDS.

```
rbpds = {[loc c1 g] :  loc(c1,g) in bpds} ∪
 {[trans0 c1 s1 c2] :  trans0(c1,s1,c2) in bpds} ∪
 {[trans1 c1 s1 c2 s2] :  trans1(c1,s1,c2,s2) in bpds} ∪
 {[trans2 c1 s1 c2 s2 s3] :  trans0(c1,s1,c2,s2,s3) in bpds}
```

Given any set `R` of facts, and a extended Datalog rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)`, referred to as *resultset*, be the set of all facts that can be inferred by that rule given the facts in `R`. For example,

```
2gtrans1(R) = {[gtrans c1 s1 g c2 s2]
       :  [loc c1 g] in R and [trans1 c1 g s1 c2 s2] in R},
10edge(R) = {[edge c1 s1 g c2 s2]
       :  [gtrans1 c1 g s1 c2 s2] in R}.
```

The meaning of the give facts and the rules used to compute the reach graph is `LFP({},F)`, where `F(R)` is the union of all resultsets, that is:

`LFP({},F)`, where `F(R)` $=$ `rbpds` $\cup$ `1erase(R)` $\cup$ `2gtrans1(R)` $\cup$
`3gtrans1e(R)` $\cup$ `4erase(R)` $\cup$ `5gtrans2(R)` $\cup$ `6gtrans2e(R)` $\cup$
`7gtrans2ee(R)` $\cup$ `8gtrans2ee_or(R)` $\cup$ `9erase(R)` $\cup$
`10edge(R)` $\cup$ `11edge(R)` $\cup$ `12edge(R).`

This least-fixed point specification of computing the reach graph is transformed into the following `while`-loop:

```
R := {};
while exists x in F(R) - R                    (2.1)
   R with := x;
```

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop.

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in constructing the reach graph are all resultsets and a workset `W`. We use fresh variables to hold each of their respective values and maintain an invariant for each of the resultsets, in addition to one for the workset: `W = rbpds + F(R) - R.`

```
Ibpds = rbpds, I1erase = 1erase(R),
I2gtrans1 = 2gtrans1(R), I3gtrans1e = 3gtrans1e(R),
I4erase = 4erase(R), I5gtrans2 = 5gtrans2(R),
I6gtrans2e = 6gtrans2e(R), I7gtrans2ee = 7gtrans2ee(R),
I8gtrans2ee_or = 8gtrans2ee_or(R), I9erase = 9erase(R),
I10edge = 10edge(R), I11edge = 11edge(R), I12edge = 12edge(R),
W = F(R) - R.
```

`W` serves as the workset. As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I2gtrans1`. `I2gtrans1` is the value of the set formed by joining elements from the set of facts of the `loc` and `trans1`

relations. `I2gtrans1` can be initialized to {} with the initialization `R = {}`. To update `Igtrans1` incrementally with update `R with:= x`, if `x` is of the form `[loc c1 g]` we consider all matching tuples of the form `[trans1 c1 s1 c2 s2]` and add the tuple `[gtrans1 c1 g s1 c2 s2]` to `I2gtrans1`. To form the tuples to add, we need to efficiently find the appropriate values of variables that occur in `[trans1 c1 s1 c1 s2]` tuples, but not in `[loc c1 g]`, i.e., the values of `s1, c2,` and `s2`, so we maintain an auxiliary map that maps `[c1]` to `[s1 c2 s2]` in the variable `I2gtrans1_trans1` shown below. Symmetrically, if `x` is a tuple of `[trans1 c1 s1 c2 s2]`, we need to consider every matching tuple of `[loc c1 g]` and add the corresponding tuple of `[gtrans1 c1 g s1 c2 s2]` to `I2gtrans1_loc`.

```
I2gtrans1_trans1 = {[[c1] [s1 c2 s2]] :
                        [trans1 c1 s1 c2 s2] in R},
I2gtrans1_loc = {[[c1] [g]] :  [loc c1 g] in R}.
```

The first set of elements in auxiliar maps is referred to as the *anchor* and the second set of elements as the *nonanchor*.

Thus, the algorithm can directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, and it considers each combination only once. Similar auxiliary maps are maintained for all maintained invariants that are formed by joining elements from two relations.

All variables holding the values of expensive computations listed above and auxiliary maps are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. When `R` is {}, `Ibpds = rbpds`, all auxiliary maps are initialized to {}, and `W = Ibpds`. When a fact is added to `R` in the loop body, the variables are updated. We show the update for the addition of a fact of relation `trans1` only for `I2gtrans1` invariant and `I2gtrans1_loc` auxiliary map , since other facts and updates to the variables and auxiliary maps are processed in the same way. The notation `E{Ys}`, where `E = {[Ys Xs]}` is an auxiliary map, is used to access all matching tuples of `E` and return all matching values of `Xs`.

```
case of x of [loc c1 g]:
  I2gtrans1 ∪:= {[gtrans1 c1 g s1 c2 s2]
        :  [s1 c2 s2] in I2gtrans1_trans1{[c1]}};
  W ∪:= {[gtrans1 c1 g s1 c2 s2]                          (2.2)
        :  [s1 c2 s2] in I2gtrans1_trans1{[c1]}
        | [gtrans1 c1 g s1 c2 s2] notin R};
  I2gtrans1_loc with:= [[c1] [g]];
```

Adding these initializations and updates, and other similar ones for the other cases, and replacing `F(R) - R` with `W` in (2.1), we obtain the following complete code:

```
initialization;
R:={};
while exists x in W:
   update using (2.2) and
   other similar updates for the other cases;
   W less:= x;
   R with:= x;
```

We next eliminate dead code. To compute the resultset R, only W and the auxiliary maps are needed; the invariants maintained for other resultsets, such as I2gtrans1 and I10edge, are dead because F(R)-R in the while loop was replaced with W. We eliminate them from the initialization and updates. For example, eliminating them from the updates in (2.2), we get:

```
case of x of [loc c1 g]:
 W ∪:= {[gtrans1 c1 g s1 c2 s2]
       :  [s1 c2 s2] in I2gtrans1_trans1{[c1]}
       | [gtrans1 c1 g s1 c2 s2] notin R};
 I2gtrans1_loc with:= [[c1] [g]];
```
(2.3)

We clean up the code to contain only uniform operations and set elements. We decompose R and W into several sets, each corresponding to a single relation that occurs in the rules. R is decomposed to Rtrans0, Rtrans1, Rtrans2, Rloc, Rerase, Rgtrans1, Rgtrans1e, Rgtrans2, Rgtrans2e, Rgtrans2ee, Rgtrans2ee_or, and Redge. W is decomposed in the same way. This decomposition lets us eliminate relation names from the first component of tuples, with appropriate changes to the while clause and case clauses. Then, we apply the following three sets of transformations.

(i) Transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a for-loop that adds the elements one at a time. For example, lines 2-4 of (3) are transformed into:

```
for [s1 c2 s2] in I2gtrans1_trans1{[c1]}:
 if [c1 g s1 c2 s2] notin Rgtrans1:
  Wgtrans1 with:=[c1 g s1 c2 s2]
```

(ii) Replace tuples and tuple operations with maps and map operations. Specifically, replace all for loops as follows:

```
for [s1 c2 s2] in I2gtrans1_trans1{[c1]}:
 if [c1 g s1 c2 s2] notin Rgtrans1:
  Wgtrans1 with:=[c1 g s1 c2 s2]
```

is transformed into:

```
for [c1] in dom(I2gtrans1_trans1):
  for [s1 c2 s2] in I2gtrans1_trans1{[c1]}:
    if [c1 g s1 c2 s2] notin Rgtrans1:
      Wgtrans1 with:=[c1 g s1 c2 s2]
```

We replace `while` loops similarly. Also, for each membership in a map test, we replace $[X\ Y]$ `notin` M with Y `notin` $M\{X\}$. For example, the membership test `[c1 g s1 c2 s2] notin Rgtrans1` is replaced with `[g s1 c2 s2] notin Rgtrans1`$\{$`c1`$\}$.

Each addition to a map M `with:=` $[X\ Y]$ is replaced with $M\{X\}$ `with`$:= Y$. For example, the addition to the workset `Wgtrans1`.

```
Wgtrans1 with:= [c1 g s1 c2 s2]
```

is replaced with

```
Wgtrans1{c1} with:= [g s1 c2 s2].
```

(iii) Test for membership before adding or deleting an element of a set. Specifically, we replace each statement S `with`$:= X$ with `if` X `notin` S `then` S `with`$:= X$.

Note that when removing an element from a workset, the membership test is unnecessary, since the element is retrieved from the workset. Also, when adding an element to a resultset, the membership test is unnecessary, since elements are moved from the corresponding workset to the resultset one at a time, and each element is put in the workset and thus in the resultset only once.

Data structures. After the above transformations, each firing of a extended Datalog rule involves a constant number of set operations. Since each set operation takes worst-case constant time in the generated code, as described below, each firing takes worst-case constant time. Next we describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M\{X\}$, element retrieval `for` X `in` S and `while exists` X `in` S, membership test X `in` S and X `notin` S, element addition S `with` X, and element deletion S `less` X. Membership test and computing image set are called *associative access*.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that are traversed by loops and both arrays and linked lists for sets that have both operations.

- `resultsets:` Resultsets are represented by nested array structures. A resultset containing tuples with `a` components is represented using an `a`-level nested array structure. The first level is an array indexed by values in the domain of the first component of the resultset; the k-th element of the array is null if there is no tuple in the resultset whose first component has value `k`, and otherwise is `true` if `a=1`, and otherwise is recursively an `(a-1)`-level nested array structure for the remaining components of tuples in the resultset whose first component has value `k`.

- `worksets:` Worksets corresponding to relations that occur in the conclusions of rules are represented by arrays and linked lists. Each workset is represented the same way as the corresponding resultset with two additions. First, for each array we add a linked list containing indices of non-null elements of the array. Second, to each linked list we add a tail pointer, i.e., a pointer to the last element, so the list can be used as a queue. One or more records are used to put each array, linked list, and tail pointer together. Each workset corresponding to a relation that does not occur in the conclusion of any rule, is represented simply as a nested queue structure (without the underlying arrays), one level for each component of the tuples, linking the elements (instead of array indices) directly.

- `auxiliary maps:` Auxiliary maps are implemented as follows. Each auxiliary map for a relation that appears in an extended Datalog rule's conclusion uses a nested array structure for all components of the tuples and additionally linked lists for each non-anchor component.Each auxiliary map for a relation that does not appear in the conclusion of any rule uses a nested array structure for the anchor components, and nested linked-lists for the non-anchor components.

2.3.2 Complexity analysis of the model checking problem

We analyze the time complexity of the model checking problem by carefully bounding the number of facts actually used by the rules. For each rule we determine precisely the number of facts processed by it, avoiding approximations that use the sizes of individual argument domains.

Calculating time complexity. We first define the size parameters used to characterize relations and analyze complexity. For a realtion `r` we refer to the number of facts of `r` that are given or can be inferred as `r`'s *size*. The parameters `#trans0`, `#trans1` and `#trans2` denote the number of transitions of the form $((c1, s1), (c2, \epsilon))$, $((c1, s1), (c2, s2))$, and $((c1, s1), (c2, s2s3))$, respectively; `#trans` denotes the total number of transitions. The parameters `#gtrans1` and `#gtrans2` denote the number of facts of relations `gtrans1` and `gtrans2`, where `#gtrans1`=`#trans1` and `#gtrans2`=`#trans2`. Parameters `#gtrans1e` and `#gtrans2e` denote the relation sizes — `#trans1` $*$ `#target_loc_trans0`, and `#trans2` $*$ `#target_loc_trans0`, respectively, and `#gtrans2ee` denotes the corresponding relation size equal to `#trans2` $*$

#target_loc_trans0^2. The parameter #erase denotes the number of facts in the erase relation; #erase.4/123 denotes the number of different values the forth argument of erase can take for each combination of values of the first three arguments. In the worst case, this is the number of control locations $c2$ such that a transition of the form $((c1, s1), (c2, \epsilon))$ exists in the automaton. We use #target_loc_trans0 to denote this number.

The time complexity for the set of rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule r, r.#firedTimes stands for the number the number of firings for the rule is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts which make the two hypotheses simultaneously true. The total time complexity is time for reading the input, i.e., O(#trans + #loc), plus the time for applying each rule, shown in the second column in the table of Figure 2.5.

rule no	time complexity	time complexity bound
1	min(#trans0*1,#loc*#trans0.23/1)	#trans0
2	min(#loc*#trans1.234/1,#trans1*1)	#trans1
3	min(#gtrans1*#erase.4/123, #erase*#gtrans1.12/34)	#trans1*#target_loc_trans0
4	min(#gtrans1e*1, 1*#gtrans1e)	#trans1*#target_loc_trans0
5	min(#loc*#trans2.2345/1,#trans2*1)	#trans2
6	min(#gtrans2*#erase.4/123, #erase*#gtrans2.12/345)	#trans2*#target_loc_trans0
7	min(#gtrans2e*#erase.4/123, #erase*#gtrans2e.12/345)	#trans2*#target_loc_trans0^2
8	min(#gtrans2ee*1,1*#gtrans2ee)	#trans2*#target_loc_trans0^2
9	min(#gtrans2ee_or*1,1*#gtrans2ee_or)	#trans2*#target_loc_trans0^2
10	min(#gtrans2ee_or*1,1*#gtrans2ee_or)	#trans2*#target_loc_trans0^2
11	#gtrans1	#trans1
12	#gtrans2	#trans2
13	min(#gtrans2e*1,1*#gtrans2e)	#trans2*#target_loc_trans0

relation	time complexity
erase	O(#trans0 + #trans1*#target_loc_trans0 + #trans2*#target_loc_trans0^2)
edge	O(#trans1 + #trans2*#target_loc_trans0)

Figure 2.5: Time complexity of computing the reach graph.

Time complexity of model checking PDS. Time complexity for processing each of the rules and computing the erase and edge relations is shown in the second table of Figure 2.5. After the reach graph has been computed, good cycles in the reach graph can be detected in time linear in the size of the reach graph, i.e., O(#edge). Thus, the asymptotic complexity of the model checking problem is dominated by the time complexity of computing the erase relation.

For a BPDS, product of $P = \{C_P, S_P, T_P\}$ where $|C_P| = 1$, and $B = \{C_B, L_B, T_B, C0_B, G_B\}$, #target_loc_trans0$\leq |C_B|$, and #trans2$\leq |T_P| * |T_B|$. For such a PDS, $O(|T_P| * |T_B| * |C_B|^2)$ is the worst case time complexity of computing the erase relation and $O(|T_P| * |T_B| * |C_B|)$ is the worst case time complexity for computing

the `edge` relation. Since only $|T_P|$ is dependent on the size of `P`, time complexity is linear in the size of the `P` and cubic in the size of `B`.

2.3.3 Performance

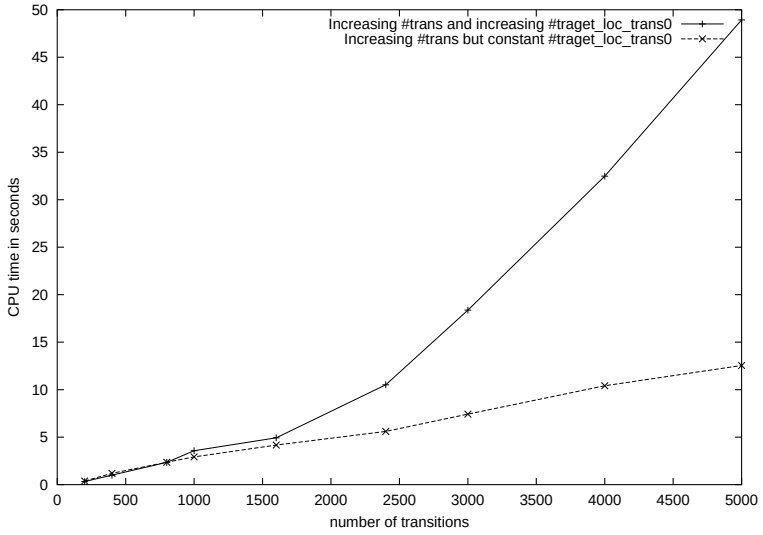


Figure 2.6: Results for computing the reach graph for the BPDS.

We tested the performance of our reach graph construction algorithm on two sets of BPDS consisting of BPDS with increasing `#trans`. BPDS in one set also had increasing `#target_loc_trans0`, while BPDS in the second set had constant `#target_loc_trans0`. The time complexity for computing reach graphs for BPDS in the first set is as shown in Figure 2.5. However, for automata in the second set time complexity should be linear — `O(#trans)`. If the PDS corresponds to a program, `#target_loc_trans0` is proportional to the total number of return points of procedures in the program. Thus, our test data corresponds to checking if a property holds on programs with an increasing number of statements and procedure calls, and programs with an number of statements, but constant number of procedures.

Results of the experiment are shown in Figure 2.6 and confirm our analysis. We used generated python code in which each operation on set elements is guaranteed to be constant time on average using default hashing in python. Running times are measured in seconds on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running SunOS 5.8. Running times are the average over ten runs.

2.4 Discussion

The problem of LTL model checking of PDS has been extensively researched, especially model checking PDS induced by CFGs of programs. The model checking problem for context-free and pushdown processes is explored in [19]. The design and implementation of Bebop: a symbolic model checker for boolean programs, is presented in [9]. Burkart and Steffen [20] present a model checking algorithm for modal mu-calculus formulas. For a PDS with one control state, a modal-mu calculus formula of alternation depth k can be checked in time $O(n^k)$, where n is the size of the PDS. The works [37, 36, 35, 18] describe efficient algorithms for model checking PDSs. Alur et al. [7] and Benedikt et al. [16] show that state machines can be used to model control flow of sequential programs. Both works describe algorithms for model checking PDS that have time complexity cubic in size of the BA and linear in size of the PDS; these works combine forward and backward reachability and obtain complexity estimations by exploiting this mixture. Esparza et al. [35] estimate time complexity of solving the model checking problem to be $\texttt{O(n*m}^3\texttt{)}$ for model checking PDS with one state only, where `n` is the size of the PDS and `m` is the size of the property BA [35]. While this is also linear in the size of the PDS, our time complexity analysis is more precise and automatic.

The algorithm derived in this work is essentially the same as the one in [35]. What distinguishes our work is that we use a novel implementation strategy for the model checking problem that combines an intuitive definition of the model checking problem in rules [13] and a systematic method for deriving efficient algorithms and data structures from the rules [57], and arrives at an improved complexity analysis. The time complexity is calculated directly from the Datalog rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules.

An implementation of the model checking problem in logical rules is presented in [13]. The rules are evaluated using the XSB system [76]. Thus, the efficiency of the computation is highly dependent on the order of hypotheses in the given rules. Our implementation is drastically different, as it finds the best order of hypotheses in the rules automatically. We do not employ an evaluation strategy for Datalog, but generate a specialized algorithm and implementation directly from the rules.

In this chapter, we presented an efficient algorithm for LTL model checking of PDS. We showed the effectiveness of our approach by using a precise time complexity analysis, along with experiments. These results show that our model checking algorithm can help accommodate larger PDS and properties. Our work is potentially a contribution not only to the model checking problem, since the idea behind the `erase` relation and the reach graph is more universal than model checking PDS. Variants of the `erase` relation are used in data flow analysis techniques, as described in [71] and related work. Applications of model checking in dataflow analysis are presented in [82, 81]. It is a topic of future research to apply our method to dataflow analysis problems.

Chapter 3

Efficient Type Inference for Secure Information Flow

3.1 Introduction

Protection of the confidentiality and privacy of data is becoming increasingly important. In addition to controlling direct access to information, it is also essential to control information flow, especially in untrusted code. Static analysis of information flow in programs allows for fine-grained control without runtime overhead.

Denning's work [29, 30] was the pioneering work in the field. It proposed a lattice model that could be used to verify secure information flow in programs. In this model, security classes are ordered in a lattice, and program variables and data are each assigned a security class. The basic security requirement is absence of information flow from higher to lower security classes. Security classes can indicate both the level of secrecy and the level of integrity of data.

Based on Denning's lattice model of information flow analysis, several type-based approaches have been developed [80, 88, 64, 1, 10]. In these works the security properties are formulated as type systems — formal systems of typing rules used to reason about information flow properties of programs. The work of Volpano, Irvine and Smith [80, 88] which formulates Denning's lattice model as a type system and shows it to be sound. Information flow is guaranteed to be secure for a program if the program type checks correctly. A decade after this type-based approach was introduced, the first algorithm for information flow analysis using a type system was proposed by Deng and Smith [28]. The algorithm's running time is quadratic in the size of the given program.

In this chapter, we describe the design, analysis, and implementation of an efficient algorithm for information flow analysis expressed using the type system presented by Volpano et al. in [88]. Our algorithm is linear in the size of the given program.

Given a program and an environment of security classes for information accessed by the program, the algorithm checks whether the program is well typed, i.e., there is no information of higher security classes flowing into places of lower security classes, by inferring the highest or lowest security class as appropriate for each program node. We express the analysis as a set of extended Datalog rules based on the typing and subtyping rules, and we use a systematic method to generate specialized algorithms and data structures directly from the extended Datalog rules. Our extended Datalog rules are Datalog rules with negation and external functions. The method described in [57] is used to generate specialized algorithms and data structures and complexity formulas for the extended Datalog rules. Given a program and an environment of security types, the algorithm infers minimum or maximum security types, as appropriate, for each program node, such that the program type checks correctly. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types. The generated implementation uses a combination of linked and indexed data structures to represent program nodes, environments, and types. The implementation employs an incremental approach that considers one program node at a time. The running time is optimal for the set of rules we use to specify type inference, in the sense that each combination of instantiations of hypotheses is considered once in `O(1)` time. We thus obtain an efficient type inference algorithm.

The time complexity of the algorithm is linear in the size of the input program, times the height of the lattice of security classes, plus a small overhead for preprocessing the lattice. This complexity is confirmed through our prototype implementation and experimental evaluation on code generated from high-level specifications for real systems.

Our main contributions are:

- A novel implementation strategy for type inference for secure information flow types. The strategy combines an intuitive specification of type inference expressed in extended Datalog rules, and a systematic method for deriving efficient algorithms and data structures from the extended Datalog rules [57].

- Precise and automated time complexity analysis for type inference for secure information flow types. The time complexity is calculated directly from the extended Datalog rules, based on a thorough understanding of the algorithm and data structures generated, reflecting the complexities of implementation back into the extended Datalog rules.

The rest of this chapter is organized as follows. Section 2 reviews the lattice model of analyzing information flow in programs and the type system for secure flow analysis [88], and defines the problem of type inference for secure information flow. Section 3 expresses type inference in extended Datalog rules, describes generation of an efficient algorithm

and data structure from the extended Datalog rules, and discusses error reporting. Section 4 presents the time complexity analysis for the generated algorithm. Section 5 presents experimental results. Section 6 discusses related work and concludes.

3.2 A Type System for Secure Information Flow

This section reviews the lattice model of information flow [29, 30], and a type system based on it [88].

3.2.1 Lattice model of secure information flow

In the lattice model of information flow [30, 29] security classes form a lattice, denoted by $(SC, \leq)$, comprising a finite set SC of security classes, and a partial order $\leq$. A *security class* is an indication of (i) the level of *secrecy* of the data — how confidential the data is, (ii) the level of *integrity* of the data — how trusted the data is, or (iii) a combination of these two properties. Every program variable is associated with a security class. The security classes of variables are determined statically and do not vary at run time. Every program node is associated with a *certification condition* — a condition relating security classes of neighboring nodes that checks whether the information flow in the node is secure.

Information is considered to *flow* from variable $v1$ into variable $v2$ whenever the value stored in $v1$ affects the value stored in $v2$. Information flow may be explicit or implicit. An *explicit flow* results from assigning the value of a variable to another variable. *Implicit flows* reflect control dependencies. For example, an implicit flow exists from the value of a conditional guard to the branches of the conditional. For example, in the following `if`-statement:

```
if a=0 then b:=1 else b:=0
```

there is an implicit flow from variable `a` to variable `b`, since after the statement has been executed, by the value of variable `b` we can determine whether the value of `a` is `0`.

The *flow relation* $\rightarrow$ is a binary relation on security classes that indicates the permitted information flows. For security classes x and y, if $x \rightarrow y$ then flow from variables of class x to variables of class y are permitted and called *secure flows*. In the lattice model, the flow relation is: $x \rightarrow y$ if $x \leq y$.

The lattice model of information flow makes it possible to check conditions on both explicit and implicit information flows by checking the certification conditions on program constructs.

3.2.2 Type system for secure flow analysis

The type system for secure information flow [88, 28] is based on Denning's lattice model of information flow. The type system guarantees that explicit and implicit flows are secure.

The security types are assumed to form a partial order, denoted by $\leq$. The strict order induced by the partial order $\leq$ is denoted by $<$. The partial order relation $\leq$ is extended to a *subtype* relation, denoted by $\subseteq$; the strict order induced by the partial order $\subseteq$ is denoted by $\subset$.

Two levels of types are used:

- *data types*, denoted by τ, and are the security classes in the lattice;
- *phrase types*, denoted by ρ, include (i) data types τ, given to expressions; (ii) variable types $\tau\ var$ given to variables; (iii) command types $\tau\ cmd$ given to commands; and (iv) array types $\tau 1\ arr\ \tau 2$ given to arrays.

A variable of type $\tau\ var$ stores information whose security class is type τ or lower. A command of type $\tau\ cmd$ contains assignments only to variables of type τ or higher. An array of type $\tau 1\ arr\ \tau 2$ contains data of security type $\tau 1$ and has length of security type $\tau 2$. Every array type is subject to the constraint $\tau 2 \leq \tau 1$. Intuitively, an array's contents include its length, so the security level of the contents should be at least as high as that of the length. To ensure that this condition holds for all global arrays, we need to check that for all locations `l`, if the type of `l` has the form $\tau 1\ arr\ \tau 2$, then $\tau 2 \leq \tau 1$.

A *phrase* is an expression or a command generated by the following grammar:

$$
\begin{array}{lll}
expression\ e & ::= & x \mid l \mid n \mid e1 + e2 \mid e1 - e2 \mid \\
 & & e1 = e2 \mid e1 < e2 \mid a[e1] \mid a.\texttt{length} \\
command & c ::= & e1 := e2 \mid \\
 & & c1; c2 \mid \\
 & & \texttt{if}\ e\ \texttt{then}\ c1\ \texttt{else}\ c2 \mid \\
 & & \texttt{while}\ e\ \texttt{do}\ c \mid \\
 & & \texttt{letid}\ x := e\ \texttt{in}\ c \mid \\
 & & a[e1] := e2 \mid \texttt{allocate}\ a[e1]
\end{array}
$$

Expressions include identifiers x, locations l, integer literals n, arithmetic expressions, and array expressions. Commands of the forms shown above are, respectively, assignments, compositions, conditional commands, local variable (i.e., identifier) declarations, array assignment, and array allocation. The array allocation command `allocate` $a[e1]$ allocates an array a of length $e1$.

Typing judgments are of the form $\lambda; \gamma \vdash p : \rho$, where γ is a mapping of identifiers to security types and λ is a mapping of locations to security types. The meaning of this typing judgment is that phrase p has type ρ, if identifiers and locations in p have security types as assigned in γ and λ.

(BASE) $\dfrac{r \leq r1}{\vdash r \subseteq r1}$

(REFLEX) $\vdash \rho \subseteq \rho$

(TRANS) $\dfrac{\vdash \rho \subseteq \rho1, \vdash \rho1 \subseteq \rho2}{\vdash \rho \subseteq \rho2}$

(CMD)$^-$ $\dfrac{\vdash \rho \subseteq \rho1}{\vdash \rho1\ cmd \subseteq \rho\ cmd}$

(SUBTYPE) $\dfrac{\lambda; \gamma \vdash p : \rho \quad \vdash \rho \subseteq \rho1}{\lambda; \gamma \vdash p : \rho1}$

Figure 3.1: Subtyping rules.

$\gamma[x : \rho]$ denotes a modification of γ that assigns type ρ to identifier x and leaves other identifier-type mappings in γ unchanged.

A typing rule has the form:

$$\frac{J_1\ J_2\ ...\ J_n}{J_{n+1}}$$

where J_i's are typing judgments. The typing judgments above the line are *hypotheses*, and the typing judgment below the line is the *conclusion*. The rule infers the typing judgment in its conclusion, if all its hypotheses hold. A judgement holds if it is an axiom or can be inferred by some typing rule.

The rules for subtyping are shown in Figure 3.1. The typing rules are shown in Figure 3.2. A typing rule for only one arithmetic expression is shown, since rules for the other arithmetic expressions are defined in the same way. The typing rules correspond directly to certification conditions in the lattice model.

The typing rule ARITH is used to infer the types of arithmetic expressions. The rule says that if expressions e and $e1$ are of security type τ, then the type of the expression $e+e1$ is also τ. Note that if the types of e and $e1$ are different, it may be possible to make them the same by coercing one or both of them to higher security types, using the subtyping rules.

The ASSIGN rule checks the explicit information flow in assignment commands. The expressions e and $e1$ must have the same security type τ. If this is the case, the assignment command is given type $\tau\ cmd$. If the types of e and $e1$ are not the same, and the type of $e1$ is lower than that of e, it may be possible to coerce the type of $e1$ to the type of e. However, if the type if $e1$ is higher than that of e, the assignment command causes information to flow from a high security type to a place of low security class, and the command is untypable.

(LITERAL) $\lambda; \gamma \vdash n : \tau$

(ID) $\lambda; \gamma \vdash x : \tau\ var$ if $\gamma(x) = \tau\ var$

(LOC) $\lambda; \gamma \vdash l : \tau\ var$ if $\lambda(l) = \tau$

(R-VAL) $$\frac{\lambda; \gamma \vdash e : \tau\ var}{\lambda; \gamma \vdash e : \tau}$$

(ARRLEN) $$\frac{\lambda(a) = \tau 1\ arr\ \tau 2}{\lambda; \gamma \vdash a.\texttt{length}:\ \tau 2}$$

(ARRACCESS) $$\frac{\lambda(a) = \tau 1\ arr\ \tau 2,\ \lambda; \gamma \vdash e :\ \tau 3}{\lambda; \gamma \vdash a[e] : \tau 1 \vee \tau 3}$$

(ARITH) $$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash e1 : \tau}{\lambda; \gamma \vdash e + e1 : \tau}$$

(ASSIGN) $$\frac{\lambda; \gamma \vdash e : \tau\ var \quad \lambda; \gamma \vdash e1 : \tau}{\lambda; \gamma \vdash e := e1 : \tau\ cmd}$$

(ARRALLOC) $$\frac{\lambda(a) = \tau 1\ arr\ \tau 2,\ \gamma \vdash e1 :\ \tau 2}{\lambda; \gamma \vdash \texttt{allocate}\ a[e1] : \tau 2\ cmd}$$

(ARRASSIGN) $$\frac{\lambda(a) = \tau 1\ arr\ \tau 2 \quad \lambda; \gamma \vdash e1 :\ \tau 1 \quad \lambda; \gamma \vdash e2 :\ \tau 1}{\lambda; \gamma \vdash a[e1] := e2 :\ \tau 1\ cmd}$$

(SEQUENCE) $$\frac{\lambda; \gamma \vdash c : \tau\ cmd \quad \lambda; \gamma \vdash c1 : \tau\ cmd}{\lambda; \gamma \vdash c; c1 : \tau\ cmd}$$

(IF) $$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau\ cmd \quad \lambda; \gamma \vdash c1 : \tau\ cmd}{\lambda; \gamma \vdash \texttt{if}\ e\ \texttt{then}\ c\ \texttt{else}\ c1 : \tau\ cmd}$$

(WHILE) $$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma \vdash c : \tau\ cmd}{\lambda; \gamma \vdash \texttt{while}\ e\ do\ c : \tau\ cmd}$$

(LETID) $$\frac{\lambda; \gamma \vdash e : \tau \quad \lambda; \gamma[x : \tau\ var] \vdash c : \tau 1\ cmd}{\lambda; \gamma \vdash \texttt{letid}\ x := e\ \texttt{in}\ c : \tau 1\ cmd}$$

Figure 3.2: Typing rules for secure information flow.

The typing rules for `IF` and `WHILE` check whether the implicit flows are secure. These rules require that the guard expressions have the same security types as the commands in the branches or loop body, respectively, since there is an implicit flow from the guard to those commands. In addition, the two commands in the branches of `if`-statements must have the same type. As usual, coercion based on subtyping can help satisfy these constraints.

The `LETID` rule ensures that information flow in local variables declarations is secure. If a local variable x is initialized to the value of expression e of type τ, the identifier-type mapping γ is updated to map variable x to type τ while type checking the body of the

`letid` command.

The ARRACCESS rule checks whether accesses to array elements are secure. If a is of type $\tau 1\ arr\ \tau 2$ and expression e of type $\tau 3$ indicates the index of the element of a being accessed, the array access expression has type $\tau 1 \vee \tau 3$, because there is information flow from both a and e to the result.

The ARRALLOC rule ensures that the information flow in array allocation is secure. If an array of type $\tau 1\ arr\ \tau 2$ is being allocated, and the array's length is equal to the value of expression $e1$, then the type of $e1$ must be $\tau 2$, because there is information flow from $e1$ to the array. If this is the case, the array allocation command is given the type $\tau 2\ cmd$, otherwise the command is untypable.

Given a program, and an environment of security types for locations accessed by the program, *type inference* is the process of inferring all possible types for each program node, if possible, so that the program is well-typed. Otherwise, type errors are reported. If the program is well-typed with respect to the secure information flow type system presented, information flow in the program is guaranteed to be secure.

3.3 Efficient Type Inference Algorithm and Data Structures

This section expresses type inference using extended Datalog rules and describes the generation of a specialized algorithm and data structures for type inference from the extended Datalog rules.

Type inference is generally done by using variables for unknown types of commands and expressions, and collecting constraints, in the form of type inequalities, that the type variables must satisfy for the program to be well-typed. These constraints characterize all typings of the program. The idea of our type inference algorithm is, given types for locations, to infer the lowest or highest security type for each program node, as appropriate, that the node can have in any typing of the program.

We define extended Datalog rules that we use to traverse the syntax tree of the program. The algorithm traverses the program top-down multiple times to infer minimum expression types, and then traverses the program bottom-up once to infer maximum command types.

3.3.1 Expressing type inference in extended Datalog rules

We use the following relations in our extended Datalog rules. We use two relations to map locations and arrays to their security types:

- `locenv(l,t)`: location `l` has type `t`.

- `arrenv(a,t1,t2)`: array `a` contains data of type `t1` and has length of type `t2`.

Together these two relations correspond to λ in the typing rules. The relations used to represent the syntax tree of the input program are:

- `root(c)`: `c` is the outermost command of a program.
- `literal(n)`: `n` is a literal.
- `loc(l)`: `l` is a location.
- `id(x)`: `x` is an identifier.
- `arith(e,e1,e2)`: expression `e` is an arithmetic expression with subexpressions `e1` and `e2` (e.g., `e` is `e1 + e2`).
- `assign(c,x,e)`: `c` is the command $x := e$.
- `if(c,e,c1,c2)`: `c` is the command `if` e `then` $c1$ `else` $c2$.
- `while(c,e,c1)`: `c` is the command `while` e `do` $c1$.
- `sequence(c,c1,c2)`: `c` is the command $c1; c2$.
- `letid(c,x,e,c1)`: `c` is the command `letid` $x := e$ `in` $c1$.
- `arraccess(e,a,e1)`: `e` is the expression $a[e1]$.
- `arrassign(c,a,e1,e2)`: `c` is the command $a[e1] := e2$.
- `arrlen(e,a)`: `e` is the expression $a.$`length`.
- `arralloc(c,a,e1)`: `c` is the command `allocate` $a[e1]$.

The following relations are used to represent inferred types of program nodes and error messages about insecure information flow:

- `type(p,t)`: program node `p` has type `t`. There may be multiple `type` facts for a program node. It is only necessary to keep the one with the highest type inferred so far.
- `htype(c,t)`: the maximum type of command `c` is `t`. The maximum type for a command is the highest type the command can have for the program to type correctly.
- `error(c)`: the program is untypable because there may be insecure information flow in command `c`. A fact of the `error` relation is inferred when an assignment or array assignment statement assigns data to a location or an element of the array, and the data has a higher security type than the location or array. As discussed in Section 3.3. we can give more detailed error messages based on the derivation of each inferred `error(c)` fact, e.g., specifying the command that caused the error.

The functions `Join(t1,t2)` and `Meet(t1,t2)` return, respectively, the least upper bound and the greatest lower bound of two security types `t1` and `t2`. `Join` and `Meet` are defined for any two security types, since the types form a lattice. We can either precompute the least upper and greatest lower bound for each possible pair of security types, or compute them as needed during type inference, possibly with memoization. Efficient algorithms to compute `Meet` and `Join` are presented by Hassan et al. in [5]. The authors present three different algorithms for computing least upper bound and greatest lower bound: one is based on a transitive closure approach, the second is a more space-efficient method, and the last one employs a grouping technique based on modulation — it drastically reduces the code size, while keeping time complexity low. Time complexity of computing the complete least upper bound and greatest lower bound relations for a lattice is $O(s^2 \times \log s)$, where `s` is the size of the lattice. Time complexity for computing least upper bound or greatest lower bound for a single pair of types is $O(\log s)$.

The extended Datalog rules used for type inference are shown in Figures 3.3 and 3.4. The typing rules in Figure 3.2 can be written directly as extended Datalog rules, but efficient analysis needs to follow a predetermined procedure of traversing the program top-down multiple times to infer minimum expression types, and then traversing the program bottom-up once to infer maximum command types using the minimum types for expression. During the top-down traversals at any point in the evaluation we keep only the minimum inferred type for each expression. This is done for efficiency reasons, and it does not affect the correctness of the algorithm since only minimum types for expressions are used to infer maximum types for commands. We have rewritten the rules to embody this procedure. The rules in Figure 3.3 infer minimum types for expressions; the rules in Figure 3.4 infer maximum types for commands.

The rules are sound and complete with respect to the typing and subtyping rules in Section 2. Soundness is the property that if our rules infer a typing, expressed as the `type` relation for expressions and the `htype` relation for commands, then types for expressions and commands in it satisfy the typing rules in Section 2. With the subtyping rules, higher expression types and lower command types also satisfy the rules. The soundness of our type inference algorithm can be proved by a structural induction.

Completeness is the property that if a typing satisfies the typing rules in Section 2, then our rules infer a typing too, and our inferred expression types, expressed in the `type` relation, are the lowest expression types that satisfy the typing rules in Section 2, and our command types, expressed as the `htype` relation, are the highest command types that satisfy the tying rules in Section 2. The completeness of our type inference algorithm can be proved by an induction on derivations that use our rules.

3.3.2 Generation of efficient algorithm and data structures

We transform the extended Datalog rules into an efficient implementation using the method in [57] for Datalog rules. Two small extensions are needed, to handle negation,

```
(ROOT)
1.   root(c)→type(c,bottom)
(LITERAL)
2.   literal(n)→ type(n,bottom)
(LOC)
3.   loc(l),locenv(l,t)→type(l,t)
(ARRLEN)
4.   arrlen(e,a), arrenv(a,t1,t2)→type(e,t2)
(ARRACCESS)
5.   arraccess(e,a,e1),arrenv(a,t1,t2),type(e1,t3)
       →type(e,Join(t1,t3))
(ARITH)
6.   arith(e,e1,e2),type(e1,t1),type(e2,t2)→type(e,Join(t1,t2))
(ASSIGN ID)
7.   assign(c,x,e),id(x),type(e,t1),type(c,t2),type(x,t3)
       →type(x,Join(t1,t2,t3))
(ASSIGN LOC)
8.   assign(c,l,e),loc(l),type(l,t1),type(e,t2),not t2⊆t1→error(c)
9.   assign(c,l,e),loc(l),type(l,t1),type(c,t2),not t2⊆t1→error(c)
(ARRALLOC)
10.  arralloc(c,a,e1),arrenv(a,t1,t2),type(e1,t3),not t3⊆t2→error(c)
(ARRASSIGN)
11.  arrassign(c,a,e1,e2),arrenv(a,t1,t2),type(e1,t3),not t3⊆t1
       →error(c)
12.  arrassign(c,a,e1,e2),arrenv(a,t1,t2),type(e2,t4),not t4⊆t1
       →error(c)
(SEQUENCE)
13.  sequence(c,c1,c2),type(c,t)→type(c1,t)
14.  sequence(c,c1,c2),type(c,t)→type(c2,t)
(IF)
15.  if(c,e,c1,c2),type(e,t1),type(c,t2)→type(c1,Join(t1,t2))
16.  if(c,e,c1,c2),type(e,t1),type(c,t2)→type(c2,Join(t1,t2))
(WHILE)
17.  while(c,e,c1),type(e,t1),type(c,t2)→type(c1,Join(t1,t2))
(LETID)
18.  letid(c,x,e,c1),type(e,t)→type(x,t)
19.  letid(c,x,e,c1),type(c,t)→type(c1,t)
```

Figure 3.3: Extended Datalog rules for inference of minimum expression types and associated command types.

which in our rules simply requires a constant time check, and external functions, such as `Meet` and `Join`.

```
(ASSIGN ID MAX)
20.  assign(c,x,e),id(x),type(x,t)→htype(c,t)
(ASSIGN LOC MAX)
21.  assign(c,l,e),loc(l),type(l,t)→htype(c,t)
(ARRALLOC MAX)
22.  arralloc(c,a,e1),arrenv(a,t1,t2)→htype(c,t2)
(ARRASSIGN MAX)
23.  arrassign(c,a,e1,e2),arrenv(a,t1,t2)→htype(c,t1)
(SEQUENCE MAX)
24.  sequence(c,c1,c2),htype(c1,t1),htype(c2,t2)→htype(c,Meet(t1,t2))
(IF MAX)
25.  if(c,e,c1,c2),htype(c1,t1),htype(c2,t2)→htype(c,Meet(t1,t2))
(WHILE MAX)
26.  while(c,e,c1),htype(c1,t)→htype(c,t)
(LETID MAX)
27.  letid(c,x,e,c1),htype(c1,t)→htype(c,t)
```

Figure 3.4: Extended Datalog rules for inference of maximum command types.

Auxiliary relations. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. For example, the `ARITH` rule is transformed into two rules with two hypotheses each as follows:

```
arith(e,e1,e2),type(e1,t1)→arithType(e,e1,e2,t1)
arithType(e,e1,e2,t1),type(e2,t2)→type(e,Join(t1,t2))
```
(3.1)

One auxiliary relation has been introduced — `arithType(e,e1,e2,t1)`, which denotes that `e` is an expression that performs an arithmetic operation on expressions `e1` and `e2`, and the type of `e1` is `t1`.

Fixed-point specification and `while` loop. The inputs to the algorithm are the given program represented by a set `program` of facts, and the relations `locenv` and `arrenv`, which map locations and arrays to security types. We let `input` be the set of facts in `program`, `locenv`, and `arrenv`, represented as tuples as described above.

```
input =
{[locenv l t]:  locenv(l,t) in locenv}∪
{[arrenv a t1 t2]:  arrenv(a,t1,t2) in arrenv}∪
{[root c]:  root(c) in program}∪
{[literal n]:  literal(n) in program}∪
{[loc l]:  loc(l) in program}∪
{[arrlen e a]:  arrlen(e,a) in program}∪
{[arraccess e a e1]:  arraccess(e,a,e1) in program}∪
{[arith e e1 e2]:  arith(e,e1,e2) in program}∪
{[assign c e1 e2]:  assign(c,e1,e2) in program}∪
{[arralloc c a e1]:  arralloc(c,a,e1) in program}∪
{[arrassign c a e1 e2]:  arraccess(c,a,e1,e2) in program}∪
{[sequence c c1 c2]:  sequence(c,c1,c2) in program}∪
{[if c e c1 c2]:  if(c,e,c1,c2) in program}∪
{[while c e c1]:  while(c,e,c1) in program}∪
{[letid c x e c1]:  letid(c,x,e,c1) in program}
```

Given any set R of facts, and an extended Datalog rule with rule number *n* and with relation *e* in the conclusion, let *ne*(R), be the set of all facts that can be inferred by that rule in one step given the facts in R. Here we use as an example the extended Datalog rules corresponding to the sequence commands. The sets *ne*(R) for other rules are defined in the same way.

```
13type(R) = {[type c1 t]:
        [sequence c c1 c2] in R,
        [type c t] in R}
14type(R) = {[type c2 t]:
        [sequence c c1 c2] in R,
        [type c t] in R}
```
(3.2)

For an auxiliary relation e introduced when splitting a rule with more than two hypotheses, let *e*(R) be the set of facts that can be inferred by the rule defining e in one step given the facts in R.

The meaning of the given facts and the extended Datalog rules used for inferring minimum types for expressions is LFP({},F), where F(R) is the union of the input and the set of all facts that can be inferred by all rules for computing types and errors in one

step given the facts in R, that is:

$$\begin{aligned}&\texttt{LFP}(\{\},\texttt{F}), \text{ where } \texttt{F(R)} = \texttt{input} \cup \\ &\quad \texttt{1type(R)} \cup \texttt{2type(R)} \cup \texttt{3type(R)} \cup \texttt{4type(R)} \cup \\ &\quad \texttt{arraccessArrenv(R)} \cup \texttt{5type(R)} \cup \texttt{arithType(R)} \cup \\ &\quad \texttt{6type(R)} \cup \texttt{assignID(R)} \cup \texttt{assignIdType(R)} \cup \\ &\quad \texttt{assignIdTypeType(R)} \cup \texttt{7type(R)} \cup \texttt{assignLoc(R)} \cup \\ &\quad \texttt{assignLocType(R)} \cup \texttt{8error(R)} \cup \texttt{9error(R)} \cup \\ &\quad \texttt{arrallocArrenv(R)} \cup \texttt{10error(R)} \cup \texttt{arrassignArrenv(R)} \cup \\ &\quad \texttt{11error(R)} \cup \texttt{12error(R)} \cup \texttt{13type(R)} \cup \texttt{14type(R)} \cup \\ &\quad \texttt{ifType(R)} \cup \texttt{15type(R)} \cup \texttt{whileType(R)} \cup \texttt{16type(R)} \cup \\ &\quad \texttt{17type(R)} \cup \texttt{18type(R)} \cup \texttt{19type(R)}\end{aligned} \tag{3.3}$$

The set, O, of newly inferred facts about types is the difference between the above set and input, i.e.,

$$\texttt{O = LFP(\{\},F) - input.}$$

The resulting minimum types for expressions are the set, MIN, of maximum types, one for each expression, in O.

The meaning of the program facts, the resulting facts above, and the extended Datalog rules used for inferring maximum types for commands is LFP({},F'), where F'(R) is the union of MIN, input, and the set of all facts that can be inferred by all rules for computing htypes in one step given the facts in R, that is:

$$\begin{aligned}&\texttt{LFP}(\{\},\texttt{F'}), \text{ where } \texttt{F'(R)} = \texttt{MIN} \cup \texttt{input} \cup \\ &\quad \texttt{assignId(R)} \cup \texttt{20htype(R)} \cup \texttt{assignLoc(R)} \cup \texttt{21htype(R)} \cup \\ &\quad \texttt{22htype(R)} \cup \texttt{23htype(R)} \cup \texttt{sequenceHtype(R)} \cup \texttt{24htype(R)} \cup \\ &\quad \texttt{ifHtype(R)} \cup \texttt{25htype(R)} \cup \\ &\quad \texttt{26htype(R)} \cup \texttt{27htype(R)}\end{aligned} \tag{3.4}$$

The resulting set of maximum types for commands is:

$$\texttt{O' = LFP(\{\},F') - (MIN} \cup \texttt{input).}$$

Efficient algorithms for computing O and O' are designed in the same way, so we show only the derivation of the algorithm for computing O. The least-fixed point expression LFP({},F) is transformed into the following while loop:

```
R := {};
while exists x in F(R) - R:
   R with:= x;
```

The idea behind this transformation is to perform small update operations in each iteration of the `while` loop.

To maintain the result set `O`, which does not contain the input facts, i.e., `O = R-input`, we add to `O` every fact that is added to `R` except for facts that are in `input`. The above `while` loop, plus code to maintain `O`, is:

```
R := {};
O := {};
while exists x in F(R) - R:
   R with:= x;
   if x not in input:
      O with:= x;
```
(3.5)

Incremental computation. Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in type inference are all sets of facts inferred by each rule and a workset `W`. We use fresh variables to hold each of their respective values and maintain an invariant for each of these sets, in addition to one for the workset: `W = F(R) - R`. Here we show the invariants maintained for the sets in (3.2). The rest of the invariants are defined in the same way.

```
I13type = 13type(R)
I14type = 14type(R)
```

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I13type`. `I13type` is the value of the set formed by joining elements from the `sequence` and `type` relations. `I13type` can be initialized to $\{\}$ with the initialization `R = {}`. To update `I13type` incrementally with the update `R with:= x`, if `x` is of the form `[sequence c c1 c2]` we consider all matching tuples of the form `[type c t]` and add each new tuple `[type c1 t]` to `I13type`. To form the tuples to be added, we need to efficiently find the appropriate values of variables that occur in `[type c t]` tuples, but not in `[sequence c c1 c2]`, i.e., the appropriate values of `t`, so we maintain an auxiliary map, called `type1_2`, that maps `c` to `t`. Symmetrically, if `x` is a tuple of the form `[type c t]`, we need to consider every matching tuple of the form `[sequence c c1 c2]` and add the corresponding tuple of the form `[type c1 t]` to `I13type`, so we need to efficiently find the value of variables that occur in `[sequence c c1 c2]` but not in `[type c t]`. Thus, we maintain an auxiliary map `sequence1_23` that maps `c` to `c1` and `c2`. These two auxiliary maps are shown below.

```
type1_2 = {[[c] [t]] :  [type c t] in R}
sequence1_23 = {[[c] [c1 c2]] :  [sequence c c1 c2] in R}
```

Since `R = O` $\cup$ `input`, the above two sets are equivalent to:

```
type1_2 = {[[c] [t]] :  [type c t] in O}
sequence1_23 = {[[c] [c1 c2]]
       :  [sequence c c1 c2] in input}
```

The first set of components in an auxiliary map is referred to as the *anchor* and the second set of elements as the *nonanchor*.

Thus, the algorithm can directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, and it considers each combination only once. Similar auxiliary maps are maintained for all maintained invariants that are formed by joining elements from two relations.

All variables holding the values of expensive computations listed above and all auxiliary maps are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. We show the update for the addition of a fact of relation `sequence` only for `I13type` invariant and `sequence1_23` auxiliary map. Other updates are processed in the same way.

```
case x of [sequence c c1 c2]:
  I13type ∪:= {[type c1 t] :  [t] in type1_2{[c]}};
  W ∪:= {[type c1 t] :  [t] in type1_2{[c]}                (3.6)
                    | [type c1 t] not in O};
  sequence1_23 ∪:= {[[c] [c1 c2]]};
```

Adding these initializations and updates, and other similar ones for the other cases, and replacing `F(R) - R` with `W` in (3.5), we obtain the following complete code:

```
initialization;
R:={};
O:={};
while exists x in W:
   update using (3.6) and other similar
   updates for the other cases
   W less:= x;
   R with:= x;
   if x not in input:
     O with:= x;
```

We next eliminate dead code. To compute the result set `O`, only `input`, `W`, and the auxiliary maps are needed; `R` is dead because all uses of it are replaced with `O` and `input`; the sets for maintaining other invariants, such as `I13type` and `I14type`, are dead because `F(R) - R` in the `while` loop was replaced with `W`. We eliminate them from the

initialization and updates. For example, eliminating them from the updates in (3.6), we get:

```
case x of [sequence c c1 c2]:
  W ∪:= {[type c1 t] :  [t] in type1_2{[c]}
                      | [type c1 t] not in O};
  sequence1_23 ∪:= {[[c] [c1 c2]]};
```
(3.7)

The complete pseudocode for inferring minimum types of expressions and maximum types for commands is in the Appendix.

We clean up the code to contain only uniform operations and set elements. This simplifies data structure design. We decompose `O` and `W` into several sets, each corresponding to one relation in the extended Datalog rules. For example, `O` is decomposed into `Otype` and `Oerror`, where `Otype` contains tuples of the `type` relation in `O` and `Oerror` contains tuples of the `error` relation in `O`. This decomposition lets us eliminate relation names from the first component of tuples, with appropriate changes to the `while` clause and `case` clauses. Then, we apply the following three sets of transformations.

(i) Transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for` loop that adds the elements one at a time. For example, lines 2 and 3 of (3.7) are transformed into:

```
for [t] in type1_2{[c]}:
  if [c1 t] not in Otype:
    Wtype with:= [c1 t];
```

(ii) Replace tuples and tuple operations with maps and map operations. For example, the above `for` loop is transformed into:

```
for [c] in dom(type1_2):
  for [t] in type1_2{[c]}:
    if [c1 t] not in Otype:
      Wtype with:= [c1 t];
```

Transform `while` loops similarly. Also, for each membership in a map test, we replace $[X\ Y]$ `not in` M with Y `not in` $M\{X\}$. For example, the membership test `[c1 t] not in Rtype` is replaced with `t not in Rtype`$\{$`c1`$\}$.

Each addition to a map M `with:=` $[X\ Y]$ is replaced with $M\{X\}$ `with`$:= Y$. For example, the addition to the workset `Wtype`.

```
Wtype with:= [c1 t];
```

is replaced with

`Wtype{c1} with:= t;`.

(iii) Test for membership before adding or deleting an element of a set. Specifically, we replace each statement S `with`$:= X$ with `if` X `not in` S`:` S `with`$:= X$.

Note that when removing an element from a workset, the membership test is unnecessary, since the element is retrieved from the workset. Also, when adding an element to a result set, the membership test is unnecessary, since elements are moved from the corresponding workset to the result set one at a time, and each element is put in the workset and thus in the result set only once.

3.3.3 Informative error reporting

Informative error messages are essential for determining the sources of information flow errors and for fixing these errors and developing secure programs.

We can easily give meaningful error messages by adding rules that keep additional information during type inference. We implement the concepts needed to produce informative error messages, as described by Deng and Smith [28], in extended Datalog rules, and apply the method presented above to generate efficient algorithms and data structures directly from the rules.

Following [28], we use two notions to collect information relevant to information flow errors — the *principal variables* for each expression, and the *security level history* for each variable. The *principal variables* for an expression are a minimum set of variables in the expression that can be used to determine the type of the expression. For example, for the expression `a + b`, where `a` and `b` are variables, the principal variables would be `{a,b}`. Intuitively, the principal variables for an expression can provide an explanation of the type of the expression. The *security level history* of a variable keeps track of the different security levels a variable had during type inference and the principal variables of the expression "responsible" for each change to the variable's security type. The type of a variable can be inferred by use of the `LETID` rules. New types for variables can be inferred by use of the `ASSIGN ID` rule. Each such change in the type of a variable is stored in the security level history of the variable by keeping the old and the new type of the variable, as well as the expression assigned to the variable in the `assign` commenad that caused the changed. The type of each program node can be raised at most `h` times. This history can be helpful for understanding how type errors occurred.

When a command generates a type error, the algorithm can provide the principal variables of the expressions that are part of the command, as well as the security level histories of all variables that are part of these expressions.

We use `prinVar(e,v)` to denote that `v` is a principal variable for expression `e`. The rules for computing principal variables are shown in Figure 3.5.

We use `hist(v1,t,v2)` to denote that, during type inference, the security type of variable `v1` changed to `t`, and `v2` is a principal variable of the expression causing the

```
id(x)→prinVar(x,x)

loc(l)→prinVar(l,l)

arrlen(e,a)→prinVar(e,a)

arraccess(e,a,e1)→prinVar(e,a)
arraccess(e,a,e1),prinVar(e1,v)→prinVar(e,v)

arith(e,e1,e2),prinVar(e1,v)→prinVar(e,v)
arith(e,e1,e2),prinVar(e2,v)→prinVar(e,v)
```

Figure 3.5: Rules for principal variables of expressions.

change. Since the security level of a variable can change only when an assignment to an identifier occurs, we need only one rule to keep the security level history:

```
assign(c,x,e),id(x),type(e,t1),type(c,t2),type(x,t3),
  t3 ⊂ Join(t1,t2,t3),prinVar(e,v)
     →hist(x,Join(t1,t2,t3),v)
```

Error reports are based on the relation `errReport(c,v1,t,v2)`, which means that there may be insecure information flow in command `c`, `v1` is a principal variable of `e` or a variable that transitively caused the security level of a principal variable to change, `t` is a new security level of `v1` for a change recorded in `v1`'s security level history, and `v2` is a variable that directly caused the security level of `v1` to change. The rules defining the `errReport` relation are:

```
error(c),assign(c,l,e),prinVar(e,v1),hist(v1,t,v2)
    →errReport(c,v1,t,v2)
error(c),arralloc(c,a,e),prinVar(e,v1),hist(v1,t,v2)
    →errReport(c,v1,t,v2)
error(c),arrassign(c,a,e1,e2),prinVar(e1,v1),hist(v1,t,v2)
    →errReport(c,v1,t,v2)
error(c),arrassign(c,a,e1,e2),prinVar(e2,v1),hist(v1,t,v2)
    →errReport(c,v1,t,v2)
errReport(c,v1,t1,v2),hist(v2,t2,v3)→errReport(c,v2,t2,v3)
```

The error report consists of all inferred facts of the `errReport` relation in order, together with the inferred types for variables mentioned in `errReport` facts.

3.4 Complexity Analysis

This section presents time and space complexity analysis for our type inference algorithm. We first analyze the complexity of the algorithm in Section 3.2 and then analyze the additional cost of the error reporting described in Section 3.3.

3.4.1 Time complexity

We analyze the time complexity of type inference by carefully bounding the number of facts actually used by the extended Datalog rules. For each rule we determine precisely the number of facts processed by it, avoiding where possible approximations that use the product of the sizes of individual argument domains.

Size parameters. We first define the size parameters used in the complexity analysis. The number of facts of a relation `r` that are given or can be inferred is called `r`'s *size*. The number of nodes in the input program is called the *program size* and is denoted by `p`. For a relation named `r`, `#r` denotes the size of `r`. We also use the following additional size parameters:

- `#array`: number of arrays in the program
- `#cmd`: number of commands in the program
- `#expr`: number of expressions in the program
- `p`: the size of the program, i.e., the number of program nodes
- `s`: the size of the lattice of security types
- `h`: the height of the lattice of security types

Analysis of time complexity. The time complexity for a set of Datalog rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule `r`, the number of firings for the rule is a count of: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts that make the two hypotheses simultaneously true. The total time complexity is time for reading the input, plus the time for applying each logic rule.

It is possible to precompute all values for the functions `Join` and `Meet` in $O(s^2 \times \log s)$ time, and, if we do so, any of them can be looked up on $O(1)$ time. However, this may be unnecessary, since it is possible that not all values of these functions are needed. Therefore, we compute the values of `Join` and `Meet` as needed and memoize already computed values which can be looked up in $O(1)$ time if needed again. The time complexity of

computing Join or Meet for two security types is O(log s). The type complexity of computing whether the subtyping relation holds between two types is O(h).

The algorithm traverses the program top-down multiple times to infer the type relation, i.e., minimum expression types, and then traverses the program bottom-up once to infer the htype, i.e., maximum command types. Facts of the type relation for variables can be inferred by use of the LETID rules. New types for variables can be inferred by use of the ASSIGN ID rule. This can cause facts of the type relation for other variables to be inferred. At any point in the evaluation at most one fact of the type relation is kept for a program node, and that is the one with the highest type for the program node that has been inferred so far. The type of each program node can be raised at most h times. Thus worst case time complexity for each of the extended Datalog rules for type inference is equal the program size multiplied by the height of the lattice of security types and the time to compute Join and Meet, i.e., O(p×h×log s).

The additional algorithms for inferring principal variables and keeping security level history of variables have time complexity O(p×pVars×s), where pVars is the maximum number of principal variables for an expression, and s is the size of the lattice of security types. The algorithm for error reporting has worst-case time complexity O(#error×pVars^2×s), where #error is the number of information flow errors inferred and #error≤(#assign + #arralloc + #arassign).

3.4.2 Space complexity

To analyze space complexity we consider the space needed beyond the space taken by the given program, lattice, and location and array type assignment relations locenv and arrenv. The total such space is the sum of the space needed for the result sets Otype, Oerror, and O'htype, the worksets Wtype, Werror, and Whtype, and the space needed for all auxiliary maps. Worksets take the same space asymptotically as the result sets, so we will not consider them separately here. We refer to the space taken by the result sets Otype, Oerror, and O'htype as *output space*. We refer to the space taken by all auxiliary maps as *auxiliary space*. Here we show the space complexity computation for the ARITH rule. Space complexity is computed in the same way for all remaining rules.

Analysis of output space. A result set is created for each relation that occurs in the conclusion of a rule, i.e., each relation for which new facts may be inferred, namely, type, htype, and error. The space taken by the result set for a relation is clearly bounded by the product of the sizes of its arguments' domains.

For the type relation, the argument domains are program constructs and types in the security type lattice, so the output space for it is O(p×s). The argument domains of the htype relation are commands in the program and types in the security type lattice; the output space for it is O(#cmd×s). The argument of the error relation is a command, so the output space for this relation is O(#cmd), which is bounded above by O(p). We conclude that the total asymptotic output space complexity is O(p×s).

To analyze the output space for informative error reporting, we consider the relations `prinVar`, `hist`, and `errReport`. The asymptotic output space for error reporting is dominated by the space for the `errReport` relation, which has four arguments - a command, two variables, and a type. The output space is $O(\#cmd \times \#id^2 \times s)$.

Analysis of auxiliary space. Auxiliary maps are created only for rules that have two hypotheses. The space needed by an auxiliary map depends on the operations that need to be performed on it. We distinguish between two kinds of auxiliary maps — ones that need to support the image set operation only, and ones that need to support the image operation and membership test. For maps in the first category, space is needed for the nested arrays for the anchors and the nested linked lists for the non-anchors. For maps in the second category, space is needed for the nested arrays for all components and the nested linked lists for the nonanchors, however the latter does not affect the asymptotic space needed.

Here we show the auxiliary space computation for the rules (1) for arithmetic expressions. The others are very similar. Four auxiliary maps are needed for these rules: in terms of the arguments of these rules, `arith2_13` maps `[e1]` to `[e e2]`, `type1_2` maps `[e1]` to `[t1]`, and `arithType3_124` maps `[e2]` to `[e e1 t1]`. The size of `type1_2` is bounded by the size of the `type` relation, which is $O(p \times s)$, as discussed above. The other two auxiliary maps need to support the image set operation only. The space required for `arith2_13` is the size of the array for the anchor plus the sum of the sizes of all the linked lists for the non-anchor components. The former is $O(\#expr)$. The latter is bounded by the size of the `arith` relation. So, the space needed for `arith2_13` is $O(\#expr + \#arith)$, which is $O(\#expr)$. The space need for `arithType3_124` is calculated in the same way and is also $O(\#expr)$. Thus the total space for these auxiliary maps is $O(p \times s)$.

Auxiliary space for type inference for information flow is summarized in Figure 3.6. The first column gives the names of all auxiliary maps used. The second column shows their anchors and nonanchors. The third column lists the rules that share each map. The fourth column shows the space needed for the map.

The total auxiliary space needed is $O(\#array \times s^2 + p \times s)$. The total space complexity of type inference for secure information flow is thus $O(\#array \times s^2 + p \times s)$.

3.5 Experimental Results

To experimentally confirm our time complexity calculations, we generated an implementation of our algorithm in Python. The generated implementation consists of 900 lines of Python code. We analyzed programs of varying size, to determine how the running time of the algorithm scales with program size. For each program, we report the CPU time for the analysis, using Python 2.3.5 on a 500MHz Sun Blade 100 with 256 Megabytes of RAM, running SunOS 5.8. Reported times are averaged over 10 trials. We use two security types in these experiments, *low* and *high*. All timing data shown is for experiments with all

auxiliary map	[anchor nonanchor]	rules that need it	space
arrlen2_1	[[a] [e]]	4	O(#expr + #array)
arraccess2_13	[[e] [a e1]]	5	O(#expr + #array)
arraccessArrenv2_12	[[a] [t1 t2]]	5	O(#array + s)
arith2_13	[[e1] [e e2]]	6	O(#expr)
type1_2	[[e1] [t1]]	5-21	O(p×s)
arithType2_13	[[e2] [e t1]]	4	O(#expr + s)
assign2_13	[[x] [c e]]	7,8,9,20,21	O(p)
assignId2_13	[[e][c x]]	7,20	O(p)
assignIdType3_124	[[x] [c t1]]	7	O(p + s)
assignIdTypeType2_134	[[x] [c t1 t2]]	7	O(p + s)
assignLoc2_13	[[e] [c x]]	8,9,21	O(p)
assignLocType1_234	[[c] [x e t1]]	9	(O(p + s)
assignLocType3_124	[[l] [c e t1]]	8	O(p + s)
arralloc2_13	[[a] [c e1]]	10,22	O(p)
arrenv1_23	[[a] [t1 t2]]	10,11,12,22,23	O(#array×s^2)
arrallocArrenv4_123	[[a] [c e1 t1]]	10,22	O(p)
arrassign2_134	[[a] [c e1 e2]]	11,12,23	O(p)
arrassignArrenv2_134	[[e1] [c a e2 t1]]	11,12	O(p+s)
sequence1_23	[[c] [c1 c2]]	13,14,24	O(#cmd)
if2_134	[[e] [c c1 c2]]	15,16,25	O(p)
ifType1_234	[[c] [c1 c2 t1]]	15,16	O(p + s)
while2_13	[[e] [c c1]]	17,26	O(#cmd + #expr)
whileType1_234	[[c] [e c1 t1]]	17	O(#expr + #cmd + s)
letid3_124	[[e] [c x c1]]	18	O(#expr + #cmd)
letid1_234	[[c] [x e c1]]	19	O(p)
letid4_123	[[c1] [c x e]]	27	O(p)
htype1_2	[[c1] [t1]]	24,25,26,27	O(p×s)
sequenceHtype2_13	[[c2] [c c1 t1]]	24	O(#cmd + s)
if3_124	[[c] [e c1 c2]]	25	O(#cmd + #expr)
ifHtype3_124	[[c2] [c e c1 t1]]	25	O(#cmd + #expr + s)
while1_23	[[c] [e c1]]	26	O(#cmd + #expr)
total auxiliary space			O(#array×s^2+ p×s)

Figure 3.6: Auxiliary space used for type inference.

global variables having the *low* security type.

Since the type system supports a relatively small number of operations, finding programs for real applications it could analyze proved a challenge. We overcame this by analyzing programs generated from SCR specifications [43], including specifications for real applications. SCR specifications use a tabular notation, built on top of a state machine model, to specify the behavior of a system. We modified OSCR, a code generator for SCR [74], to generate programs using only the operations the type system supports. This involved adding an outer loop that waits for events, rather than using function calls to notify the generated code of events. We also extended OSCR to output the abstract syntax tree of the generated code as Datalog facts. This allows us to analyze realistic systems.

Table 3.7 gives the results for inferring minimum types of expressions and maximum types of commands. The second column gives the program size, expressed as the number of nodes in the abstract syntax tree. The third column gives the CPU time required to infer minimum expression types for each program. The fourth column gives the time per fact required to infer minimum types of expressions, which should remain constant. Indeed, it is nearly the same for all programs except the smallest and largest; we suppose the variation

Description of Program	Number of Program Nodes	Time to Infer Expression Types		Time to Infer Command Types(ms)
		Total (ms)	Per node (μs)	
Thermostat	89	6	67	10
ralphSIS	150	10	77	14
Safety Injection System	159	12	75	16
Shutdown Control Logic for a Nuclear Power Plant	411	32	78	44
Cruise Control System	465	36	77	50
FDIR	519	38	73	58
Contol Panel	3471	370	107	904

Figure 3.7: Time to infer minimum types for expressions and maximum types for commands.

is due to the memory hierarchy. The final column gives the CPU time taken to infer the maximum types of commands. The results show that CPU for inferring minimum time for expressions is linear in the number of program nodes. The maximum types of commands were inferred given the inferred minimum types of expressions. Since the input is the set of types of all program nodes, but maximum types are inferred just for commands, the time complexity was linear in a combination of the number of program nodes and the number of commands in the program.

3.6 Related Work and Conclusion

A large amount of research has been done on information flow analysis since Denning's pioneering work [29, 30]. A survey of language-based information flow security appears in [75]. Various analysis frameworks have been used, including abstract interpretation, e.g., [12, 8, 38, 39], and type systems, e.g., [88, 89, 61, 66, 78, 84, 28].

Type-based approaches have been studied extensively, because types are inherently compositional, provide good documentation as well as correctness guarantees, and seem more familiar to programmers (who are familiar with standard type systems). As the survey [75] shows, there are many information-flow type systems. We focus here on the ones for which type inference algorithms have been developed. The difficulty of type inference depends on many factors, notably whether polymorphism is allowed, and whether the security levels, which in general form a partial order, are assumed to form a lattice. Volpano, Irvine and Smith present an information-flow type system for a simple imperative programming language with local variables, and prove that the type system is sound [88]. The language does not have procedures, so there is no polymorphism. A decade later, Deng and Smith give a type inference algorithm for this language extended with arrays but without local variables and assuming the security levels form a lattice [28]. Their algorithm uses

explicit iteration to compute a least fixed point. The worst-case time complexity of their algorithm is $O(n^2h)$, where n is the program size, and h is the height of the lattice. Their time complexity analysis assumes that joins can be computed in constant time. At a high level, their algorithm and our algorithm are very similar. The main difference is that, by expressing the algorithm using rules and applying a systematic implementation method, we obtain a more efficient implementation, whose worst-case time complexity is linear, rather than quadratic, in the program size.

Type inference algorithms for languages with polymorphism typically have two main aspects: generating sets of constraints during traversals of the program's abstract syntax tree, and solving (specifically, checking satisfiability of and simplifying) those sets of constraints. Basically, the constraints are inequalities involving meta-variables that range over security levels.

Volpano and Smith give a type inference algorithm for the language in [88] extended with polymorphic procedures [89]. Their constraint generation algorithm handles polymorphism in a simple but impractical (expensive) way: a procedure body is re-analyzed in each calling context. As a result, the worst-case time complexity of their algorithm is exponential in the depth of the procedure call graph. Checking satisfiability of the constraints is NP-complete in general, but it can be done more efficiently if the security levels form a disjoint union of lattices. Recent work on type-based information-flow security considers many additional features found in modern programming languages, such as dynamically allocated mutable objects, subclassing, method overriding, type casts, dynamic type tests, and exceptions [61, 66, 78, 84]. Myers' work on JFlow, an extension of Java with type-based information-flow control, considers only intra-procedural type inference [61], so users must annotate methods and fields. Pottier and Simonet consider type inference for an extension of ML with information-flow types [66, 78]. They use an existing technique [83] to generate constraints and focus on solving the constraints. They give an algorithm for solving the constraints and point out that advanced techniques will be needed to optimize it.

Sun, Banerjee, and Naumann consider type inference for an object-oriented language in which polymorphic types may be given for libraries but (to make type inference more tractable) mutually recursive classes and methods in the analyzed part of the program are treated monomorphically [84]. The inference is modular, done via a library of types that have already been computed. The types for each module are computed incrementally, and new results are merged into the library. The time complexity of the inference algorithm is $O(\mathtt{mn(s + t)}^3)$, where `m` is the number of methods in the unit, `n` is the size of the unit, `s` and `t` are the number of distinct variables in class level and method level, respectively. The authors use several functions to retrieve the types of fields and methods, in the library and in the current unit. These functions can be implemented as Datalog facts, and the type inference rules — as Datalog rules. It is our plan to extend the language our current algorithm handles to an object-oriented language and generate an efficient algorithm for information flow type inference for a language with more features.

In short, while several information-flow analysis algorithms exist, they have been developed manually under different assumptions and for different language features and different definitions of information flow, so it is difficult to compare them. Furthermore, relatively little is known about the worst-case or typical time complexity of these algorithms.

In summary, this chapter presents an approach to systematically deriving efficient algorithms for type inference for secure information flow types. We applied the approach to a classic information flow type system [88] and obtained an efficient type inference algorithm and a precise characterization of its time complexity. We plan to apply the approach to information flow type systems for richer programming languages, compare the time complexity and precision of the resulting algorithms, and evaluate their performance on real applications.

Chapter 4

Efficient Trust Management Policy Analysis

4.1 Introduction

Trust management is a unified approach to specifying and enforcing security policies in distributed systems [40, 17] and has become increasingly important as systems become increasingly interconnected. At the same time, logic-based languages and frameworks have been used increasingly for expressing security and trust management policies, e.g., [48, 53]. For analysis and enforcement of security and trust management policies, a method for generating efficient algorithms and implementations from policies specified using logic rules is highly desired.

This chapter describes a systematic method for deriving efficient algorithms and precise time complexities from extended Datalog rules as it is applied to the analysis of trust management policies specified in SPKI/SDSI, a well-known trust management framework designed to facilitate the development of secure and scalable distributed computing systems. SPKI/SDSI [34] is based on public keys and incorporates *Simple Public Key Infrastructure* (SPKI) and *Simple Distributed Security Infrastructure* (SDSI). It provides fine-grained access control using local name spaces and a security policy model.

The SPKI/SDSI framework facilitates granting and delegating authorizations, as well as naming. It uses name certificates to define names in principals' local name spaces as keys or other names, and uses authorization certificates to grant authorizations and to delegate the ability to grant authorizations. A principal is authorized to access a resource by an authorization certificate or by a chain of certificates involving naming and delegation. Designing efficient algorithms for inferring authorizations and answering related queries is

essential for enforcing SPKI/SDSI policies.

We express policy analysis problems using extended Datalog, extended with list constructors and external functions. We represent certificates as facts, and describe rules and queries for computing the reduction closure, inferring authorizations, and solving other policy analysis problems for SPKI/SDSI. These other analysis problems include ones about the current state of the policy, as well as ones about changes in the state that would be caused by possible changes in the policy, such as expiration or addition of a set of certificates.

We describe our method for systematically generating specialized algorithms and data structures, together with precise time complexity formulas, from extended Datalog rules as it is applied to computing reduction closure and inferring all authorizations. The generated algorithms employ an incremental approach that considers one certificate or intermediate analysis fact at a time, and use a combination of linked and indexed data structures to represent different certificates and intermediate values. The running time is optimal for the respective rules, in the sense that each combination of instantiations of hypotheses is considered once in $O(1)$ time.

We then describe other policy analysis problems as additional rules and queries, and use a method to systematically push given inputs for the analyses from queries into hypotheses of rules, yielding specialized and simplified rules for the given queries. This is similar to pushing demands by queries in magic set transformations [15], but instead of yielding more complicated rules with magic predicates, we obtain simplified, specialized rules that are much easier for generating efficient implementations and precise complexities.

Contrasting various previous works, our rules and algorithms for policy analysis support all aspects defined in the specification for SPKI/SDSI, including any number of resources and accesses, names consisting of any number of identifiers, and validity intervals. We also have a prototype implementation, and experimental results confirm our precise complexity analysis.

A significant amount of work has been done on algorithms for SPKI/SDSI policy enforcement and analysis [72, 2, 34, 52, 25, 55, 41, 6, 42, 54, 49, 32]. Our approach of expressing policy analysis problems as extended Datalog rules is much simpler than previous techniques for analysis of SPKI/SDSI policies. Our method also derives better, more precise time complexities than before in addition to generating complete algorithms and data structures. The method is general, with many applications beyond policy analysis. It extends our previous method for Datalog [57] to handle list constructors, external functions, and queries.

The rest of this chapter is organized as follows. Section 2 reviews the SPKI/SDSI trust management framework and defines the problems of computing the reduction closure and inferring authorizations. Section 3 expresses computing reduction closure and inferring authorizations in rules, describes generation of efficient algorithms and data structures from the rules, and precise time complexity analysis. Section 4 presents specialized policy analysis problems expressed in rules, along with precise time complexity analysis for them. Section 5 discusses related work and concludes.

4.2 SPKI/SDSI

In SPKI/SDSI systems, *principals* are the users and are identified by public keys, which we will simply refer to as *keys*. *Identifiers* are words over a standard alphabet and are used to refer to principals and resources. A *name* is a key followed by a sequence of identifiers.

SPKI/SDSI certificates are *name certificates* and *authorization certificates*. A *name certificate* defines a local name in its issuer's local name space. A name certificate is the 4-tuple (K, I, S, V), where K is the public key of the issuer of the certificate; I is an identifier from the local name space of the issuer; S is the name or key that the local name KI stands for; V is the validity time interval for the certificate and is of the form $[t1, t2]$, where $t1$ and $t2$ are absolute time constants. The 4-tuple defines the name KI to stand for S during validity interval V. A name certificate can only be issued by the principal to whom the name being defined is local. We refer to certificates in which S is a name as *name-name* certificates, and to ones in which S is a key as *name-key* certificates. A name can correspond to a set of keys.

Principals use *authorization certificates* to grant permissions for accessing resources to other principals. An authorization certificate is a 5-tuple (K, S, D, P, V), where K is the public key of the certificate issuer — the principal granting authorization; S is the subject of the certificate — the key or name that is being granted authorization; D is a boolean delegation bit indicating if the subject is granted the right to delegate the permissions granted by the certificate to others; P is the set of permissions, i.e., operation-resource pairs, being granted; V is a validity interval as for name certificates.

A principal Pr has permission for an operation on a resource if there is a valid authorization certificate (R, Pr, D, P, V), where P contains the operation-resource pair, and R is the owner of the resource involved in permission P, or if such a certificate can be inferred, i.e., there is a chain of certificates that authorizes the access. Certificates are composed in chains by use of the following composition rules.

- Two name certificates, such as $(k1, id1, k2\ id2\ ids, v1)$ and $(k2, id2, s, v2)$, can be composed to infer $(k1, id1, s\ ids, v3)$, where $v3$ is the intersection of validity intervals $v1$ and $v2$.

- Two authorization certificates, $(k1, k2, d1, p1, v1)$, where $d1 = TRUE$, and $(k2, s, d2, p2, v2)$ can be composed to infer the certificate $(k1, s, d2, p3, v3)$, where $p3$ is the intersection of authorization sets $p1$ and $p2$, $v3$ is the intersection of validity intervals $v1$ and $v2$.

- An authorization certificate $(k1, k2\ \ id\ \ ids, d, p, v1)$ and a name certificate $(k2, id, s, v2)$, can be composed to infer $(k1, s\ ids, d, p, v3)$, where $v3$ is the intersection of validity intervals $v1$ and $v2$.

The *closure* of a set of certificates contains all given certificates and all certificates that can be inferred using the above rules. However, the closure of a set of certificates may

be infinite. The *reduction closure* of a set of authorization and name certificates contains all given certificates and all certificates that can be inferred using chains in which every certificate after the first one has a key as its subject. For each name occurring in a set of certificates, the reduction closure contains all name-key certificates that define the name as a key, as in the full closure of the set of certificates. Also, for a given key, the reduction closure contains all authorization certificates in which the key is a subject, that occur in the full closure. Thus, the reduction closure can be used to find all keys that a name stands for, as well as to find all permissions that a key has.

4.3 Computing Reduction Closure Efficiently

This section expresses reduction closure and authorization inference using extended Datalog rules, and describes the generation of specialized algorithms and data structures from the rules, we also analyze precisely the time complexities, expressing the complexities in terms of characterizations of the given set of certificates.

4.3.1 Expressing reduction closure in rules

We use the following relations to denote certificates:

- `nameCert(k,id,s,v)`: a given name certificate.
- `authCert(k,s,d,p,v)`: a given authorization certificate.
- `name(k,id,s,v)`: an inferred name certificate.
- `auth(k,s,d,p,v)`: an inferred authorization certificate.

We use three external functions. The symbol | separates the head from the tail in a sequence of identifiers and `NIL` denotes the empty list. The functions `PInt(p1,p2)` and `VInt(v1,v2)` return the intersections of two sets of permissions, and two validity intervals, respectively.

The rules for composing chains of certificates can readily be written as the extended Datalog rules shown in Figure 4.1.

4.3.2 Generating efficient algorithms and data structures

We transform the extended Datalog rules into an efficient implementation using the method in [57] for Datalog rules. A small extension is needed to handle the external functions `|`, `VInt`, and `PInt`.

1. `nameCert(k,id,s,v)`→`name(k,id,s,v).`
2. `authCert(k,s,d,p,v)`→`auth(k,s,d,p,v).`
3. `name(k1,id1,k2|(id2|ids),v1),name(k2,id2,k3|NIL,v2)`
   →`name(k1,id1,k3|ids,VInt(v1,v2)).`
4. `auth(k1,k2|NIL,TRUE,p1,v1),auth(k2,k3|NIL,d2,p2,v2)`
   →`auth(k1,k3|NIL,d2,PInt(p1,p2),VInt(v1,v2)).`
5. `auth(k1,k2|(id|ids),d,p,v1),name(k2,id,k3|NIL,v2)`
   →`auth(k1,k3|ids,d,p,VInt(v1,v2)).`

Figure 4.1: Rules for computing the reduction closure.

Fixed-point specification and `while`-loop. The input to the algorithm is the given set of certificates represented by a set `certs` of facts. We define `rcerts` to be the set of facts in `certs` represented as tuples as described above.

```
rcerts = {[authCert k s d p v]:
            authCert(k,s,d,p,v) in certs}
       ∪{[nameCert k id s v] :
            nameCert(k,id,s,v) in certs}.
```

Given any set `R` of facts, and an extended Datalog rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)` be the set of all facts that can be inferred by that rule given the facts in `R`. For our rules we have:

```
1name={[name k id s v] :
  [nameCert k id s v] in R},
2auth={[auth k s d p v] :
  [authCert k s d p v] in R},
3name={[name k1 id1 k3|ids VInt(v1,v2)] :
  [name k1 id1 k2|(id2|ids) v1] in R and
  [name k2 id2 k3|NIL v2] in R},
4auth={[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] :
  [auth k1 k2|NIL TRUE p1 v1] in R and
  [auth k2 k3|NIL d2 p2 v2] in R},
5auth={[auth k1 k3|ids d p VInt(v1,v2)] :
  [auth k1 k2|(id|ids) d p v1] in R and
  [name k2 id k3|NIL v2] in R}.
```

The meaning of the given set of certificates and the extended Datalog rules for reduction closure is:

$$\texttt{LFP}(\{\},\texttt{F}),\ \text{where}\ \texttt{F(R)}=\texttt{R}\cup\texttt{rcerts}\cup \texttt{1name(R)}\cup\texttt{2auth(R)}\cup\texttt{3name(R)}\cup\texttt{4auth(R)}\cup\texttt{5auth(R)} \tag{4.1}$$

This least-fixed point specification of computing the reduction closure is transformed into the following `while` loop:

```
R := {};
while exists x in F(R) - R:
  R with := x;
```
(4.2)

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop. After the execution of this loop `R` contains all facts that are given or can be inferred by the rules. `R` is referred to as the *resultset*.

Incremental computation Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in type inference are all sets of facts inferred by each rule and a workset `W`. We use fresh variables to hold each of their respective values and maintain an invariant for each of these sets, in addition to one for the workset.

```
I1name = 1name(R), I2auth = 2auth(R),
I3name = 3name(R), I4auth = 4auth(R),
I5auth = 5auth(R), W = F(R) - R.
```

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I3name`. `I3name` is the value of the set formed by joining two name certificates. `I3name` can be initialized to $\{\}$ with the initialization `R := {}`. To update `I3name` incrementally with update `R with:= x`, if `x` is of the form `[name k1 id1 k2|(id2|ids) v1]`, we consider matching tuples of the form `[name k2 id2 k3|NIL v2]` and add all corresponding new tuples `[name k1 id1 k3|ids VInt(v1,v2)]` to `I3name`. To form the tuples to be added, we need to efficiently find the appropriate values of variables that occur in `[name k2 id2 k3|NIL v2]` tuples, but not in `[name k1 id1 k2|(id2|ids) v1]`, i.e., the values of `k3` and `v2`, so we maintain an auxiliary map, `I3name1`, shown below, that maps `[k2 id2]` to `[k3 v2]`. Symmetrically, if `x` is a tuple of the form `[name k2 id2 k3|NIL v2]`, we need to consider every matching tuple of the form `[name k1 id1 k2|(id2|ids) v1]` and add the corresponding tuple of the form `[name k1 id1 k3|ids VInt(v1,v2)]` to `I3name`, so we maintain the auxiliary map `I3name2` below.

```
I3name1 = {[[k2 id2] [k3 v2]] :
      [name k2 id2 k3|NIL v2] in R},
I3name2 = {[[k2 id2] [k1 id1 ids v1]] :
      [name k1 id1 k2|(id2|ids) v1] in R}.
```

The first set of components in an auxiliary map is referred to as the *anchor* and the second set of elements as the *nonanchor*.

Thus, the algorithm can directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, and it considers each combination only once. Auxiliary maps are maintained similarly for all maintained invariants, `I4auth` and `I5auth` here, that are formed by joining two relations.

All variables holding the values of expensive computations listed above, and auxiliary maps, are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. We show the update for the addition of a fact of relation `name` only for `I3name` and auxiliary map `I3name2`. Other updates are processed in the same way.

```
case of x of [name k1 id1 k2|(id2|ids) v1]:
 I3name ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]
      :  [k3 v2] in I3name1{[k2 id2]}};
 W ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]                    (4.3)
      :  [k3 v2] in I3name1{[k2 id2]}
      | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
 I3name2 ∪:= {[[k2 id2] [k1 id1 ids v1]]};
```

Adding these initializations and updates, and other similar ones for the other cases, and replacing `F(R) - R` with `W` in (4.2), we obtain the following complete code:

```
initialization; R:={};
while exists x in W:
   update using (4.3) and                                   (4.4)
   similar updates for the other cases;
   W less:= x; R with:= x;
```

Next, we eliminate dead code. To compute the resultset `R`, only `W` and the auxiliary maps are needed; the invariants maintained, i.e., `I3name`, `I4auth`, and `I5auth`, are dead because `F(R)-R` in the `while` loop was replaced with `W`. We eliminate them from the initialization and updates. For example, eliminating them from the updates in (4.3), we eliminate lines 2-3.

```
case of x of [name k1 id1 k2|(id2|ids) v1]:
 W ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]
      :  [k3 v2] in I3name1{[k2 id2]}
      | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
 I3name2 ∪:= {[[k2 id2] [k1 id1 ids v1]]};
```
(4.5)

We clean up the code to contain only uniform operations on set elements. This simplifies data structure design. We decompose `R` and `W` into several sets, each corresponding to one relation in the extended Datalog rules. `R` is decomposed to `RnameCert`, `RauthCert`, `Rname` and `Rauth`; `W` is decomposed to `WnameCert`, `WauthCert`, `Wname`, and `Wauth`. This decomposition lets us eliminate relation names from the first component of tuples, with appropriate changes to the rest of the code. Then, we apply the following three sets of transformations.

`(i)` Transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for`-loop that adds the elements one at a time. For example, lines 2-4 of (4.5) are transformed into:

```
for [k3 v2] in I3name1{[k2 id2]}:
  if [k1 id1 k3|ids VInt(v1,v2)] notin Rname:
    Wname with:= [k1 id1 k3|ids VInt(v1,v2)];
```
(4.6)

`(ii)` Replace tuples and tuple operations with maps and map operations. Specifically, replace all `for`-loops as follows. (4.6) is transformed into:

```
for [k2 id2] in dom(I3name1):
 for [k3 v2] in I3name{[k2 id2]}:
  if [k1 id1 k3|ids VInt(v1,v2)] notin Rname:
   Wname with:= [k1 id1 k3|ids VInt(v1,v2)];
```

We replace the `while` loop similarly. Also, we replace each $[X\ Y]$ `notin` M with Y `notin` $M\{X\}$. Each addition to a map M `with:=` $[X\ Y]$ is replaced with $M\{X\}$ `with:=`Y.

`(iii)` Test for membership before adding or deleting an element to or from a set. Specifically, we replace each statement S `with:=`X with `if` X `notin` S `:` S `with:=`X.

Note that when removing an element from a workset, the membership test is unnecessary, since the element is retrieved from the workset. Also, when adding an element to a resultset, the membership test is unnecessary, since elements are moved from the corresponding workset to the resultset one at a time, and each element is put in the workset and thus in the resultset only once.

After the above transformations, each firing of an extended Datalog rule involves a constant number of set operations. Since each set operation takes worst-case constant time

```
W := rcerts;
I3name1 := {}; I3name2 := {};
I4auth1 := {}; I4auth2 := {}; I5auth3 := {};
R := {};

while exists x in W:

  case x of [nameCert k id s v]:
    if [name k id s v] notin R:
    W with := [name k id s v];

  case x of [authCert k s d p v]:
    if [auth k s d p v] notin R:
      W with := [auth k s d p v];

  case x of [name k1 id1 k2|(id2|ids) v1]:
    W U:= {[name k1 id1 k3|ids VInt(v1,v2)] : [k3 v2] in I3name1{[k2 id2]}
                                          | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
    I3name2 with:= [[k2 id2] [k1 id1 ids v1]];

  case x of [name k2 id2 k3|NIL v2]:
    W U:= {[name k1 id1 k3|ids VInt(v1,v2)] : [k1 id1 ids v1] in I3name2{[k2 id2]}
                                           | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
    W U:= {[auth k1 k3|ids d p VInt(v1,v2)] : [k1 ids d p v1] in I5auth3{[k2 id2]}
                                           | [auth k1 k3|ids d p VInt(v1,v2)] notin R};
    I3name1 with:= [[k2 id2] [k3 v2]];

  case x of [auth k1 k2|NIL TRUE p1 v1]:
    W U:= {[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] : [k3 d2 p2 v2] in I4auth2{[k2]}
                                  | [auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] notin R};
    I4auth1 with:= [[k2] [k1 TRUE p1 v1]];

  case x of [auth k2 k3|NIL d2 p2 v2]:
    W U:= {[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] : [k1 d1 p1 v1] in I4auth1{[k2]}
                                  | [auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] notin R};
    I4auth2 with:= [[k2] [k3 d2 p2 v2]];

  case x of [auth k1 k2|(id|ids) d p v1]:
    W U:= {[auth k1 k3|ids d p VInt(v1,v2)] : [k3 v2] in I3name1{[k2 id]}
                                           | [auth k1 k3|ids d p VInt(v1,v2)] notin R};
    I5auth3 with:= [[k2 id] [k1 ids d p v1]];

  W less:= x;
  R with:= x;
```

Figure 4.2: Pseudocode for computing reduction closure.

in the generated code, as described below, each firing takes worst-case constant time. The complete pseudocode for computing reduction closure efficiently is shown in Figure 4.2.

4.3.3 Time complexity analysis

We analyze the time complexity of computing reduction closure by carefully bounding the number of facts actually used by the rules. For each rule we determine precisely the number of facts processed by it, avoiding where possible approximations that use the

product of the sizes of individual argument domains.

We first define the size parameters used in the complexity analysis. The number of facts of a relation `r` that are given or can be inferred is called `r`'s *size*. For a relation named `r`, `#r` denotes the size of `r`. We use the following size parameters about inferred certificates:

- `nameKey` — number of name certificates that have keys as subjects.
- `nameKeyPerName` — maximum number of name certificates, that have keys as subject, for one name.
- `namePerSubject` — maximum number of name certificates for one subject.
- `authD` — number of authorization certificates with a delegation bit TRUE.
- `authPerIssuer` — maximum number of authorization certificates for one issuer.
- `authPerIssuerD` — maximum number of authorization certificates with delegation bit TRUE for one issuer.
- `authPerSubject` — maximum number of authorization certificates for one subject.

In addition, we use `key` for the total number of different keys in the given certificates.

The time complexity for a set of Datalog rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule `r`, the number of firings for the rule is: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts that make the two hypotheses simultaneously true. The total time complexity is time for reading the input, plus the time for firing all rules.

The total time complexity for computing the reduction closure is time for reading the input, which is $O(\texttt{\#authCert} + \texttt{\#nameCert})$, plus the time for applying each of the rules. `VInt(v1, v2)` is computed in constant time; `PInt(p1,p2)` can be computed in time $O(\texttt{p})$, where `p` is maximum size of a permission argument in the given authorization certificates. List operations involving `|` can be performed in time $O(1)$.

Time complexity of the rules used for name-reduction closure and inferring authorizations is as follows:

1. $O(\texttt{\#nameCert})$
2. $O(\texttt{\#authCert})$
3. $O(\min(\texttt{\#name} \times \texttt{nameKeyPerName},$
   $\texttt{nameKey} \times \texttt{namePerSubject}))$
4. $O(\min(\texttt{authD} \times \texttt{authPerIssuer},$
   $\texttt{\#auth} \times \texttt{authPerIssuerD}))$
5. $O(\min(\texttt{\#auth} \times \texttt{nameKeyPerName},$
   $\texttt{nameKey} \times \texttt{authPerSubject}))$

The time complexity for the whole reduction closure is the sum of the time complexities for rules 1, 2, 3, 4, and 5. The sum for rules 1 and 2 is the number of given certificates. The sum for rules 3-5 is larger and decides the total time complexity.

To compare with previous results, suppose we eliminate the permission and validity interval arguments, or consider only certificates with the given permission and validity interval, as in [25, 49]. Then `nameKeyPerName` is the maximum number of keys a single local name reduces to, and is `key` in the worst case; and `authPerIssuerD` is the maximum number of keys authorized by one issuer with delegation bit TRUE, and is again `key` in the worst case. Thus, our precise complexity formulas for rules 3-5 is $O((\texttt{\#name} + \texttt{\#auth}) \times \texttt{key})$ in the worst case. `#name`+`#auth` is the total number of certificates inferred and, as noted in [49], is bounded by `in`$\times$`key`, where `in` is the size of the input, i.e., the sum of the sizes of the given certificates; note that the size of a certificate might not be a constant because its subject may be a key followed by a list of identifiers. Therefore, the time complexity $O(\texttt{in} \times \texttt{key}^2)$ from previous work [49] is an upper bound of our more precise complexity analysis.

4.4 Specialized Policy Analysis Problems

This section discusses how to solve specialized certificate analysis problems and analyze their algorithm complexities. The algorithms for computing reduction closure can be used to solve specialized analysis problems. However, these algorithms compute all authorizations and all name-key correspondences, given a set of certificates. This may be unnecessary, since many policy analysis problems require computing only a few authorizations or resolving only a few names. Therefore, we use specialized extended Datalog rules for the specialized analysis problems; these specialized rules can be used to generate an efficient algorithm for each analysis problem, and infer only the authorizations and resolve only the names needed for that problem. Also, the original reduction closure algorithm does not give a direct way of solving some important policy analysis problems, specifically when questions about name certificates are asked, when sets of resources or keys are given. There are algorithms for solving these problems in [49], but these require complex pushdown system structures that are not inherent to the problems' structure.

We first introduce extended Datalog rules to solve the problems and then show a way to construct specialized rules from given rules, by pushing the constants bound by the query into the rules. There are automatic ways of generating on-demand rules such as *Magic Set Transformation* (MST) [15, 68]. MST introduces demand relations corresponding to the query; and makes changes that limit the facts being inferred to ones demanded by the query. We chose to push the constants in a naïve manner despite the fact that MST may do better than our technique for some problems; mainly because MST is much more sophisticated, the order of hypotheses in the original rules may significantly change the efficiency of the transformed rules and moreover there is no reason (except the resulting

complexity) to prefer an order to the other before starting the transformation. By pushing constants into the rules, we obtain simpler rules and precise complexities.

4.4.1 Policy analysis and complexity analysis in a logic framework

We consider all the analysis problems studied in [49]. All problems are solved with respect to a given set of certificates. In the rules, we use the names "permissions" and "resources" interchangably, in our context "permission" means an access to a resource without loss of generalization. In the construction of rules, we leave the unknown to the question as the last argument, and try to remain consistent on the order of keys, permissions, etc. otherwise. The relations are named as close to the real meaning of the relation, e.g. `canAccess(K,P)` stands for the relation "a key K is authorized for permission P". For each problem, we first give a set of rules, followed by a Prolog-like query, that will return the requested result.

Figure 4.3 shows all of the rules for the problems below. In the rules, `owner(o,p)` denotes that `o` is an owner of permission `p`, and `auth` is as defined before. In the last four analysis problems, where some certificates are removed, `canAccess2` is defined in a similar way as `canAccess` in the first analysis, but uses authorizations inferred using only the remaining certificates, i.e., using an `auth2` relation computed as the reduction closure of the remaining certificates.

We introduce the notation for the auxiliary values used for the complexity analysis.

- `authPerKey` is the maximum number of authorizations that has a specific key as a subject.
- `ownersPerRes` is the maximum number of owners for a single resource.
- `l` is the maximum number of identifiers occurring in a name that is the subject of a certificate.
- `identifiers` is the number of distinct identifiers occurring in the certificates.
- `len(N)` for a name N is the length of the name N.

Authorized Access 1: Is a principal K authorized to permission P? This is determined in time $O($`ownersPerRes` $\times$ `authPerKey`$)$.
Authorized Access 2: Given a permission P and name N, is N authorized to P?
Authorized Access 3: Given a permission P, what names are authorized to access P?
These two questions are answered the same way as question 1; the preprocessing for adding the certificates for reduction closure takes linear time in the length of the name N for question 2; and for question 3 this procedure needs to assign a valid string of identifiers of at most length `l`, which would take `key` $\times$ `identifiers`$^{\texttt{l}}$, but the key is not bounded either so instead of `authPerKey` as a factor, we have `#auth`. Notice that this exponential

behaviour for the third question comes from the nature of the problem, since the set of names authorized to access P might be an infinite set. So the precise complexities for question 2 and 3 respectively are: $O(\texttt{ownersPerRes} \times \texttt{\#authPerKey} + \texttt{len(N)})$ and $O(\texttt{ownersPerRes} \times \texttt{\#auth} + \texttt{key} \times \texttt{identifiers}^{\texttt{l}})$.

Shared Access 1: Given two permissions P_1 and P_2, which principals are authorized for both? A straightforward analysis just as above shows that the complexity for the solution to this problem is $O(\texttt{key} + \texttt{\#auth} \times \texttt{ownersPerRes})$.

Shared Access 2: Given two principals K_1 and K_2 and a permission P, is both K_1 and K_2 authorized for P? This question is answered in a constant time factor of the answer to the Authorized Access 1 question, so it takes time $O(\texttt{ownersPerRes} \times \texttt{authPerKey})$.

Shared Access 3: Given two principals K_1 and K_2 and a finite set of permissions $Ps = \{P_1, ..., P_n\}$, what is the subset of Ps that K_1 and K_2 are authorized for? This question is answered using the rule in Shared Access 2, by checking for all elements in Ps, but the permission is not bound for `canAccess`, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time.

Compromisation Assessment 1 (also called Expiration Vulnerability 1): What permissions from a finite set of permissions $Ps = \{P_1, ..., P_n\}$ would a given principal K lose authorization for, if a subset C' of the original certificate set C were to be removed? This question is answered using `canAccess` without `p` being bound, checked for each element in Ps, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time (since `#auth2`, the number of authorizations inferred not using C' is less than `#auth`, we can ignore that part).

Compromisation Assessment 2 (also called Expiration Vulnerability 2): What principals would have lost authorization for a permission P if a subset C' of the original certificate set C were to be removed? This question is answered using the rule in Authorization Access 1 without binding `k`, by checking for all keys in the system, so it is answered in $O(\texttt{key} + \texttt{ownersPerRes} \times \texttt{\#auth})$ time (since `#auth2`, the number of authorizations inferred not using C' is less than `#auth`, we can ignore that part).

Universally Guarded Access 1: Must all authorizations for permission P involve a certificate signed by principal K? We answer the negation of this question for simplicity, in other words our rule gives a "no" for a "yes" instance and vice versa. This question is answered using the rule in Authorization Access 1 without binding `k`, by checking for all keys in the system, so it is answered in $O(\texttt{key} + \texttt{ownersPerRes} \times \texttt{\#auth})$ time (since `#auth2`, the number of authorizations inferred not using certificates signed by K is less than `#auth`, we can ignore that part).

Universally Guarded Access 2: Must all authorizations that grant a given principal K' a finite set of permissions $Ps = \{P_1, ..., P_n\}$ involve a certificate signed by K? Again we answer the negation of this question for simplicity. This question is answered using the rule in Authorization Access 1 without binding `p`, by checking for all elements in Ps, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time (since `#auth2`, the number of authorizations inferred not using certificates signed by K is less than `#auth`, we can ignore that part).

4.4.2 Constructing specialized rules

We demonstrate how to push constants to create specialized rules on one of the analysis problems. Consider the rule set and the query for the problem Compromisation Assessment 2:

```
canAccess(k,p,t), ¬canAccess2(k,p,t)
    →compromisedPrinciples(p,t,k)
```
Query : `compromisedPrinciples(P,T,k).`

Now since the permission P and time T is given when the question is asked, we push them inside the relations on the right hand side, yielding:

```
canAccess(k,P,T), ¬canAccess2(k,P,T)
    →compromisedPrinciples(P,T,k)
```

Now it is easy to observe that the conclusion expresses the constants unnecessarily, since the hypotheses are already aware of the values of them. So we can rewrite :

```
canAccess(k,P,T), ¬canAccess2(k,P,T)
    →compromisedPrinciples_PT(k)
```

This new rule is the `compromisedPrinciples` rule specialized to constants P and T; it returns precisely what we are looking for, the resulting keys. Notice that this push-and-specialize method can be applied iteratively in general, and it is particularly simple in this case since there is no recursion. In other words, in this example `canAccess(k,P,T)` can be rewritten as `canAccess_PT(k)` by pushing the constants into hypotheses properly.

4.5 Experimental Results

To experimentally confirm our time complexity calculations, we generated an implementation of our algorithm for computing reduction closure in Python. The generated implementation consists of 180 lines of Python code. We analyzed sets of certificates of varying sizes, to determine how the running times of the algorithms scale with the number of given certificates. For each certificate set, we report the CPU time for the analysis, using Python 2.3.5 on a 1.73 GHz Pentium M processor, with 366 MHz 448 MB RAM, running Windows XP. Reported times are averaged over 10 trials.

For the experiments we first infer all name facts using rules 1 and 3, and then infer the authorizations by rules 2,4 and 5. This does not affect the resulting facts and was just done for the purpose of having separate experiments for the two parts of the algorithm, so that the effect of changing certain parameters can be seen. Also, the data was generated in such a way that the number of given and inferred certificates are of the same order.

Authorized Access 1:
`owner(o,p), auth(o,k|NIL,d,ps,v), p in ps, t in v` $\rightarrow$ `canAccess(k,p,t).`
Query: `canAccess(K,P,T)`.
Authorized Access 2:
Suppose the asked given name is $\mathtt{N} = \mathtt{K}\ \mathtt{I}_1\ \mathtt{I}_2\ ..\ \mathtt{I}_n$, add new name certificates:
$(\mathtt{K}, \mathtt{I}_1, \mathtt{K}_1|\mathtt{NIL}, \mathtt{V}), (\mathtt{K}_1, \mathtt{I}_2, \mathtt{K}_2|\mathtt{NIL}, \mathtt{V}), ...(\mathtt{K}_{n-1}, \mathtt{I}_n, \mathtt{K}_n|\mathtt{NIL}, \mathtt{V})$, where each $\mathtt{K}_i$ is a fresh key, `V` is a validity interval satisfied at the current time `T`.
Query: $\mathtt{canAccess(K_n,P,T)}$.
Authorized Access 3:
For all keys and identifiers, construct all possible names up to `l` identifiers, and add new name certificates corresponding to these names as shown above.
Query: `canAccess(k,P,T)`.

Shared Access 1:
`canAccess(k,p1,t), canAccess(k,p2,t)` $\rightarrow$ `sharingPrinciple(p1,p2,t,k).`
Query : `sharingPrinciple(P1,P2,T,k).`
Shared Access 2:
`canAccess(k1,p,t), canAccess(k2,p,t)` $\rightarrow$ `sharingResource(k1,k2,p,t).`
Query : `sharingResource(K1,K2,P,T).`
Shared Access 3:
`p in ps, sharingResource(k1,k2,p,t)` $\rightarrow$ `sharingResources(k1,k2,ps,t,p).`
Query : `sharingResources(K1,K2,{P1,P2,...,Pn},T,p).`

Compromisation Assessment 1:
`p in ps, canAccess(k,p,t),` $\neg$`canAccess2(k,p,t)`
$\rightarrow$ `compromisedResource(k,ps,t,p).`
Query : `compromisedResource(K,{P1,P2,..,Pn},T,p).`
Compromisation Assessment 2:
`canAccess(k,p,t),` $\neg$`canAccess2(k,p,t)` $\rightarrow$ `compromisedPrinciple(p,t,k).`
Query : `compromisedPrinciple(P,T,k).`

Universally Guarded Access 1:
`canAccess(k1,p,t), canAccess2(k1,p,t)` $\rightarrow$ `needNotInvolve(p,t).`
Query : `needNotInvolve(P,T).`
Universally Guarded Access 2:
`p in ps, canAccess(k,p,t), canAccess2(k,p,t)`
$\rightarrow$ `needNotInvolveMultiple(k,ps,t).`
Query : `needNotInvolveMultiple(K,{P1,P2,..,Pn},T).`

Figure 4.3: Rules and queries for solving policy analysis problems.

Figure 4.4 shows the running times for inferring all name certificates. Two series of sets of certificates were used. In both series the number of certificates increases, however in the first one `nameKeyPerName` remains constant. In the second test series `nameKeyPerName` increases as the number of given certificates increase, and both of these parameters increase at the same rate. The results show that CPU time for inferring all name certificates is linear in the number of given name certificates, if `nameKeyPerName`

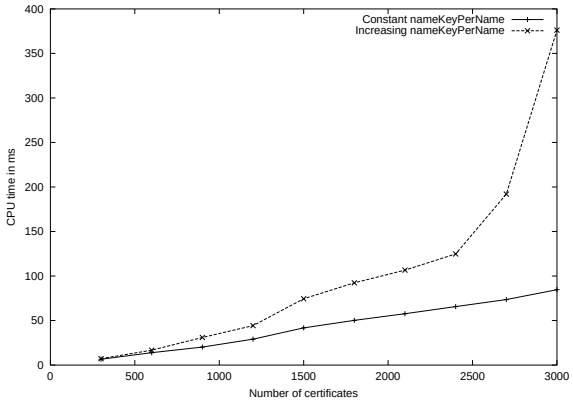


Figure 4.4: Time to infer all name certificates only.

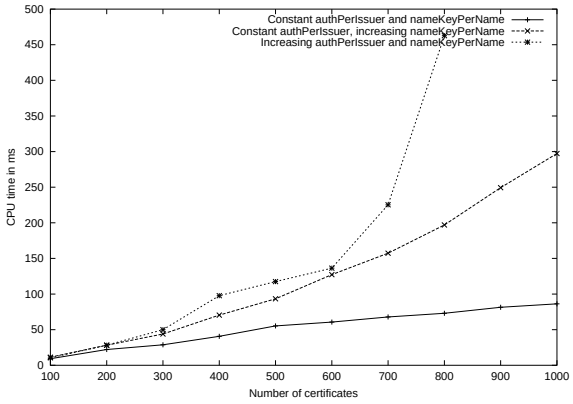


Figure 4.5: Time to infer authorization certificates given all name certificates.

is a constant. Figure 4.5 shows the running times for inferring all authorization certificates, once name certificates have been inferred. Three series of sets of certificates were used. In all three series the number of given certificates increases, however in the first one `authPerIssuer` and `nameKeyPerName` remain constant. In the second test series only `authPerIssuer` remains constant, while `nameKeyPerName` increases as the number of given certificates increases, and both of these parameters increase at the same rate. In the third series both `authPerIssuer` and `nameKeyPerName` increase along with the number of given certificates and at the same rate as the number of given certificates. The results show that CPU time for inferring all authorization certificates is linear in the number of given certificates, if `authPerIssuer` and `nameKeyPerName` are kept constant; CPU time grows faster if only `authPerIssuer` remains constant. These experimental results confirm our time complexity analysis results. In all experiments the search space increases along with the number of given certificates, so that the test results are not influenced by a disproportionately small search space. The data was generated so that the ratio of the number of keys to the number of given certificates remains the same.

4.6 Related Work and Conclusion

Surveys of trust management are presented in [40, 17, 53]. Li et al [56] define security analysis problems for trust management systems and analyze their complexity.

SDSI was proposed by Rivest and Lampson [72], as a public-key infrastructure that uses linked local names. SPKI was developed concurrently; it emphasized delegation of authorizations. These two infrastructures were merged to create SPKI/SDSI, described in RFC 2693 [34].

The reduction closure that our algorithm computes includes all the certificates inferred by previous algorithms [25, 49]. Clarke et al. [25] analyze the time complexity of their algorithm to be $O(\mathtt{n}^3 \times \mathtt{l})$, where $\mathtt{n}$ is the number of given certificates, and $\mathtt{l}$ is the length of the longest subject in any given certificate. Jha and Reps [49] give a more precise bound of $O(\mathtt{in} \times \mathtt{key}^2)$, where $\mathtt{in}$ is the size of the input and is bound by $\mathtt{n} \times \mathtt{l}$; it is more precise because $\mathtt{key}$ is bound by $O(\mathtt{n})$. Our complexity analysis is even more precise because one of their $\mathtt{key}$ factors is $\mathtt{nameKeyPerName}$ in our complexity formula, which corresponds to the maximum number of keys a single local name reduces to; while this could be $\mathtt{key}$ in the worst case, it is much smaller on average in practice and is close to a constant in large systems. Thus, our complexity analysis is more precise and informative than using only worst-case sizes.

A different algorithm for certificate chain discovery is presented by Li et al [55]. The algorithm as described does not accommodate names containing more than one identifier, but it could be altered to do so. It combines forward and backward search in a graph representation of credentials, and has a time complexity equal to that of the algorithm in [25]. Halpern and Meyden [42, 41] define a semantics for SPKI and SDSI, that facilitates reasoning about SPKI's and SDSI's design. A first-order logic semantics for SPKI/SDSI is presented in [54] and is used to analyze the design of SPKI/SDSI.

Policy analysis for SPKI/SDSI has also been studied. Jha and Reps [49] establish a connection between SPKI/SDSI and pushdown systems, and use existing algorithms for model checking pushdown systems to solve analysis problems for SPKI/SDSI. A similar approach is used in [32], but in addition, propertied of an SPKI/SDSI policy are expressed using a first order temporal logic.

What distinguishes our work [46] is that first we use a novel implementation strategy for reduction closure and inferring authorization that combines an intuitive definition of certificate composition in rules and a systematic method for deriving efficient algorithms and data structures from the rules [57]. The time complexity is calculated directly from the rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules. We also solve known policy analysis problems in a logic framework, and show a straightforward way of constructing specialized rules from our proposed solutions, which allows for easy bottom-up computation of results. Furthermore, we present precise time complexities for our proposed solutions. We achieve more precise worst-case time complexity guarantees that those in previous work. Moreover, our algorithms for authorization and other analysis support any

number of resources and types of access, as well as validity intervals and delegation; all aspects defined in the SPKI/SDSI specification.

This method of generating efficient implementations from rules has also been applied to problems beyond the area of policy analysis. In the model checking area, the method is used to derive an efficient algorithm with improved complexity analysis for linear temporal logic model checking of pushdown systems [44]. This model checking framework can express and check many practical properties of programs, including many dataflow properties and general correctness and security properties. For secure information flow analysis, the method was used to develop the first linear-time algorithm for inferring information flow types of programs for a formal type system [45]. The algorithm is also extended with informative error reporting to facilitate error detection and corrections.

Chapter 5

Answering Rule-Based Queries Efficiently

5.1 Introduction

Much work has been directed towards finding efficient ways to evaluate logic rules and queries. This is an important research problem since many practical problems, which may be difficult to understand and implement otherwise, can conveniently and intuitively be expressed in rules. Among these are deductive database queries, verification and model checking problems, and problems in program analysis and security policy frameworks.

Logic rules allow a programmer to conveniently and intuitively express the logic underlying an application, without having to consider implementation details. Queries are used to select facts of interest from facts that can be inferred, given a set of rules and facts. A query is a question about the set of rules and facts, asking about facts of a specific relation that can be inferred, possibly with certain constants arguments, specified by the query.

The two principal approaches to answering logic program queries are the top-down and bottom-up methods. The bottom-up approach computes all facts that can be inferred given a set of rules and facts and then selects the facts that have the same relation and constant arguments as the given query. This approach is likely to perform a much larger amount of computation than is actually needed. Thus, it may be very inefficient. The top-down approach for answering a query is computation on demand, guided by the given query. This approach is prone to repeated computation and infinite loops for certain kinds of programs and data.

A number of improvements have been developed for both approaches that resolve their major drawbacks. However, some difficult issues still remain. The efficiency of all existing

implementations is dependent on the order of hypotheses in the given rules or on the order of rules. Providing precise complexity analysis for answering logic program queries has been a major challenge for existing approaches.

We describe a method that combines a prominent bottom-up optimization called Magic Set Transformation (MST) [15] with a systematic method for deriving efficient algorithms and data structures from the rules [57]. The method focuses on Datalog, which is an important logic-based programming language and can be used to specify a significant class of practical problems. We apply the method to graph reachability, role based access control, information flow analysis, and model checking problems.

The rest of this chapter is organized as follows. Section 2 introduces Datalog rules and queries. Section 3 describes the transformation method, and Section 4 the method for complexity calculation. Section 5 presents applications of the approach. Section 6 discusses related work and concludes.

5.2 Problem and Approach

5.2.1 Queries

We consider queries of the form

$$P(X_1, ..., X_a)? \tag{5.1}$$

where P is a relation of a arguments, and each X_i is either a constant or a variable. The meaning of a query with respect to a set of rules and a set of facts is the set of facts that (1) are in the meaning of the given rules and given facts and (2) have the same relation and constant arguments as the given query. The use of queries allows the expression of only given or inferred facts that are of interest, not all facts that are given or can be inferred.

Example. We use various graph reachabilities as running examples. We use `a` and `b` to denote constants, i.e., specific vertices, and use `u`, `v`, and `w` to denote variables, i.e., any vertices. We use `edge(u,v)` to denote that there is an edge from a vertex `u` to a vertex `v`, and use `path(u,v)` to denote that there is path from a vertex `u` to a vertex `v` following the edges.

The following graph reachability variants are used as examples:

$$\begin{array}{l}\texttt{edge(u,v)}\rightarrow\texttt{path(u,v)}\\ \texttt{edge(u,w),path(w,v)}\rightarrow\texttt{path(u,v)}\end{array} \tag{5.2}$$

$$\begin{array}{l}\texttt{edge(u,v)}\rightarrow\texttt{path(u,v)}\\ \texttt{path(w,v),edge(u,w)}\rightarrow\texttt{path(u,v)}\end{array} \tag{5.3}$$

$$\begin{array}{l}\texttt{edge(u,v)}\rightarrow\texttt{path(u,v)}\\ \texttt{path(u,w),edge(w,v)}\rightarrow\texttt{path(u,v)}\end{array} \tag{5.4}$$

$$\begin{array}{l}\texttt{edge(u,v)}\rightarrow\texttt{path(u,v)}\\ \texttt{edge(w,v),path(u,w)}\rightarrow\texttt{path(u,v)}\end{array} \tag{5.5}$$

The meaning of each of these pairs of rules given a set of facts about edges is the set of facts that contains all `edge(u,v)`'s given and all `path(u,v)`'s that can be inferred. The set of all pairs `u` and `v` such that `path(u,v)` can be inferred is the transitive closure of edges in the graph.

The rules in 5.2 and 5.3 infer paths by starting at the end edges of paths. The two rules in 5.2 say that if there is an edge from `u` to `v`, then there is a path from `u` to `v`, and if there is an edge from `u` to `w` and there is a path from `w` to `v`, then there is a path from `u` to `v`. The two rules in 5.3 have the same meaning as the rules in 5.2. The only difference is that in 5.3 the order of hypotheses is reversed. The next two pairs of rules infer paths by starting at the starting edges of paths. The two rules in 5.5 have the same meaning as the rules in 5.4. The only difference is that in 5.5 the order of hypotheses is reversed.

Queries about paths are in one of the following four forms.

$$\texttt{path(a,b)?}\quad\texttt{path(a,v)?}\quad\texttt{path(u,b)?}\quad\texttt{path(u,v)?} \tag{5.6}$$

The meaning of `path(a,b)?` is the set of all `path(u,v)`'s that can be inferred and that $\texttt{u} = \texttt{a}$ and $\texttt{v} = \texttt{b}$, so it contain exactly `path(a,b)` if `path(a,b)` can be inferred, and is empty otherwise. The meaning of `path(a,v)?` is the set of all `path(u,v)`'s that can be inferred and that $\texttt{u} = \texttt{a}$, so it corresponds to the set of vertices reachable from `a` following edges in the graph. The meaning of `path(u,b)?` is symmetric and corresponds to the set of vertices that can reach `a`. Finally, the meaning of `path(u,v)?` is the set of all `path(u,v)`'s that can be inferred; so it gives exactly the transitive closure, omitting `edge(u,v)`'s in the meaning of the rules.

5.2.2 Efficient implementation with complexity guarantees

The problem considered in this chapter is to efficiently answer queries, i.e., compute the meaning of queries, and provide precise time complexity guarantees for such computations.

There are two main approaches to such a computation. The first one is a brute force strategy commonly referred to as bottom-up computation. This approach infers the meaning of the set of rules and facts and then selects the facts that have the same relation and

constant arguments as the given query. The inferred facts may be much more than the ones in the meaning of the query. Thus, the brute force approach may be very inefficient.

The second approach for computing the meaning of a query is computation on demand, usually referred to as top-down evaluation. The computation is guided by the given query. It starts with the query and the rules that have the query relation in their conclusions. New queries are generated as needed corresponding to the hypotheses of these rules and the available bindings for their arguments. Thus, the process is driven by the meanings of different queries. Since the computation is query specific it is likely to process fewer facts and be more efficient than the brute force approach. However, the top-down approach may repeatedly generate and evaluate the same subgoals, causing unacceptable performance. For many examples of rules and facts on-demand computation does not terminate. A rule is said to be *recursive* if the relation in the conclusion of the rule also occurs in a hypothesis in the rule. If this hypothesis is the first on in the rule, the rule is *left-recursive*. If a left-recursive rule is evaluated via top-down, it may cause non-termination. Moreover, even recursion that is not left-recursion in the rules can cause non-termination.

Example. For the query `path(a,v)?`, brute force evaluation infers all facts, and then selects the `path` facts with `u=a`. The on-demand strategy starts out with the query and returns `path(u,v)` facts with `u=a` in one step where `v`'s are the vertices reachable from `a` by traversing one edge. The second rule generates a `path(v,u)?` query, for each instantiation of `v`. Then the first rule is used again, and so on. The evaluation is guided by generated queries needed to compute the meaning of the original query. However, for certain data, the on-demand evaluation of the query `path(a,v)?` would never terminate. Such an example follows.

If the given facts are `edge(a,b)` and `edge(b,a)`, the evaluation proceeds as follows. First the first rule generates the query `edge(a,v)` and the first of the given facts is used to generate the fact `path(a,b)`. Then the second rule generates the queries `edge(a,v)?` and `path(v,w)?`. The first of the given facts binds `v` to vertex `a`. So the query `path(a,w)?` with first argument bound to vertex `a`, and second argument free, is generated again. This repeats infinitely.

A method for computing the meanings of queries is needed that combines the advantages of the two main approaches to query evaluation. The evaluation should process as few facts as possible. It should not depend on the order of rules and the order of hypotheses in them.

Our method is to transform a query and a rule set to a new set of rules and a fact. Given a set of facts, the facts of the query relation that are inferred by the new set of rules include those that are in the meaning of the query. The brute force evaluation of the new set of rules with respect to the facts mimics on-demand evaluation of the original query and set of rules.

For the query `path(a,v)?` and the set of rules in 5.2, our method produces the following set of rules and fact.

```
demand_path_bf(a)
demand_path_bf(u), edge(u,w)→demand_path_bf(w)
demand_path_bf(u), edge(u,v)→path_bf(u,v)
demand_path_bf(u), edge(u,w), path_bf(w,v)→path_bf(u,v)
```
(5.7)

The meaning of this set of rules and the new fact, given a set of facts about edges, contains all `path(u,v)` facts with `u=a` and only those `path` facts that would have been inferred in an on-demand evaluation of the original set of rules. This set of rules infers only paths that are reachable from vertex `a`. The worst-case time complexity is cubic in the number of vertices, as in the original set of rules, however, the actual complexity is dependent on the number of paths that are reachable from vertex `a`.

The method we describe consists of two steps. Step 1 defines a query specific set of rules and is described in the next section. Step 2 performs time complexity analysis of the set of rules and is described in Section 4.

5.3 Defining Query-Specific Rules

In this step a given set of rules and a query are transformed into a new set of rules and a fact whose meaning when projected on the query relation includes the meaning of the query.

Example. The query-specific for the query `path(a,v)?` and the set of rules in 5.2 were shown in 5.7.

Hypotheses of the `demand_path_bf` relation act as filters during the bottom-up evaluation of this program. They limit the number of facts that can be inferred because any inferred instances of the relation `demand_path_bf` would have an argument, say `a`, such that the query `path(a,v)?` would have been generated in the on-demand evaluation of the original set of rules and query. The instances of the `demand_path_bf` relation are the new fact whose argument is `a` — the bound one from the query, and the facts inferred by the rule that has the `demand_path_bf` relation in its conclusion. This rule infers `demand_path_bf` facts whose arguments are the vertices reachable from vertex `a` — the first argument of the query.

The last two rules infer facts of the `path_bf` relation. A fact `path_bf(u,v)` is inferred if `u` is a vertex reachable from vertex `a` and there is a path between `u` and `v`. The `demand_path_bf` hypotheses in the two rules ensure that the first argument of all inferred `path_bf` facts is a vertex reachable from `a`.

The Magic Set Transformation algorithm is used to obtain such query specific sets of rules.

5.3.1 Magic Set Transformation algorithm

This transformation is completed in three steps [15]. During the first step annotations are added to the query, the rules' conclusions, and hypotheses of relations that occur in rules' conclusions. We refer to such relations as *derived relations* and such hypotheses as *derived hypotheses*. They match argument bindings that will be used in the evaluation of the rules. In the second step a new relation is created. Hypotheses using the new relation are added to the rules, and rules with the new relation in conclusions are generated. In the third step a fact of the new relation is added and a rule with the original query relation in its conclusion is generated.

Step 1: Annotating bindings. Annotation is added to the query, the rules' conclusions and occurrences of derived relations in the rules. The purpose of the annotation is to capture the constants in the query, and define the way these constants are used in the rules.

An argument is *bound* in a relation occurrence if it is a constant and *free* if it is a variable. An *annotation* for an `n`-ary relation `p` is a string of length `n`, on the alphabet {`b, f`}, where `b` stands for bound and `f` stands for free [15]. An annotated occurrence of a relation `p` corresponds to inferring all facts of the relation with some arguments bound, and other arguments free. For instance `p_bbf` corresponds to computing `p` with the first two arguments bound and the last argument free, i.e., such a computation would have been demanded in a top-down evaluation. A relation annotated with only `f`'s corresponds to a computation of the complete relation, thus such annotations are discarded in this algorithm.

The query relation is annotated first. Initially its annotation is an empty string. For each argument in the query, if it is bound a `b` is added to the annotation, otherwise an `f` is added. Occurrences of the query relation in the rules' conclusions are annotated as the query.

Then, each derived hypothesis is annotated. Initially annotations are empty strings. For each argument of a hypothesis we add a character to the hypothesis's annotation as follows: (i) if the argument is a constant, a `b` is added; (ii) if the argument occurs in any of the hypotheses to the left of the current one, a `b` is added; (iii) if the argument occurs as bound in the rule's conclusion, a `b` is added; (iv) if none of the above cases holds, an `f` is added.

Example. The query `path(a,b)?` is annotated to become `path_bb(a,b)?`, since its two arguments are bound. This annotation corresponds to computing the `path` relation with both arguments bound. `path(a,v)?` is transformed to `path_bf(a,v)?` and corresponds to computing the `path` relation with the first argument bound. `path(u,b)?` with bound second argument becomes `path_fb(u,b)?`. `path(u,v)?` is not annotated since it has no bound arguments.

For the query `path_bf(a,v)?` and the set of rules in 5.2, the following set of rules results:

```
edge(u,v)→path_bf(u,v)
edge(u,w),path_bf(w,v)→path_bf(u,v)
```
(5.8)

The conclusions are annotated as the query. In the second rule the first argument of the `path(w,v)` hypothesis occurs in the `edge(u,w)` hypothesis to the left of it, so `w`'s position is marked as bound in the annotation. The rule stands for using bindings genarated by evaluating `edge(u,w)` hypothesis to evaluate the `path(w,v)` hypothesis with a bound first argument.

For the same query and the set of rules in 5.3, the following set of rules results:

```
edge(u,v)→path_bf(u,v)
path(w,v),edge(u,w)→path_bf(u,v)
```
(5.9)

In the second rule the hypothesis of the derived relation `path` would be evaluated without any bindings and thus has no annotation.

For the query `path(u,b)?` and the set of rules in 5.2, the following set of rules results:

```
edge(u,v)→path_fb(u,v)
edge(u,w),path_bb(w,v)→path_fb(u,v)
```
(5.10)

The second rule stands for evaluating the `edge(u,w)` hypotheses and using the bindings generated to evaluate the occurrence of the `path` relation with both arguments bound.

For the query `path(u,b)?` and the set of rules in 5.3, we get the set of rules:

```
edge(u,v)→path_fb(u,v)
path_fb(w,v),edge(u,w)→path_fb(u,v)
```
(5.11)

The second rule stands for evaluating the `path` relation first with only the second argument bound.

The annotation for the `path(a,b)?` query is as for the `path(u,b)?` query.

For the query `path(u,v)?` that does not have any bound arguments, we could annotate the rules in 5.2 as follows:

```
edge(u,v)→path(u,v)
edge(u,w),path_bf(w,v)→path(u,v)
```
(5.12)

The first rule has no annotation, and the second rule corresponds to computing the `path` relation facts with the first argument bound. However, the binding for the argument is passed from the `edge(u,w)` hypothesis, evaluated with both arguments free, and is therefore all vertices that have incoming edge. We cannot restrict the computation in the following steps of the transformation because of the lack of bound arguments in the query. In the following steps we omit this query.

There are cases in which this transformation is beneficial even if the query contains no bound arguments. When the occurrences of derived relations contain constant arguments, their annotations can be used to restrict the computation.

After annotating bindings, the rules may contain hypotheses of the query relation annotated differently from ones that occur in the annotated conclusions of the rules. Annotating bindings is repeated for the original rules and a new query — corresponding to the original query relation and a different annotation of the query relation that occurred, until all annotated relations appear in the conclusions of the rules.

Step 2: Adding demand relations. A new relation is created and hypotheses using it are added to the rules. These new hypotheses limit the inferred facts and thus act as filters during evaluation. We refer to them as *demand relations*.

A hypothesis named `demand_p_annotation` is added to each rule with a `p_annotation` conclusion relation. The arguments of the new hypothesis are the bound arguments in the rule's conclusion.

For each annotated hypothesis `p_annotation`, a new rule is generated. Its conclusion relation is `demand_p_annotation` and the conclusion's arguments are the bound arguments in `p_annotation`. The body of the new rule consists of all hypotheses to the left of the corresponding `p_annotation` occurrence except for those without bound arguments.

Example. The example set of rules 5.8 for the `path_bf(a,v)?` query and the set of rules in 5.2 is rewritten as follows:

```
demand_path_bf(u),edge(u,v)→path_bf(u,v)
demand_path_bf(u),edge(u,w),path_bf(w,v)→path_bf(u,v)
```

For both rules a `demand_path_bf` hypothesis is added corresponding to the conclusion relation. During evaluation the `demand_path_bf(u)` hypotheses generate bindings for the argument `u` and thus limit the `path_bf` facts inferred by the rules to ones in which the first argument is equal to `u`.

The following rule is added to the set of rules. It corresponds to the annotated occurrence of the `path` relation in the second rule.

```
demand_path_bf(u), edge(u,w)→demand_path_bf(w)
```

This rule infers facts of the `demand_path_bf` relation. It provides bindings for the arguments of the relation corresponding to the vertices whose outgoing paths are in the meaning of the query. The resulting rules for the set of rules in 5.9 are:

```
demand_path_bf(u),edge(u,v)→path_bf(u,v)
demand_path_bf(u),path(w,v),edge(u,w)→path_bf(u,v)
```

For the `path(u,b)?` query, for the set of rules in 5.10 the sets of resulting rules are:

```
demand_path_fb(v),edge(u,v)→path_fb(u,v)
demand_path_fb(v),edge(u,w),path_bb(w,v)→path_fb(u,v)    (5.13)
demand_path_fb(v),edge(u,w)→demand_path_bb(w,v)
```

The resulting rules for the set of rules in 5.11 are:

```
demand_path_fb(v),edge(u,v)→path_fb(u,v)
demand_path_fb(v),path_fb(w,v),edge(u,w)→path_fb(u,v)    (5.14)
demand_path_fb(v)→demand_path_fb(v)
```

Step 3: Adding a fact. The query is transformed to a fact which provides initial values for the arguments of the relation created in the previous step of the transformation. Thus the bindings from the query are reflected in the set of facts. The fact relation is the annotated query relation prepended with `demand_`. The arguments of the fact are the bound arguments in the query. For example, the demand fact for the query `path(a,b)?` is `demand_path_bb(a,b)`; for the query `path(a,v)?` — `demand_path_bf(a)`, and for the query `path(u,b)?` — `demand_path_fb(b)`.

Supplementary Magic Sets. Supplementary Magic Sets [15] is an additional transformation that is applied after MST, on the set of rules resulting from MST. This additional transformation is concerned with removing pairs of hypotheses that occur in more than one rule, and which cause duplication computation. In order to avoid recomputing pairs of hypothesis, the Supplementary Magic Set transformation introduces supplementary magic relations that are used to store intermediate results — results that have been computed and are likely to be useful later. The supplementary magic relations are auxiliary relations with the arguments necessary to combine two hypotheses. The transformation introduces new rules that have pairs of hypotheses from the set of rules in their bodies, and whose conclusion is of the supplementary magic relations. In this work we do not need to use the Supplementary Magic Sets transformation, since our method for generating efficient programs from rules splits all rules into ones with at most two hypotheses. For each rule with more than two hypotheses, we transform it to multiple rules with two hypotheses each. Moreover, we do this in all possible way and use the resulting set of rules with best time complexity.

5.3.2 Different hypotheses orders and MST

Changing the hypotheses order in the rules does not change the meaning of a set of rules with respect to any set of facts. The sets of rules in 5.2 and 5.3 differ in the order of hypotheses in the second rule, but have the same meanings. However, sets of rules produced by MST may be different for the same query and the same set of rules but with different orders of hypotheses in the rules. For instance, the sets of rules in 5.2 and 5.3 for the query `path(u,b)?` result in the different sets of rules in 5.13 and 5.14.

Different sets of resulting rules may have significantly different time complexities. In order to determine the most efficient query-specific set of rules, we could generate all possible query-specific sets of rules and choose the one with best time complexity. This can

be accomplished by generating all possible hypotheses orders for each rule and performing MST for all resulting sets of rules. We describe an improvement to this strategy here. The goal is to avoid generating results repeatedly by eliminating permutations of non-derived hypotheses between each two derived ones.

During MST, two aspects of the resulting set of rules are influenced by different hypotheses orders. These are: (i) the annotation of the derived hypotheses in rules, and (ii) the hypotheses in the bodies of the rules that have the demand relations in their conclusions. The following algorithm would generate all versions of a rule. For each rule in the set of rules:

- Generate all permutations of the derived hypotheses.
- Find all possible ways to place each of the remaining hypotheses before or after each of the derived ones.
- Make one version of the rule for each permutation of derived hypotheses and each of the ways to place remaining hypotheses.

The actual order of hypotheses that occur before a derived hypothesis or between two derived hypotheses is not of importance. This is why we eliminate some hypotheses orders automatically.

For example, if set A and set B constitute all non-derived hypotheses, and there is one derived hypothesis, we assume the body of the rule is `A, derived_hypothesis, B`. We do not make any assumptions about the order of hypotheses in the sets A and B.

Correctness. Only hypotheses orders that would produce repeated results after MST are eliminated. There is no resulting set of rules from MST that one can obtain by taking out all permutations of all hypotheses, and that our algorithm would omit.

The proof is by contradiction. Let us assume that there is a resulting set of rules from MST that is obtained by using all permutations of all hypotheses and is not obtained by our algorithm.

Let this different resulting set of rules correspond to the following permutation of rule and some query:

`R_1, d_1, R_2, d_2...., R_n, d_n, R_n+1`$\rightarrow$`c`

where `R_k`'s stand for sets of non-derived relations or the empty set and `d_k`'s stand for derived relations.

MST on this rule would produce:

`magic_c_ann, R1, d1_ann, R2, d2_ann...., Rn, dn_ann, Rn+1`
$\rightarrow$ `c_ann`
`magic_c_ann, R1` $\rightarrow$ `magic_d1_ann`
`magic_c_ann, R1, d1_ann, R2` $\rightarrow$ `magic_d2_ann`
...
`magic_c_ann, R1, d1_ann, R2, d2_ann...., Rn` $\rightarrow$ `magic_dn_ann`

We have two observations:

- Since our method uses all permutations of derived hypotheses, we have a rule in which d_1, d_2, ..., d_n appear in this same order.
- R_1, R_2, ..., R_n+1 are sets of non-derived hypotheses or empty sets, but include all non-derived hypotheses. The sets R_1, R_2,...,R_n+1 constitute a way to place all non-derived hypotheses in n+1 slots. Our method generates all such ways. Thus, an arrangement in which the sets R_1, R_2, ..., R_n+1 contain the exact same hypotheses as ones in the example and the sets appear in the same order, is generated by our approach.

The following are the two possibilities considered in the second observation:

Case 1: If the order of hypotheses in the R_k's is exactly the same as the one generated by our method, then the hypotheses order in the example rule is exactly the same. The result of MST on this rule is generated by our method. We reach a contradiction.

Case 2: If the order of hypotheses in some of the R_k's differs, we have:

$$\texttt{R\_1, d\_1, R\_2, d\_2...., R\_n, d\_n, R\_n+1} \rightarrow \texttt{c}$$

The annotations we obtain for d_k's are still equivalent to the ones above, since the same hypotheses occur to the left of each of the d_k's with either order within the R_k's. Moreover, the rules with demand relations in their conclusions contain the same hypotheses since the same sets of hypotheses occur to the left of each d:

$$\texttt{magic\_c\_ann, R1} \rightarrow \texttt{magic\_d1\_ann}$$
$$\texttt{magic\_c\_ann, R1, d1\_ann, R2} \rightarrow \texttt{magic\_d2\_ann}$$
$$\ldots$$
$$\texttt{magic\_c\_ann, R1, d1\_ann, R2, d2\_ann...., Rn} \rightarrow \texttt{magic\_dn\_ann}$$

Only the particular order of hypotheses in the rules with demand relations in their conclusions would differ. Thus, the rules are equivalent. This is a contradiction. The result is obtained by our method.

Any set of rules resulting from MST that is generated by taking all permutations of hypotheses in a rule is also generated by the suggested method. The algorithm still exhaustively tries all orders which produce different results after MST, however, it does so with an improved time complexity for the transformation method.

To characterize the improvement, we introduce the following notation.

- $\#d$: the number of derived hypotheses in a rule
- $\#r$: the number of non-derived hypotheses in a rule
- $\#h$: the total number of hypotheses in a rule

$\#d!$ represents the number of permutations permutations of derived hypotheses. $(\#d+1)^{\#r}$ is the number of different possibilities of placing the non-derived hypotheses. The number of hypothesis orders our method generates is $\#d! * (\#d+1)^{\#r}$. The straightforward method of generating all possible permutations of all hypotheses produces $\#h!$ which is $(\#d+\#r)!$. Our result is a significant improvement for rules with a small number of derived hypotheses.

5.4 Generating Efficient Implementations and Analyzing Complexity

This section describes the generation of efficient implementation from Datalog rules and computing precisely the time complexity of the generated algorithms, expressing the complexity in terms of characterizations of the facts. We transform Datalog rules into an efficient implementation using the method in [57]. For some of the applications that required Datalog rules with some extensions, we have implemented small extensions in the resulting algorithms as well.

5.4.1 Generating efficient implementations

Rules with multiple hypotheses. For each rule with more than two hypotheses we transform it to multiple rules with two hypotheses each. For example, the set of rules in 5.7 resulting from the example set of rules in 5.2 and the `path_bf(a,v)?` query contains a rule with three hypotheses:

```
demand_path_bf(u),edge(u,w),path_bf(w,v)→path_bf(u,v)
```

There are 3 ways of decomposing this rule, as follows. Combining the first two hypotheses into a new rule produces:

```
demand_path_bfEdge(u,w), path_bf(w,v)→path_bf(u,v)
demand_path_bf(u),edge(u,w)→demand_path_bfEdge(u,w)
```
(5.15)

Combining the first and third hypotheses into a new rule we obtain:

```
demand_path_bfPath_bf(u,w,v),edge(u,w)→path_bf(u,v)
demand_path_bf(u),path_bf(w,v)→demand_path_bfPath_bf(u,w,v)
```
(5.16)

Combining the second and third hypotheses results in:

```
edgePath_bf(u,v),demand_path_bf(u)→path_bf(u,v)
edge(u,w),path_bf(w,v)→edgePath_bf(u,v)
```
(5.17)

When calculating the time complexity for the set of rules we have to consider each of these three ways of splitting the above rule.

5.4.2 Computing time complexity

For each rule, we determine precisely the number of facts actually processed by it, avoiding approximations that use only the sizes of individual argument domains. Such complexity expressions would allow us to pick the best definition of each rule in a set of rules.

Size Parameters. We present the size parameters used for a relation `P`. We use `P.i` to denote the projection of `P` on its `i`-th argument. We use `P.I`, where `I = i1,i2,...,ik` to denote the projection of P on its `i1`-th, `i2`-th,..., and `ik`-th arguments. For any of the given relations we can use the following sizes:

- relation size

  `#P`: the number of facts that actually hold for relation `P`

- domain size

  `#D(P.i)`: the size of the domain from which the `i`-th argument of `P` gets its value

- argument size

  `#P.i`: the number of different values that `P.i` actually takes

  `#P.I`: the number of different combinations of values that elements of `P.I` together can actually take. For `I`=∅, `#P.I=1`.

- relative argument size

  `#P.i/j`: the maximum number of different values that `P.i` can actually take for each possible value of `P.j`, where `i`≠`j`.

  `#P.I/J`: the maximum number of different combinations of values that elements of `P.I` together can actually take for each possible combination of values of elements of `P.J`, where `I`∩`j`=∅. For `I`=∅, `#P.I/J=1`. For `J`=∅, we take `#P.I/J=#P.I`.

Example. We use as a running example the following set of rules, copied from 5.7:

```
demand_path_bf(u),edge(u,w)→demand_path_bf(w)
demand_path_bf(u),edge(u,v)→path_bf(u,v)
demand_path_bf(u),edge(u,w),path_bf(w,v)→path_bf(u,v)
path_bf(u,v)→path(u,v)
```

For the `demand_path_bf` relation, the size parameters are the following. `#demand_path_bf` is the size of the relation, i.e., the number of vertices which are the values of first arguments in demanded facts of the `path_bf` relation. `#D(demand_path_bf.1)` is the domain size for the argument of the `demand_path_bf` relation, i.e., the number of vertices in the graph. `#demand_path_bf.1` is the actual size of the argument, which is the number of vertices which are the values of first arguments in demanded facts of the `path_bf` relation.

For the `edge` relation, the size parameters are the following. `#edge` is the size of the `edge` relation, i.e., the number of edges in the graph. `#D(edge.1)` is the domain size for the first argument in edge, i.e., the number of vertices in the graph. `#edge.1` is the size of the first argument of the `edge` relation, i.e., the number of vertices that are sources of edges. `#edge.1/2` is a relative argument size and is the maximum number of predecessors of a vertex, i.e., the maximum indegree of vertices.

Size parameters are defined similarly for the remaining relations in the set of rules.

Basic constraints. The following basic constraints hold:

`#P`=`#P.`$\{$`1,...,a`$\}$ for relation `P` of `a` arguments
`#P.i`$\leq$`#D(P.i)`
`#P.I`$\leq$`#P.J` for `I`$\subseteq$`J`
`#P.(I`$\cup$`J)`$\leq$`#P.I`$\times$`#P.J/I` and `#P.I/J`$\leq$`#P.J` for `I`$\cap$`J`$=\emptyset$

For the running example, let `vertex` be the domain of the arguments of relation `edge`, and thus also the domain of the arguments of `path_bf`, `demand_path_bf`, and `path`. The constraints are:

```
#demand_path_bf.1≤#D(demand_path_bf.1)=#vertex
#demand_path_bf≤#D(demand_path_bf.1)=#vertex
#path_bf.2/1≤#path_bf.2≤#D(path_bf.2)=#vertex
#path_bf.1/2≤#path_bf.1≤#D(path_bf.1)=#vertex
#path_bf≤#D(path_bf.1)×#D(path_bf.2)=#vertex²
#path.2/1≤#path.2≤#D(path.2)=#vertex
#path≤#D(path.1)×#D(path.2)=#vertex²
#edge.1/2≤#edge.1≤#D(edge.1)=#vertex
#edge≤#D(edge.1)×#D(edge.2)=#vertex²
```

Additional constraints that capture dependencies among relations and relation arguments can be defined and used in the time complexity calculation. Such constraints, if available, can provide more precise results depending on the information that is available about the relations and their arguments.

Time complexity. The time complexity for a set of rules is the total number of firings of the rules. For each rule `r`, `r.#firedTimes` stands for the number of firings of the rule, i.e., `r.#firedTimes` is a count of:

- For rules with one hypothesis: the number of facts which make the hypothesis true.
- For rules with two hypotheses: the number of combinations of facts which make the two hypotheses simultaneously true.

For a rule `r`, `r.#firedTimes` is calculated as follows. We use `IX`'s to denoted the indices of arguments `X`'s in a hypothesis. If the rule has one hypothesis, say `P`, we have `r.#firedTimes = #P`. If the rule has two hypotheses, say `P1` and `P2`, we have:
`r.#firedTimes`$\leq$`min(#P1`$\times$`#P2.IX2s/IYs, #P2`$\times$`#P1.IX1s/IYs)`.
Here `IY`'s are the indices of arguments in `P1` or `P2` which occur in both hypotheses.

For any set of rules with given characteristics of facts in terms of the four kinds of size parameters and the constraints on these sizes as described, the total time complexity for the set of rules is the sum of `#firedTimes` over all rules, minimized symbolically with respect to the given sizes and the constraints. Specifically, we can decide in what terms we would like the complexity formula to be expressed. If a relative argument size is used but not given. we use the corresponding non-relative argument size. If an argument size `#P.I` is used but not given we use the minimum of (i) the product of domain sizes for arguments of `P` that are in `I` and (ii) the argument size of `P` for arguments that are a superset of `I`, if given.

Our complexity calculation method allows us to express the time complexity of sets of rules in different parameters and make an accurate decision, depending on the characteristics of the relations.

Example. We use as an example the following set of rules resulting from the example set of rules in 5.7 and the first version of splitting the rule with three hypotheses into shorter rules in 5.15.

```
demand_path_bf(u), edge(u,w) → demand_path_bf(w)
demand_path_bfEdge(u,w), path_bf(w,v) → path_bf(u,v)
demand_path_bf(u),edge(u,w) → demand_path_bfEdge(u,w)
demand_path_bf(u),edge(u,v) → path_bf(u,v)
```

Each of the following complexity formulas is for the corresponding rule in the above set of rules:

```
O(min(#demand_path_bf×#edge.2/1, #edge ×1))
O(min(#demand_path_bf×#edge.2/1, #edge ×1))
O(min(#demand_path_bf×#edge.2/1, #edge ×1))
O(min(#demand_path_bfEdge×#path_bf.2/1,
      #path_bf×#demand_path_bfEdge.1/2))
O(#path_bf)
```
(5.18)

The time complexity of the set of rules is the sum of the formulas listed for each of the rules.

When only #edge and #vertex are used as parameters in the complexity formula, based on the basic constraints this sum is bounded by:

$$O(\min(\#vertex^2, \#edge) + \#vertex \times \#edge, \#edge + \#vertex^2)$$

Simplifying it further based on $\#edge \leq \#vertex^2$, we get $O(\#vertex^2)$.

The second formula, shown in 5.18, demonstrates that the efficiency of the set of rules is mostly dependent on the number of edges in the graph and the number of vertices which have incoming edges. Expressing the complexity in different size parameters enables us to make more accurate comparisons among sets of rules.

For the query path(a,v)? we obtain the algorithm with the smallest time complexity by applying MST to the specification 5.4. We achieved an improved time complexity over computing all paths. The following set of rules results from applying MST:

```
demand_path_bf(a)
demand_path_bf(x),edge(x,y)→path_bf(x,y)
demand_path_bf(x),path_bf(x,z),edge(z,y)→path_bf(z,y)
demand_path_bf(x)→demand_path_bf(x)
```
(5.19)

The last rule never infers any new fact, so it can be removed from the set of rules with no effect on the rules' meaning. The second rule has three hypotheses, and splitting it into two rules in the optimal way is by combining the first two hypotheses in a separate rule. We thus get the following set of rules:

```
demand_path_bf(a)
demand_path_bf(x),edge(x,y)→path_bf(x,y)
demand_path_bf(x),path_bf(x,z)→demand_path_bf_path_bf(x,z)
demand_path_bf_path_bf(x,z),edge(z,y)→path_bf(x,y)
```
(5.20)

Since no facts of the demand_path_bf relation can be inferred, and only one fact of the relation is given, the size of this relation is 1 throught the computation. The size of the demand_path_bf_path_bf relation can be estimated by use of the following constraint:

$$\begin{aligned}&\texttt{\#demand\_path\_bf\_path\_bf} \leq \\ &\texttt{\#demand\_path\_bf\_path\_bf.1} \times \texttt{\#demand\_path\_bf\_path\_bf.2/1}\end{aligned} \tag{5.21}$$

The number of different values the first argument of the relation can take is 1. This is evident from the second rule — this argument can only get values from the argument of the `demand_path_bf` hypothesis, and we already concluded there is only one such value. Thus:

$$\begin{aligned}&\texttt{\#demand\_path\_bf\_path\_bf} \leq \\ &\texttt{1} \times \texttt{\#demand\_path\_bf\_path\_bf.2/1} \leq \\ &\texttt{\#demand\_path\_bf\_path\_bf.2/1} \leq \texttt{\#edge.2}\end{aligned} \tag{5.22}$$

The time complexity for the set of rules in 5.20 is thus:

$$\begin{aligned}&\texttt{O(\#edge.2/1 + \#edge.2 + min(\#edge} \times \texttt{1, \#edge.2} \times \texttt{\#edge.2/1))} \leq \\ &\texttt{O(min(\#edge, \#edge.2} \times \texttt{\#edge.2/1))}\end{aligned} \tag{5.23}$$

The time complexity of the algorithm generated is thus the minimum of linear in the number of edges in the graph, and the number of vertices with incoming edges times the maximum indegree of vertices in the graph.

For the query `path(u,b)?` the algorithm with the best time complexity is generated by using the specification in 5.3 instead. The following rule set results from applying MST:

$$\begin{aligned}&\texttt{demand\_path\_fb(b)} \\ &\texttt{demand\_path\_fb(y),edge(x,y)} \rightarrow \texttt{path\_fb(x,y)} \\ &\texttt{demand\_path\_fb(y),path\_fb(z,y),edge(x,z)} \rightarrow \texttt{path\_fb(z,y)} \\ &\texttt{demand\_path\_fb(y)} \rightarrow \texttt{demand\_path\_fb(y)}\end{aligned} \tag{5.24}$$

The last rule never infers any new fact, so it can be removed from the set or rules with no effect on the set's meaning. The second rule has three hypotheses and splitting it into two rules in the optimal way is by combining the first two hypotheses in a separate rule. We thus get the following set of rules:

```
demand_path_fb(b)
demand_path_fb(y),edge(x,y)→path_fb(x,y)
demand_path_fb(y),path_fb(z,y)→demand_path_fb_path_fb(z,y)
demand_path_fb_path_fb(z,y),edge(x,z)→path_fb(z,y)
```
(5.25)

Since no facts of the `demand_path_fb` relation can be inferred, and only one fact of the relation is given, the size of this relation is 1 throught the computation. The size of the `demand_path_fb_path_fb` relation can be estimated by use of the following constraint:

```
#demand_path_fb_path_fb≤
#demand_path_fb_path_fb.1×#demand_path_fb_path_fb.2/1
```
(5.26)

The number of different values the first argument of the relation can take is 1. This is evident from the second rule — this argument can only get values from the argument of the `demand_path_fb` hypothesis, and we already concluded there is only one such value. Thus:

```
#demand_path_fb_path_fb≤
1×#demand_path_fb_path_fb.2/1≤
#demand_path_fb_path_fb.2/1≤#edge.1
```
(5.27)

The time complexity for the set of rules in 5.24 is thus:

```
O(#edge.1/2 + #edge.1 + min(#edge×1, #edge.1×#edge.1/2))≤
O(min(#edge, #edge.1×#edge.1/2))
```
(5.28)

The time complexity of the algorithm generated using is thus the minimum of linear in the number of edges in the graph, and the number of vertices with outgoing edges times the maximum outdegree of vertices in the graph.

The time complexity of computing the complete `path` relation by the reachability definitions is significantly higher than the ones we obtained for queries on the `path` relation using MST. The graph reachability specifications in 5.2, 5.3, 5.4, and 5.5 all have the same first rule, and its time complexity is `O(#edge)`. For each of these specifications the time complexity is dominated by the second rule. The time complexity for the second rule, and thus the whole specification is:

$$\begin{array}{ll} \text{specifications in 5.2 and 5.3:} & \texttt{O(min(\#edge}\times\texttt{\#path.2/1,} \\ & \texttt{\#path}\times\texttt{\#edge.1/2))}\leq \\ & \texttt{O(\#edge}\times\texttt{\#edge.2)} \\ \text{specifications in 5.4 and 5.5:} & \texttt{O(min(\#edge}\times\texttt{\#path.1/2,} \\ & \texttt{\#path}\times\texttt{\#edge.2/1))}\leq \\ & \texttt{O(\#edge}\times\texttt{\#edge.1)} \end{array} \tag{5.29}$$

5.5 Applications

We applied the transformation and complexity analysis to a number of analysis problems and obtained more precise algorithm complexities and better understanding for them.

We implemented the Magic Set Transformation algorithm. Python 2.3 was used for the implementation. The algorithm first generates all relevant orders of hypotheses for each of the given rules, and then performs the Magic Set Transformation. Finally, all sets of rules resulting from the different hypotheses orders are output, along with time complexity analysis for each of them. The sets of rules in all examples in this chapter were generated automatically using our implementation.

5.5.1 Graph reachability

Graph reachability finds all vertices reachable in a graph from a given set of source vertices. Graph reachability is a fundamental problem and has extensive applications in many other problems, such as ones in program analysis, model checking and security frameworks. The rules for transitive closure along with all different queries for it were discussed in the previous subsection as running examples. The complexity formulas for each program and query give specific information, which allows for a tighter complexity bound. Complexity can often be expressed in terms of the out-degree or in-degree of vertices which is significantly better than using the number of vertices in the graph. Thus, the complexity analysis results we obtained are drastically more precise.

The time complexity of computing the complete `path` relation for any of the specifications is shown in 5.29. This would also be the time complexity of computing any of the queries on paths in a graph, if we compute the complete `path` relation and then select the facts that are in the meaning of the query. By use of the method described in this section, we can transform queries that contain any bound arguments, and a reachability specification, to a new rule set, and generate an algorithm with significantly improved time complexity.

To experimentally confirm our time complexity calculations, we generated implementations of the algorithm for the original rules and the algorithms resulting from MST in Python. We generated implementations for the query `path(a,v)?` for the algorithms generated from the rules in 5.4 and the rules resulting from MST on the rules in 5.4, as

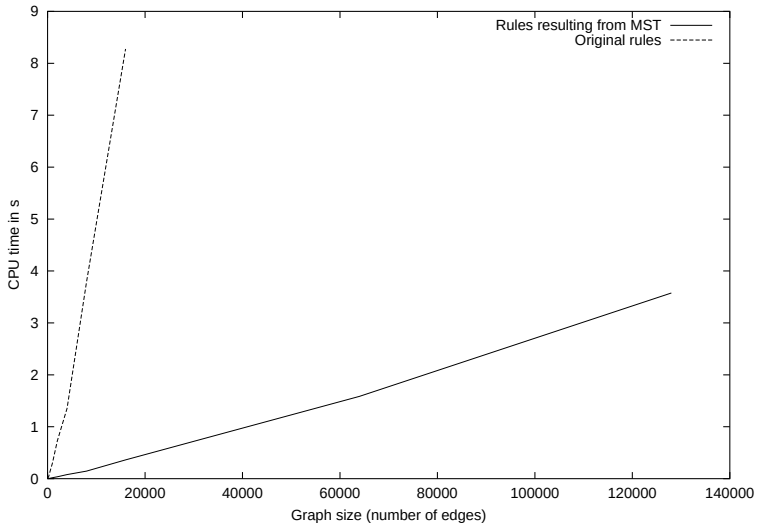


Figure 5.1: A comparison of CPU time for the query `path(a,v)?` for the algorithms generated from the rules in 5.4 and the set of rules resulting from MST on the rules in 5.4.

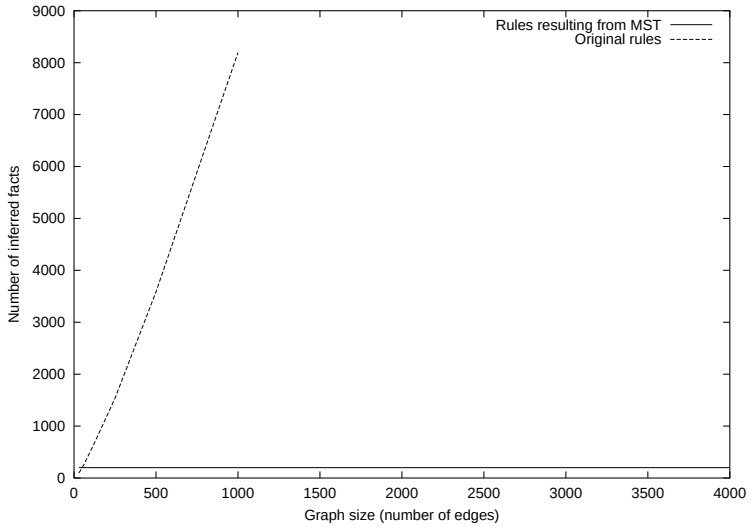


Figure 5.2: A comparison of the number of inferred facts (total of `path` and `demand` relations) for the query `path(a,v)?` for the algorithms generated from the rules in 5.4 and the set of rules resulting from MST on the rules in 5.4.

well as for the query `path(u,b)?` for the algorithms generated from the rules in 5.3 and the set of rules resulting from MST on the rules in 5.3. The generated implementations consisted of 40 lines of Python code for the original sets of rules and 90 lines of Python code for the sets of rules resulting from MST. We analyzed graphs trees of varying size, to determine how the running time of the algorithms scales with graph size. All experiments were conducted using Python 2.5 on a 794MHz Intel Pentium M 1.73GHz with 448

Megabytes of RAM, running Microsoft Windows XP Professional.

For the experiments on the query `path(a,v)?` we used a dataset of binary trees that grow in size by having a new layer of leaves added on each test run. The queries asked were searching for paths to different vertices, but the distance between the vertex in the query and the leaves of the tree remained constant as the tree grew.

For the experiments on the query `path(u,b)?` we used a dataset of binary trees that grow in size by having a new layer of leaves added on each test run. The queries asked were searching for paths to the same vertices, which remained equidistant from the root of the tree regardless of the size of the tree.

Figures 5.1 and 5.3 report CPU time it took to answer the queries for each size of tree in the data set, using each of the two generated algorithms. Reported times were averaged over 10 trials. It is evident that the CPU time to answer the query grew linearly with respect to the number of edges in the graph for the algorithm generated from the rules resulting from MST, and it grew polynomially for the algorithm generated from the reachability rules without MST. As a matter of fact, the latter algorithms became impractically slow form much smaller graphs.

Figures 5.2 and 5.4 report the number of facts that were inferred for each size of tree in the data set, using each of the two generated algorithms. The algorithm generated from the rules resulting from MST inferred a constant number of facts regardless of the size of the tree, while the number of inferred facts grew polynomially for the algorithm generated from the reachability rules without MST.

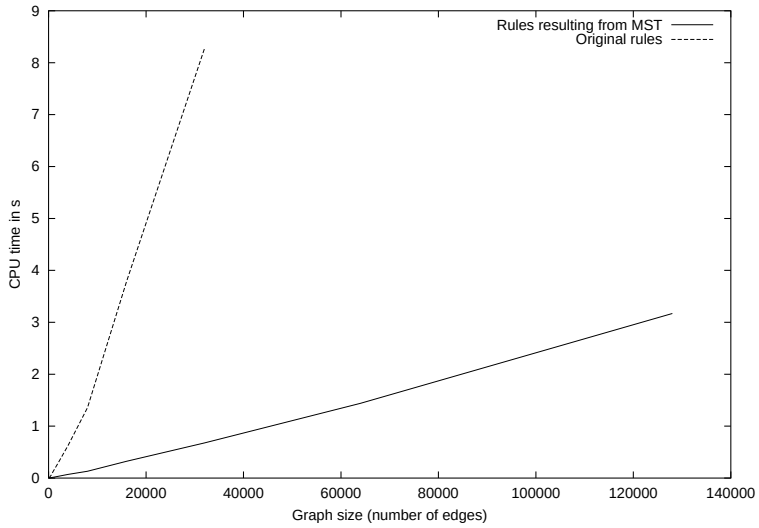


Figure 5.3: A comparison of CPU time for the query `path(u,b)?` for the algorithms generated from the graph reachability rules in 5.3 and the set of rules resulting from MST on the specification in 5.3.

In both cases, it is evident that the algorithms generated from the rules resulting from MST infer much fewer facts and can thus accomodate much larger graphs.

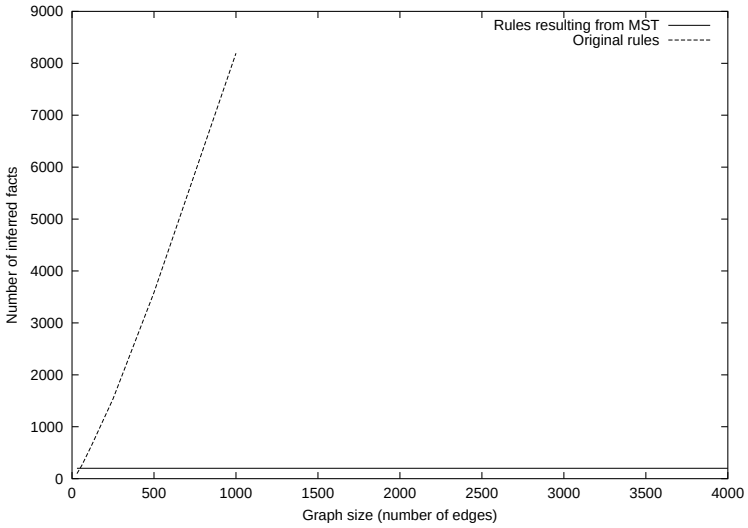


Figure 5.4: A comparison of the number of inferred facts (total of `path` and `demand` relations) for the query `path(u,b)?` for the algorithms generated from the graph reachability rules in 5.3 and the set of rules resulting from MST on the specification in 5.3.

We also conducted experiments using some of the worse sets of rules that result from MST. We used a dataset of complete graphs that grow in size by having five new vertices, and the corresponding edges, added on each test run. The queries asked were searching for paths to and from the same vertices. Figures 5.5 and 5.6 report the number of facts inferred in the process of answering the queries for each size of graph in the data set, using each of the four generated algorithms. It is evident that the number of facts inferred in each case was superlinear with respect to the number of vertices in the graph. However, the number of facts inferred by the algorithms generated from the rules resulting from MST, was larger that inferred by the algorthms generated from the reachability rules without MST. The reason for this is that all algorithms inferred all paths in the complete graph, but, in addition, the former algorithms inferred a number of facts of the `demand` relations introduced by MST.

In both cases it is evident that the algorithms generated from the rules resulting from MST inferred more facts and were thus more costly to use.

5.5.2 Model checking pushdown systems

In Chapter 2 of this thesis we described the design, analysis and implementation of an algorithm for LTL model checking of PDS. The central part of the algorithm was computing the reach graph. In that algorithm, when generating the reach graph we computed all of its edges and then only looked for a good cycle in the graph. A more sophisticated on demand method would be more efficient, as in many cases we never need the complete

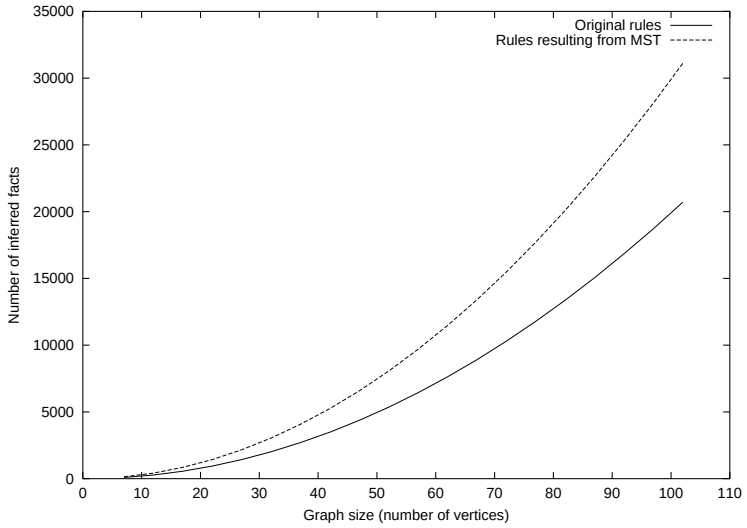


Figure 5.5: A comparison of the number of inferred facts (including facts of `demand` relations) for the query `path(a,v)?` for the algorithms generated from the rules in 5.2 and the set of rules resulting from MST on the rules in 5.2.

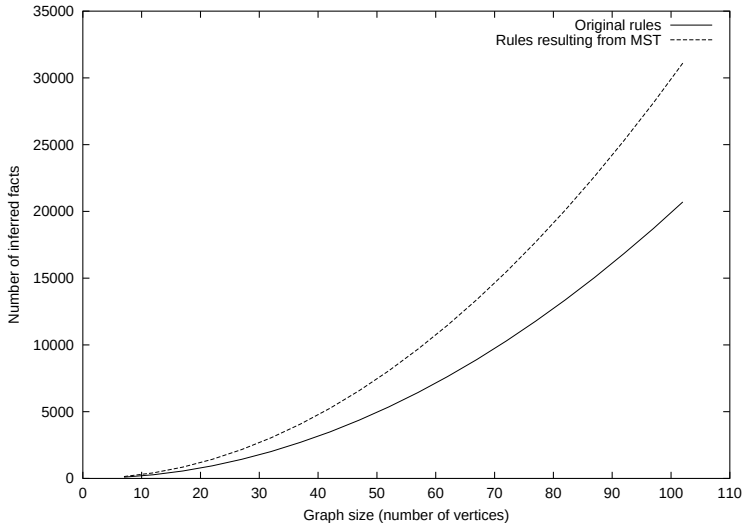


Figure 5.6: A comparison of the number of inferred facts (including facts of `demand` relations) for the query `path(u,b)?` for the algorithms generated from the rules in 5.4 and the set of rules resulting from MST on the rules in 5.4.

reach graph. The algorithm used to detect a good cycle can be used to guide us in computing only the parts of the reach graph that are demanded in the search for a good cycle. In that algorithm, there are several queries that require us to find all edges starting from one specific vertex in the graph, that is, they require us to compute the answer to the query `edge(a1,b1,g,c2,s2)?` where `(a1,b1)` is a configuration in the product PDS and

```
demand_edge_bbfff(startC,startS)
demand_erase_bbff(c1,s1),trans0(c1,s1,c2),loc(c1,g)
    →erase_bbff(c1,s1,g,c2)
demand_erase_bbff(c1,s1),trans1(c1,s1,c2,s2),erase_bbff(c2,s2,g2,c3),
    loc(c1,g1),or(g1,g2,g) →erase_bbff(c1,s1,g,c3)
demand_erase_bbff(c1,s1),trans2(c1,s1,c2,s2,s3),erase_bbff(c2,s2,g2,c3),
    erase_bbff(c3,s3,g3,c4),loc(c1,g1),or(g1,g2,g4),or(g4,g3,g)
    →erase_bbff(c1,s1,g,c4)
demand_erase_bbff(c1,s1),trans1(c1,s1,c2,s2) →demand_erase_bbff(c2,s2)
demand_erase_bbff(c1,s1),trans2(c1,s1,c2,s2,s3) →demand_erase_bbff(c2,s2)
demand_erase_bbff(c1,s1),trans2(c1,s1,c2,s2,s3),erase_bbff(c2,s2,g2,c3)
→demand_erase_bbff(c3,s3)
demand_edge_bbfff(c1,s1),trans1(c1,s1,c2,s2),loc(c1,g)
    →edge_bbfff(c1,s1,g,c2,s2)
demand_edge_bbfff(c1,s1),trans2(c1,s1,c2,s2,s3),loc(c1,g)
    →edge_bbfff(c1,s1,g,c2,s2)
demand_edge_bbfff(c1,s1),trans2(c1,s1,c2,s2,s3),erase_bbff(c2,s2,g2,c3),
    loc(c1,g1),or(g1,g2,g) →edge_bbfff(c1,s1,g,c3,s3)
demand_edge_bbfff(c1,s1),trans2(c1,s1,c2,s2,s3)→demand_erase_bbff(c2,s2)
```

Figure 5.7: Rules for computing a portion of the reach graph on demand.

`a1` and `b1` are bound.

We used the Magic Set Transformation, to achieve on-demand computation of the reach graph. We were able to generate complete algorithms and data structures with precise complexity guarantees. The rule set resulting from applying MST to the rules presented in Chapter 2 for computing the edges of the reach graph and the relevant query is shown in Figure 5.7. Rather than computing the complete reach graph, these rules only compute the edges starting with a certain configuration, i.e., they only compute the part of the reach graph that is reachable from a certain configuration.

To experimentally confirm our time complexity calculations, we generated an implementation of the reach graph computation algorithm and the algorithms resulting from MST, shown in Figure 5.7. The generated implementations consisted, respectively, of 400 and 600 lines of Python code. We analyzed PDS of varying size, to determine how the running time of the algorithms scales with PDS size. All experiments were conducted using Python 2.5 on a 794MHz Intel Pentium M 1.73GHz with 448 Megabytes of RAM, running Microsoft Windows XP Professional.

For the algorithm resulting from MST, we computed three parts of the reach graph, starting from three different configurations, chosen randomly, and report the average time for these computations.

Figure 5.8 reports CPU time it took to answer the queries for each size of tree in the data set, using each of the two generated algorithms. Reported times were averaged over 10 trials. It is evident that the CPU time to compute the reach graph grew at a much higher rate than the CPU time for the on-demand computation.

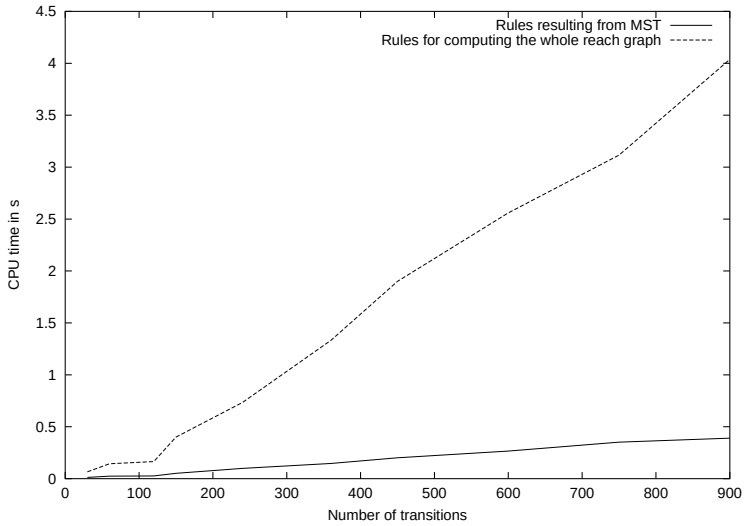


Figure 5.8: A comparison of CPU time for the algorithm that computes the complete reach graph and the algorithm resulting from MST that computes parts of the reach graph on demand.

5.5.3 Trust management policy analysis

In Chapter 4 of this thesis we derive an algorithm to compute the reduction closure. The precise time complexity for this algorithm is `#cert`$\times$`#cert.3/{1,2,4=[]}`, where `cert` is the set of inferred certificates, and `#cert.3/{1,2,4=[]}` is the maximum number of keys a single local name reduces to (this may be more than one, because local names may represent groups). In the worst case, the latter may include all keys (`key`) that appear in the certificates. As noted in [49], `#cert` is bounded by `in`$\times$`#key`, where `in` is the size of the input, i.e., the sum of the sizes of the given certificates. The auxiliary space is used for mapping key-id pairs to key-id pairs. Our complexity analysis is more precise and informative than using only worst-case sizes [25, 49]. Our method allows these complexities, together with efficient algorithms that realize them, to be automatically derived from the rules.

The algorithms for computing reduction closure can be used to solve specialized analysis problems. However, these algorithms compute all authorizations and all name-key correspondences, given a set of certificates. This may be unnecessary, since many policy analysis problems require computing only a few authorizations or resolving only a few names. Therefore, we use specialized extended Datalog rules for the specialized analysis problems; these specialized rules can be used to generate an efficient algorithm for each analysis problem, and infer only the authorizations and resolve only the names needed for that problem. Also, the original reduction closure algorithm does not give a direct way of solving some important policy analysis problems, specifically when questions about name certificates are asked, when sets of resources or keys are given. There are algorithms for solving these problems in [49], but these require complex pushdown system structures that

are not inherent to the problems' structure.

In Chapter 4 of this thesis, we first introduced extended Datalog rules to solve the problems and then showed a way to construct specialized rules from given rules, by pushing the constants bound by the query into the rules. However,

We demonstrate the use of MST to create specialized rules on one of the analysis problems. Consider the rule set and the query for the problem Compromisation Assessment 2:

```
canAccess(k,p,t), ¬canAccess2(k,p,t)
    →compromisedPrinciples(p,t,k)
Query : compromisedPrinciples(P,T,k).
```

Now since the permission `P` and time `T` are given when the question is asked, we apply MST with the query `compromisedPrinciples(P,T,k)?`, yielding:

```
demand_compromisedPrinciplesfbb(p,t),canAccessfbb(k,p,t),
canAccess2bbb(k,p,t)
    →compromisedPrinciples(p,t,k)
demand_compromisedPrinciplesfbb(p,t)
    →demand_canAccessfbb(p,t)
demand_compromisedPrinciplesfbb(p,t),canAccessfbb(k,p,t)
    →demand_canAccess2bbb(k,p,t)
```

These new rules are the `compromisedPrinciples` rule specialized to constants `P` and `T`. Notice that MST can be applied iteratively in general, until recursion is encountered in the rules. In other words, in this example the rules for `canAccess(k,P,T)` can be rewritten by MST to yield a rule set specialized to the `canAccess(k,P,T)?` query, where `P` and `T` are bound.

All specilized rules for the policy analysis problems are given in Figure 5.9.

MST yields better algorithms and time complexity results than the push-and-specialize method described in Chapter 4 of this thesis, because in addition to pushing the constants in the query into the hypotheses, it can also make use of any bindings of arguments acquired from hypotheses that are to the left of any hypothesis. In the example above, the push-and-specialize method yielded rules that still needed to evaluate the `canAccess2` hypothesis with only two bount arguments — the ones give in the query. The set of rules resulting from MST has an addition binding for the canAccess2 hypothesis — the argument `p` is bound by the hypotheses to the left of `canAccess2`, and thus `canAccess2` needs to be evaluated with all three arguments bound.

We limited use of MST to the non-recursive rules of specialized problems. We did not apply MST to try and compute only certain parts of the reduction closure for the following reason. The rules for computing the reduction the closure include 3 rules with 2 inferred hypotheses in each. MST was only useful in some cases. Specifically, it is not useful for the queries most frequently needed in specialized problems — find all permissions that a

Shared Access 1:

```
demand_sharingPrinciplebbbf(p1,p2,T),canAccessfbb(k,p1,t),
canAccessfbb(k,p2,t)
    → sharingPrinciple(p1,p2,t,k)
demand_sharingPrinciplebbbf(p1,p2,T)→ demand_canAccessfbb(p1,t)
demand_sharingPrinciplebbbf(p1,p2,T),canAccessfbb(k,p1,t)
    → demand_canAccessfbb(p2,t)
```

Shared Access 2:

```
demand_sharingResourcebbbb(k1,k2,p,t),canAccessbbb(k1,p,t),
canAccessbbb(k2,p,t)
    → sharingResource(k1,k2,p,t)
demand_sharingResourcebbbb(k1,k2,p,t) → demand_canAccessbbb(k1,p,t)
demand_sharingResourcebbbb(k1,k2,p,t),canAccessbbb(k1,p,t)
    → demand_canAccessbbb(k2,p,t)
```

Shared Access 3:

```
demand_sharingResources(k1,k2,ps,t), p in ps,
sharingResourcebbbb(k1,k2,p,t)
    → sharingResources(k1,k2,ps,t,p)
demand_sharingResources(k1,k2,ps,t), p in ps,
    → demand_sharingResourcebbbb(k1,k2,p,t)
```

Compromisation Assessment 1:

```
demand_compromisedResource(k,ps,t), p in ps,
canAccessbbb(k,p,t), ¬canAccess2bbb(k,p,t)
    → compromisedResource(k,ps,t,p)
demand_compromisedResource(k,ps,t), p in ps
    → demand_canAccessbbb(k,p,t)
demand_compromisedResource(k,ps,t), p in ps,
canAccessbbb(k,p,t)
    → demand_canAccess2bbb(k,p,t)
```

Compromisation Assessment 2:

```
demand_compromisedPrinciplesfbb(p,t),canAccessfbb(k,p,t),
canAccess2bbb(k,p,t)
    → compromisedPrinciples(p,t,k)
demand_compromisedPrinciplesfbb(p,t)
    → demand_canAccessfbb(p,t)
demand_compromisedPrinciplesfbb(p,t),canAccessfbb(k,p,t)
    → demand_canAccess2bbb(k,p,t)
```

Universally Guarded Access 1:

```
demand_needNotinvolvebb(p,t),canAccessfbb(k1,p,t), canAccess2fbb(k1,p,t)
    → needNotInvolve(p,t)
demand_needNotinvolvebb(p,t) → demand_canAccessfbb(p,t)
demand_needNotinvolvebb(p,t),canAccessfbb(k1,p,t)
    → demand_canAccess2fbb(p,t)
```

Universally Guarded Access 2:

```
demand_needNotInvolveMultiplebbb(k,ps,t), p in ps,
canAccessbbb(k,p,t), canAccess2bbb(k,p,t)
    → needNotInvolveMultiple(k,ps,t)
demand_needNotInvolveMultiplebbb(k,ps,t), p in ps,
    → demand_canAccessbbb(k,p,t)
demand_needNotInvolveMultiplebbb(k,ps,t), p in ps,
canAccessbbb(k,p,t)
    → demand_canAccess2bbb(k,p,t)
```

Figure 5.9: Specialized rules for solving policy analysis problems.

certain principal is authorized to. It is not useful for this because with MST we get several annotations of both auth and name relations and essentially end up computing the complete relations anyways. So MST would introduce extra rules, relations and hypotheses and would not make any positive difference in the computation.

5.5.4 Type inference for secure information flow

In Chapter 3 of this thesis, we describe the design, analysis, and implementation of an efficient algorithm for information flow analysis expressed using the type system presented by Volpano et al. in [88]. Our algorithm is linear in the size of the given program.

The Datalog rules used for type inference for secure information flow are shown in Figures 3.3 and 3.4. These rules infer types for all program nodes, if such types exist, and infer errors if there is no correct typing of the program. Using MST may seem incompatible with the concept of type inference as the idea of type inference is to infer a typing of the whole program, or to infer errors if such a typing does not exist. However, in some cases it may be useful to be able to infer only the types of certain nodes or check for errors caused by specific variables or commands. In those cases we can use MST to generate specialized rules and algorithms. Examples of such queries are:

- Find the type of one program node only
- Find out if one assignment command causes a type error
- Find out if there is a type error associated with one variable
- Find the highest possible type for a specific command

If we query the type of root, an array, a literal, or an `arrlen` expression the MST rules compute only the queried type in time $O(1)$. If we query (or generate demand fact for) the type of a subcommand in an `if` command, or a `sequence`, these rules would compute only the type of the subcommand in the query and would not compute the type of the other subcommands, saving some computation. If we query the type of a variable, then both types of expressions and commands are demanded, which triggers propagation of types both up and down the syntax tree. Depending on the commands in which the variable is used, some computation may be saved, e.g., if the variable is only used in one branch of an `if` command or in one subcommand of a `sequence` command. Queries to all other kinds of commands and expressions can generate demand facts for the types of variables, so the above holds.

The time complexity of type inference for the entire program is $O(\mathtt{p}\times\mathtt{h}\times\log\ \mathtt{s})$, where `p` is the size of the program, i.e., the number of nodes in the program AST, `h` is the height of the security type lattice, and `s` is the size of the security type lattice.

It is possible to precompute all values for the functions `Join` and `Meet` in $O(\mathtt{s}^2\times\log\ \mathtt{s})$ time, and, if we do so, any of them can be looked up on `O(1)` time. However, this may

be unnecessary, since it is possible that not all values of these functions are needed. Therefore, we compute the values of `Join` and `Meet` as needed and memoize already computed values which can be looked up in $O(1)$ time if needed again. The time complexity of computing `Join` or `Meet` for two security types is $O(\log s)$. The type complexity of computing whether the subtyping relation holds between two types is $O(h)$.

The algorithm for type inference traverses the program top-down multiple times to infer the `type` relation, i.e., minimum expression types, and then traverses the program bottom-up once to infer the `htype`, i.e., maximum command types. Facts of the `type` relation for variables can be inferred by use of the `LETID` rules. New types for variables can be inferred by use of the `ASSIGN ID` rule. This can cause facts of the `type` relation for other variables to be inferred. At any point in the evaluation at most one fact of the `type` relation is kept for a program node, and that is the one with the highest type for the program node that has been inferred so far. The type of each program node can be raised at most `h` times. Thus worst case time complexity for each of the extended Datalog rules for type inference is equal the program size multiplied by the height of the lattice of security types and the time to compute `Join` and `Meet`, i.e., $O(p \times h \times \log s)$.

The time complexity of the specialized algorithm for the `type(Z,t)?` query is $O(1)$ if `Z` is a root, an array, a literal, or an `arrlen` expression, and $O(pReach \times h \times \log s)$ in all other cases, where `pReach` stands for the sum of (i) the number of nodes in the syntax tree that are reachable from `Z`, and (ii) the number of nodes that are above `Z` in the syntax tree. This number may be significantly smaller than the size of the whole program.

The query `type(Z,t)?` asks for the type `t` of a specific program node `Z`. The set of rules specific to it, generated by MST, is shown in Figure 5.10.

The query `error(L,E)?`, where `L` is a specific variable and `E` is a specific expression, can be used to check whether a certain assignment command, `L:=E`, causes a type error. The rules generate demand facts for types of certain locations and expressions only. The rules specific to this query, generated by MST, are shown in Figure 5.11. The time complexity of the specialized algorithm for the `error(L,E)?` query is bounded above by the time complexity of computing the `type(Z,t)?` query, as described above.

The query `error(L,e)?` can be used to check whether there is insecure information flow into a specific variable `L` from any expression `e`. The rules generate demand facts for types of certain locations and expressions only. The number of expressions whose type is demanded is thus limited and there may be some saved computation as explained above. The rules specific to this query, generated by MST, are shown in Figure 5.12. The time complexity of the specialized algorithm for the `error(L,e)?` query is bounded above by the time complexity of computing the `type(Z,t)?` query, as described above.

The query `htype(C,t)?` can be used to find the highest possible type `t` of a given command `C`. Some facts of the `type` and `htype` relation are demanded, but the number of demanded facts is limited by the rules and some computation may be saved, e.g., if the variable is only used in one branch of an `if` command or in one subcommand of a `sequence` command. The specialized rules for this query are shown in Figure 5.13. The time complexity of the specialized algorithm for the `htype(C,t)?` query is bounded

```
1.  demand_typebf(c), root(c) → typebf(c,bottom)
2.  demand_typebf(n), literal(n) → typebf(n,bottom)
3.  demand_typebf(l), loc(l),locenv(l,t) → typebf(l,t)
4.  demand_typebf(e), arrlen(e,a), arrenv(a,t1,t2) → typebf(e,t2)
5.  demand_typebf(e), arraccess(e,a,e1),arrenv(a,t1,t2),typebf(e1,t3)
      → typebf(e,Join(t1,t3))
    demand_typebf(e), arraccess(e,a,e1),arrenv(a,t1,t2)
      → demand_typebf(e1)
6.  demand_typebf(e), arith(e,e1,e2),typebf(e1,t1),typebf(e2,t2)
      → typebf(e,Join(t1,t2))
    demand_typebf(e), arith(e,e1,e2)→ demand_typebf(e1)
    demand_typebf(e), arith(e,e1,e2),typebf(e1,t1) → demand_typebf(e2)
7.  demand_typebf(x), assign(c,x,e),id(x),typebf(e,t1),
    typebf(c,t2),typebf(x,t3)
      → typebf(x,Join(t1,t2,t3))
    demand_typebf(x), assign(c,x,e),id(x) → demand_typebf(e)
    demand_typebf(x), assign(c,x,e),id(x),typebf(e,t1)
      → demand_typebf(c)
13. demand_typebf(c1), sequence(c,c1,c2),typebf(c,t) → typebf(c1,t)
    demand_typebf(c1), sequence(c,c1,c2) → demand_typebf(c)
14. demand_typebf(c2), sequence(c,c1,c2),typebf(c,t) → typebf(c2,t)
    demand_typebf(c2), sequence(c,c1,c2) → demand_typebf(c)
15. demand_typebf(c1), if(c,e,c1,c2),typebf(e,t1),typebf(c,t2)
      → typebf(c1,Join(t1,t2))
    demand_typebf(c1), if(c,e,c1,c2) → demand_typebf(e)
    demand_typebf(c1), if(c,e,c1,c2),typebf(e,t1) → demand_typebf(c)
16. demand_typebf(c2), if(c,e,c1,c2),typebf(e,t1),typebf(c,t2)
      → typebf(c2,Join(t1,t2))
    demand_typebf(c2), if(c,e,c1,c2) → demand_typebf(e)
    demand_typebf(c2), if(c,e,c1,c2),typebf(e,t1) → demand_typebf(c)
17. demand_typebf(c1), while(c,e,c1),typebf(e,t1),typebf(c,t2)
      → typebf(c1,Join(t1,t2))
    demand_typebf(c1), while(c,e,c1) → demand_typebf(e)
    demand_typebf(c1), while(c,e,c1),typebf(e,t1)
      → demand_typebf(c)
18. demand_typebf(x), letid(c,x,e,c1),typebf(e,t) → typebf(x,t)
    demand_typebf(x), letid(c,x,e,c1) → demand_typebf(e)
19. demand_typebf(c1), letid(c,x,e,c1),typebf(c,t) → typebf(c1,t)
    demand_typebf(x), letid(c,x,e,c1) → demand_typebf(c)
```

Figure 5.10: Specialized rules for the `type(Z,t)?` query.

above by the time complexity of computing the `type(Z,t)?` query.

In general it is hard to say whether it is worthwhile to use MST for any given query in this application. For any of the above queries MST may or may not lead to a performance

```
8.   demand_errorbb(l,e), assign(c,l,e),loc(l),typebf(l,t1),
     typebf(e,t2),not t2⊆t1
        → errorbb(l,e)
     demand_errorbb(l,e), assign(c,l,e),loc(l) → demand_typebf(l)
     demand_errorbb(l,e), assign(c,l,e),loc(l),typebf(l,t1)
       → demand_typebf(e)
9.   demand_errorbb(l,e), assign(c,l,e),loc(l),typebf(l,t1),
     typebf(c,t2),not t2⊆t1
       → errorbb(l,e)
     demand_errorbb(l,e), assign(c,l,e),loc(l) → demand_typebf(l)
     demand_errorbb(l,e), assign(c,l,e),loc(l),typebf(l,t1)
       → demand_typebf(c)
10.  demand_errorbb(a,e1), arralloc(c,a,e1),arrenv(a,t1,t2),
     typebf(e1,t3),not t3⊆t2
        → errorbb(a,e1)
     demand_errorbb(a,e1), arralloc(c,a,e1),arrenv(a,t1,t2)
       → demand_typebf(e1)
11.  demand_errorbb(a,e1), arrassign(c,a,e1,e2),arrenv(a,t1,t2),
     typebf(e1,t3), not t3⊆t1 → errorbb(a,e1)
     demand_errorbb(a,e1), arrassign(c,a,e1,e2),arrenv(a,t1,t2)
       → demand_typebf(e1)
12.  demand_errorbb(a,e), arrassign(c,a,e1,e2),arrenv(a,t1,t2),
     typebf(e2,t4),not t4⊆t1
       → errorbb(a,e1)
     demand_errorbb(a,e1), arrassign(c,a,e1,e2),arrenv(a,t1,t2)
       → demand_typebf(e1)
```

Figure 5.11: Specialized rules for the `error(L,E)?` query.

gain, and whether it does depends on the specific query and program structure, so it is very difficult to analyze — we need to look at the program AST and analyze this way to predict. This is very time consuming and takes a significant effort. Also, if there is no performance gain and we complete MST anyways, this introduces a lot of extra rules and hypotheses, since there are over 14 rules that infer facts of the type relation and they contain 15 recursive occurrences of the type relation. So, with MST the number of rules involved in the computation more than doubles and the size of the given rules increases. For queries involving the `htype` and `error` hypotheses MST is at least as costly because it causes `bf` annotations of the `type` relation. Thus, the algoritms for computing such queriues require computing the `type(Z,t)?` query as well.

5.6 Related Work and Discussion

A lot of work has been done in optimizing the evaluation of Datalog rules in search of a strategy which combines the advantages of the brute force approach and on-demand

```
8.  demand_errorbf(l),assign(c,l,e),loc(l),typebf(l,t1),
    typebf(e,t2),not t2⊆t1
      → errorbf(l,e)
    demand_errorbf(l),assign(c,l,e),loc(l) → demand_typebf(l)
    demand_errorbf(l),assign(c,l,e),loc(l),typebf(l,t1)
      → demand_typebf(e)
9.  demand_errorbf(l),assign(c,l,e),loc(l),typebf(l,t1),
    typebf(c,t2),not t2⊆t1
      → errorbf(l,e)
    demand_errorbf(l),assign(c,l,e),loc(l) → demand_typebf(l)
    demand_errorbf(l),assign(c,l,e),loc(l),typebf(l,t1)
      → demand_typebf(c)
10. demand_errorbf(a),arralloc(c,a,e1),arrenv(a,t1,t2),
    typebf(e1,t3),not t3⊆t2
      → errorbf(a,e1)
    demand_errorbf(a),arralloc(c,a,e1),arrenv(a,t1,t2)
      → demand_typebf(e1)
11. demand_errorbf(a),arrassign(c,a,e1,e2),arrenv(a,t1,t2),
    typebf(e1,t3),not t3⊆t1
      → errorbf(a,e1)
    demand_errorbf(a),arrassign(c,a,e1,e2),arrenv(a,t1,t2)
      → demand_typebf(e1)
12. demand_errorbf(a),arrassign(c,a,e1,e2),arrenv(a,t1,t2),
    typebf(e2,t4),not t4⊆t1
      → errorbf(a,e1)
    demand_errorbf(a),arrassign(c,a,e1,e2),arrenv(a,t1,t2)
      → demand_typebf(e1)
```

Figure 5.12: Specialized rules for the `error(L,e)?` query.

computation.

The problem with the possibility of infinite loops in on-demand evaluation has been adequately addressed by memoing techniques. Pereira and Warren [65] present a method for memoing top-down evaluation based on Earley deduction — a top-down deduction framework, intended for natural language processing. QSQ [87] is a method by which goals are generated top-down, but whenever possible, goals are propagated in sets at a time, rather than one at a time, and all generated goals and facts are memoized. The extension tables approach [31] is very similar to QSQ, but computation is done tuple at a time. Kifer and Lozinski have developed a method called filtering [50, 51]. It is based on constructing a rule-goal graph. Tuples are propagated along the arcs and computation is restricted by attaching filters to the arcs.

The XSB system [76, 24, 23] uses tabling to prevent infinite loops and to avoid repeated computations. It is the fastest existing system for evaluation of sets of rules, and resolves the major deficiencies of the top-down approach. However, some problems remain.

XSB's efficiency is dependent on the order of hypotheses in the rules. Figure 5.14 shows a comparison between evaluating the query `path(a,b)?` for the graph reachability

20. `demand_htypebf(c),assign(c,x,e),id(x),typebf(x,t)` → `htypebf(c,t)`
    `demand_htypebf(c),assign(c,x,e),id(x)` → `demand_typebf(x)`
21. `demand_htypebf(c),assign(c,l,e),loc(l),typebf(l,t)` → `htypebf(c,t)`
    `demand_htypebf(c),assign(c,l,e),loc(l)` → `demand_typebf(l)`
22. `demand_htypebf(c),arralloc(c,a,e1),arrenv(a,t1,t2)` → `htypebf(c,t2)`
23. `demand_htypebf(c),arrassign(c,a,e1,e2),arrenv(a,t1,t2)`
    → `htypebf(c,t1)`
24. `demand_htypebf(c),sequence(c,c1,c2),htypebf(c1,t1),htypebf(c2,t2)`
    → `htypebf(c,Meet(t1,t2))`
    `demand_htypebf(c),sequence(c,c1,c2)` → `demand_htypebf(c1)`
    `demand_htypebf(c),sequence(c,c1,c2),htypebf(c1,t1)`
    → `demand_htypebf(c2)`
25. `demand_htypebf(c),if(c,e,c1,c2),htypebf(c1,t1),htypebf(c2,t2)`
    → `htypebf(c,Meet(t1,t2))`
    `demand_htypebf(c),if(c,e,c1,c2)` → `demand_htypebf(c1)`
    `demand_htypebf(c),if(c,e,c1,c2),htypebf(c1,t1)` → `demand_htypebf(c2)`
26. `demand_htypebf(c),while(c,e,c1),htypebf(c1,t)` → `htypebf(c,t)`
    `demand_htypebf(c),while(c,e,c1)` → `demand_htypebf(c1)`
27. `demand_htypebf(c),letid(c,x,e,c1),htypebf(c1,t)` → `htypebf(c,t)`
    `demand_htypebf(c),letid(c,x,e,c1)` → `demand_htypebf(c1)`

Figure 5.13: Specialized rules for the `htype(C,t)?` query.

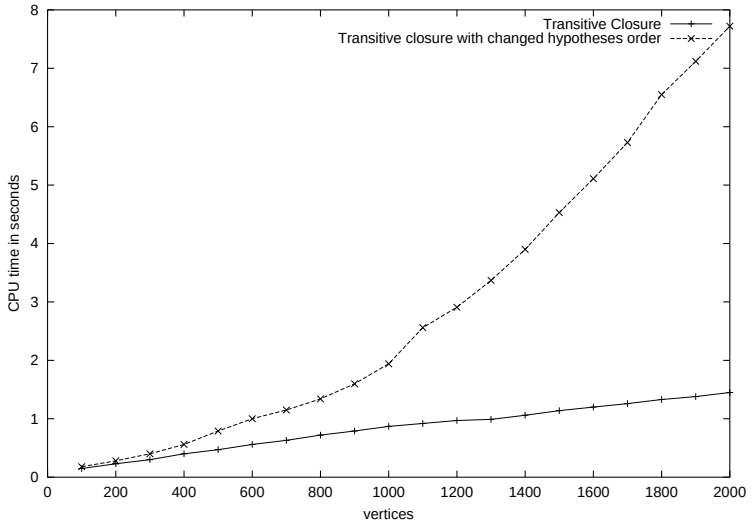


Figure 5.14: A comparison for the query `path(a,b)?` for the graph reachability rules and the same rules with the order of hypotheses changed in the second rule.

rules and the same rules, but with a changed order of the hypotheses in the second rule. For the first set of rules, the CPU time for evaluating the query grows linearly in terms of the number of vertices and edges in the graph, whereas for the second set of rules the growth

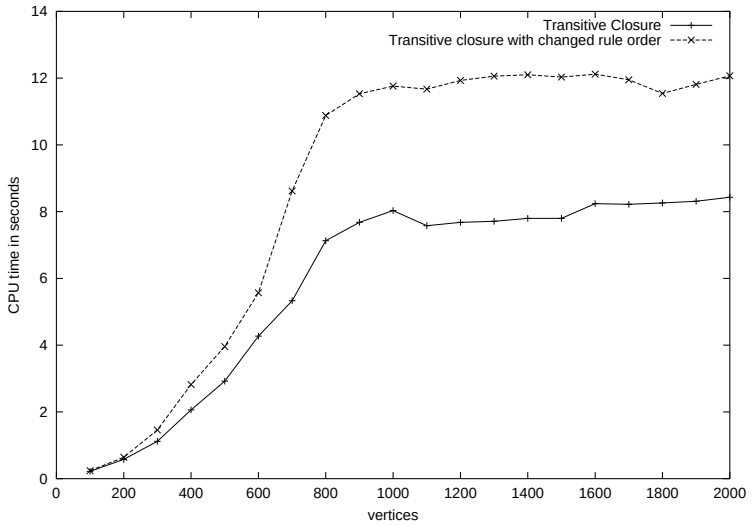


Figure 5.15: A comparison for the query `path(a,v)?` for the graph reachability rules and the same rules but interchanged.

is superlinear. This is a significant difference. The same is observed for the `path(a,v)?` query. For the `path(u,b)?` and `path(u,v)?` queries, the difference is even more significant.

Figure 5.15 shows a comparison between evaluating the query `path(a,v)?` for the graph reachability rules and the same set of rules but with changed rule order. The difference in the evaluation time of the two sets of rules is not asymptotic, but does still exist. The result is very similar for the `path(a,b)?` query. For the `path(u,b)?` and `path(a,b)?` queries there is no notable difference in evaluation time when rule order is changed.

In addition, all of the mentioned strategies based on top-down computation, evaluate tuple-at-a-time, which may be a disadvantage with database applications since it can make disk accesses inefficient [70]. Thus, improvements to bottom-up evaluation remain attractive for database applications.

Several methods exist for improving the efficiency of brute force evaluation. The main thrust for them is to decrease the number of facts generated during the computation. The most prominent among these is Magic Set Transformation, referred to as MST. It was first presented in the paper [11] and developed in [14, 68, 86]. It is a rewriting method, which we use to define a query-specific set of rules. The Alexander method [73] was proposed independently of MST and is essentially the supplementary variant of the MST algorithm. Seki has generalized the method to work with non-ground facts, i.e., facts whose arguments can be variables rather than constants, and function symbols [77].

MST was developed specifically for recursive queries, however, it has been adapted to non-recursive ones and extended to deal with SQL concepts, such as grouping, aggregation and arithmetic conditions [47, 59]. Mumick et. al. [58] present a performance evaluation

of MST over DB2 and concludes that it performs often a lot better than standard DB techniques. With these extensions the MST method has become applicable to a wider range of problems. However, its major drawbacks remained.

The efficiency of the transformated program is dependent on the original order of hypothesis in the rules. The efficiency of MST is dependent on the hypotheses order in the input rules [11, 79, 85]. Using a different order of hypotheses in the original rules lead to passing and using the variable bindings from the query and among hypotheses in different ways. This can cause a difference in orders of magnitude in the complexity of the resulting program. The efficiency of MST on a specific hypotheses order can also be influenced by the given facts [79].

No solution currently exists for choosing the best hypotheses order for MST. Some implementations allow the user to specify an order to be used [69]. This approach requires an understanding of MST and the way certain characteristics of the data influence the efficiency of hypotheses orders. It is time-consuming and error-prone. In some implementations, if a set of rules obtained by MST is worse than the original program, the original is used [60]. This approach fails to make use of good optimization possibilities in many cases.

Our evaluation method addresses these drawbacks. It chooses the best hypotheses order while taking into consideration the specific query. Also, it provides precise time complexity analysis for the resulting sets of rules.

Chapter 6

Conclusion

Our experience shows that many practical problems that may be difficult to understand and implement otherwise can conveniently and intuitively be expressed using Datalog rules. By focusing on translating problem descriptions to precise Datalog rules, the software life-cycle gets rid of the very time consuming and costly phase of implementation. Also, if a change in requirements occurs, it is possible to make small changes in the high-level specifications and automatically generate a new implementation. Because the code is generated automatically it is amenable to an accurate complexity analysis. Our work resulted in clearer and more efficient implementations, and improved time and space complexity bounds, for some of the problems.

Our current work has been a successful step in the direction of improving software productivity, and has led us to several problems which we hope to work on in the short run. We plan to continue to define intuitive high-level specifications and generate automated implementations for specifying and enforcing security policies, such as role-based access control policies, trust management, and access policies for web ontologies, sensor networks, and program analysis. The method involving the MST would be further improved by extending it to amortize the costs of MST for a sequence of queries. Our intentions are to continue to explore possibilities for creative applications of declarative programming and code generation from high-level specifications to tackle practical problems in different areas of computer science.

Bibliography

[1] M. Abadi. Secrecy by typing insecurity protocols. In *TACS '97: Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*, pages 611–638, London, UK, 1997. Springer-Verlag.

[2] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] H. Ait-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst.*, 11(1):115–146, 1989.

[6] S. Ajmani, D. E. Clarke, C.-H. Moh, and S. Richman. ConChord: Cooperative SDSI certificate storage and name resolution. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 141–154, London, UK, 2002. Springer-Verlag.

[7] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 207–220, London, UK, 2001. Springer-Verlag.

[8] M. Avvenuti, C. Bernardeschi, and N. D. Francesco. Java bytecode verification for secure information flow. *SIGPLAN Notices*, 38(12):20–27, 2003.

[9] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[10] J.-P. Ban$\check{a}$tre, C. Bryce, and D. L. Métayer. Compile-time detection of information flow in sequential programs. In *ESORICS '94: Proceedings of the Third European Symposium on Research in Computer Security*, pages 55–73, London, UK, 1994. Springer-Verlag.

[11] S. Y. Bancilhon F., Maier D. and U. J. Magic sets and other strange ways of implementing logic programs. In *Procedeeings of 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, Cambridge*, pages 1–15. ACM Press, New York, 1986.

[12] R. Barbuti, C. Bernardeschi, and N. D. Francesco. Checking security of java bytecode by abstract interpretation. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 229–236, New York, NY, USA, 2002. ACM Press.

[13] S. Basu, K. N. Kumar, L. R. Pokorny, and C. R. Ramakrishnan. Resource-constrained model checking of recursive programs. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 236–250, London, UK, 2002. Springer-Verlag.

[14] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems, San Diego, CA, March 1987*, pages 269–284, 1987.

[15] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991.

[16] M. Benedikt, P. Godefroid, and T. W. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP '01: Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, pages 652–666, London, UK, 2001. Springer-Verlag.

[17] P. A. Bonatti and P. Samarati. Logics for authorization and security. In *Logics for Emerging Applications of Databases*, pages 277–323, 2003.

[18] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[19] O. Burkart, D. Caucal, F. Moller, and B. Steffen. *Verification on infinite structures*. North Holland, 2000.

[20] O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *ICALP '97: Proceedings of the 24th International Colloquium on Automata, Languages and Programming*, pages 419–429, London, UK, 1997. Springer-Verlag.

[21] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[22] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[23] W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[24] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[25] D. E. Clarke, J.-E. Elien, C. M. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[26] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[27] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[28] Z. Deng and G. Smith. Type inference and informative error reporting for secure information flow. In *Proceedings of ACMSE 2006: 44th ACM Southeast Conference*, 2006.

[29] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

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

[31] S. Dietrich. Extension tables: memo relations in logic programming. In *Proceedings of IEEE Symposium on Logic Programming*, pages 264–272, 1987.

[32] A. K. Eamani and A. P. Sistla. Language based policy analysis in a SPKI trust management system. In *NIST: 4th Annual PKI Research and Development Workshop: Multiple Paths to Trust*, pages 162–176, Gaithersburg, MD, 2005.

[33] J. Edmund M. Clark, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[34] C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. Thomas, and T. Ylonen. RFC 2693: SPKI certificate theory, Sept. 1999.

[35] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 232–247, London, UK, 2000. Springer-Verlag.

[36] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 324–336, London, UK, 2001. Springer-Verlag.

[37] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Proceedings of INFINITY '97: the 2nd Internationa Workshop on Verification of Infinite State Systems*, volume 9 of *Electronic Notes in Theoretic Computer Science*. Elsevier, 1997.

[38] N. D. Francesco, A. Santone, and L. Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. *Fundamenta Informaticae*, 54(2-3):195–211, 2003.

[39] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197, New York, NY, USA, 2004. ACM Press.

[40] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys and Tutorials*, 3(4), 2000.

[41] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1-2):105–142, 2001.

[42] J. Y. Halpern and R. van der Meyden. A logical reconstruction of SPKI. *Journal of Computer Security*, 11(4):581–614, 2003.

[43] C. Heitmeyer. Using the scr* toolset to specify software requirements. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 12, Washington, DC, USA, 1998. IEEE Computer Society.

[44] K. Hristova and Y. A. Liu. Improved algorithm complexities for linear temporal logic model checking of push down systems. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *LNCS*, pages 190–206. SV, 2006.

[45] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient type inference for secure information flow. Technical Report DAR 07-35, Computer Science Department, SUNY Stony Brook, May 2007. A preliminary version of this work appeared in *PLAS'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

[46] K. Hristova, K. T. Tekle, and Y. A. Liu. Efficient trust management policy analysis from rules. In *PPDP*, pages 211–220, 2007.

[47] H. P. I. S. Mumick and R. Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the International Conference on Very Large Data Bases*, 1996.

[48] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[49] S. Jha and T. W. Reps. Model checking SPKI/SDSI. *Journal of Computer Security*, 12(3-4):317–353, 2004.

[50] M. Kifer and E. L. Lozinskii. A framework for an efficient implementation of deductive databases. In *Proceedings of the 6-th Advanced Database Symposium Aug. 29-30*, 1986.

[51] M. Kifer and E. L. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Syst.*, 15(3):385–426, 1990.

[52] N. Li. Local names in SPKI/SDSI. In *Proceedings of SCFW: the 13th IEEE Computer Security Foundations Workshop*, page 2, Washington, DC, USA, 2000. IEEE Computer Society.

[53] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of PADL '03: Practical Aspects of Declarative Languages*, pages 58–73, 2003.

[54] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *CSFW '03: Proceedings of IEEE Computer Security Foundations Workshop*, pages 89–, 2003.

[55] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS '01)*, pages 156–165, New York, NY, USA, 2001. ACM Press.

[56] N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 123–139, 2003.

[57] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 172–183. ACM Press, 2003.

[58] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 247–258. ACM Press, 1990.

[59] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. *ACM Transactions on Database Systems*, 21(1):107–155, 1996.

[60] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *Proceedings of ACM SIGMOD, Minneapolis, MN, May 1994*, pages 103–114, 1994.

[61] A. C. Myers. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, New York, NY, USA, 1999. ACM Press.

[62] R. Paige. Real-time simulation of a set machine on a ram. In *Proceedings International Conference on Computing and Information, volume 2*, pages 68–73, 1989.

[63] R. Paige and S. Koenig. Finite differencing of computable expressions. *TOPLAS: ACM Transactions of Programming Languages and Systems*, 4(3):402–454, 1982.

[64] J. Palsberg and P. Ørbæk. Trust in the lambda-calculus. In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, pages 314–329, London, UK, 1995. Springer-Verlag.

[65] F. C. N. Pereira and D. H. D. Warren. Parsing as deduction. In *Proceedings of 21st Annual Meeting of the Association for Computational Linguistics*, jun 1983.

[66] F. Pottier and V. Simonet. Information flow inference for ml. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 319–330, New York, NY, USA, 2002. ACM Press.

[67] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[68] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3 and 4):189–216, 1991.

[69] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. The CORAL deductive system. *VLDB Journal: Very Large Data Bases*, 3(2):161–210, 1994.

[70] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[71] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

[72] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at the Sixteenth Annual Crypto Conference (CRYPTO '96) Rumpsession, 1996.

[73] L. R. Rohmer J. and K. J-M. The alexander method: a technique for the processing of recusive axioms in deductive databases. *New Generation Computing*, 4(3):522–528, 1986.

[74] T. Rothamel, Y. A. Liu, C. L. Heitmeyer, and E. I. Leonard. Generating optimized code from scr specifications. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers and Tool Support for Embedded Systems*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[75] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[76] K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of SIGMOD '94: the 1994 ACM SIGMOD International Conference on Management of Data*, pages 442–453, 1994.

[77] H. Seki. On the power of alexander templates. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 150–159. ACM Press, 1989.

[78] V. Simonet. Flow caml in a nutshell. In *Proceedings of the First APPSEM-II Workshop*, pages 152–165, 2003.

[79] S. Sippu and E. Soisalon-Soininen. An analysis of magic sets and related optimization strategies for logic queries. *J. ACM*, 43(6):1046–1088, 1996.

[80] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 355–364, New York, NY, USA, 1998. ACM Press.

[81] B. Steffen. Generating data flow analysis algorithms from modal specifications. In *TACS'91: Selected Papers of the Conference on Theoretical Aspects of Computer Software*, pages 115–139, Amsterdam, The Netherlands, 1993. Elsevier Science Publishers B. V.

[82] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In *CONCUR '95: Proceedings of the 6th International Conference on Concurrency Theory*, pages 72–87, London, UK, 1995. Springer-Verlag.

[83] M. Sulzmann. A general type inference framework for hindley/milner style systems. In *FLOPS '01: Proceedings of the 5th International Symposium on Functional and Logic Programming*, pages 248–263, London, UK, 2001. Springer-Verlag.

[84] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of the 11th International Static Analysis Symposium, Lecture Notes in Computer Science*, volume 3148, pages 84–99, 2004.

[85] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 140–149. ACM Press, 1989.

[86] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

[87] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In L. Kerschberg, editor, *Proc. First Intl. Conf. on Expert Database Systems*, pages 179–193, 1986.

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

[89] D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.

Appendix A

Appendix

A.1 Pseudocode for inferring the reach graph:

```
//output format in X, after inc, high-level
W := givenFacts;
loc1_2, or23_1, erase12_34 := {};
trans01_23, trans11_234 := {}, gtrans145_123, gtrans1e45_123 := {};
trans21_2345, gtrans245_1236, gtrans2e34_1256, gtrans2e56_1234:= {};
gtrans2ee45_1236, gtrans2ee_or45_123 := {};
R := {};

while exists x in W:

  case x of [trans0 b1 a b2]:
    //update for rule 1
    W +:= {[erase b1 a good b2] : [good] in loc1_2{[b1]} | [erase b1 a good b2] notin R};
    trans01_23 with:= [[b1], [a b2]];

  case x of [trans1 b1 a b3 a1]:
    //update for rule 2
    W +:= {[gtrans1 b1 g2 a b3 a1] : [g2] in loc1_2{[b1]}
                                | [gtrans1 b1 g2 a b3 a1] notin R};
    //update auxiliary map used in rule 2
    trans11_234 with:= [[b1], [a b3 a1]];
```

```
case x of [loc b1 good]:
  //update for rule 1
  W +:= {[erase b1 a good b2] : [a b2] in trans01_23{[b1]}
                              | [erase b1 a good b2] notin R};
  //update for rule 2
  W +:= {[gtrans1 b1 good a b3 a1] : [a b3 a1] in trans11_234{[b1]}
                              | [gtrans1 b1 good a b3 a1] notin R};
  //update for rule 5
  W +:= {[gtrans2 b1 good a b3 a1 a2] : [a b3 a1 a2] in trans21_2345{[b1]}
                              | [gtrans2 b1 good a b3 a1 a2] notin R};
  //update to auxiliary map used in rules 1,2, and 5
  loc1_2 with:= [[b1], [good]];


case x of [or g1 g2 good]:
  // update for rule 4
  W +:= {[erase b1 a good b2] : [b1 a b2] in gtrans1e45_123{[g1 g2]}
                             | [erase b1 a good b2] notin R};

  //update for rule 8
  W +:= {[gtrans2ee_or b1 a b2 g3 good]
                             : [b1 a b2 g3] in gtrans2ee45_1236{[g1 g2]}
                             | [gtrans2ee_or b1 a b2 g3 good] notin R};

  // update for rule 9
  W +:= {[erase b1 a good b2] : [b1 a b2] in gtrans2ee_or45_123{[g2 g1]}
                             | [erase b1 a good b2] notin R};

  //update for rule 12
  W +:= {[edge p1 a1 good pi ai] : [p1 a1 ai pi] in gtrans2e56_1234{[g1 g2]}
                             | [edge p1 a1 good pi ai] notin R};

  //map used in rules 4, 8, 9 and 12
  or23_1 with:= [[g1 g2], [good]];

case x of [gtrans1 p1 good a1 pi ai]:
  //update for rule 3
```

```
  W +:= {[gtrans1e b1 a b2 g1 g2] : [g1 b2] in erase12_34{[b3 a1]}
                              | [gtrans1e b1 a b2 g1 g2] notin R};
  //update auxiliary map used in rule 3
  gtrans145_123 with:= [[b3 a1], [b1 g2 a]];

  //update for rule 10
  if [edge p1 a1 good pi ai] notin R:
    W with := [edge p1 a1 good pi ai];

case x of [trans2 b1 a b3 a1 a2]:
  //update for rule 5
  W +:= {[gtrans2 b1 g3 a b3 a1 a2] : [g3] in loc1_2{[b1]} | [gtrans2 b1 g3 a b3 a1 a2] notin R};
  trans21_2345 with:= [[b1], [a b3 a1 a2]];

case x of [gtrans2 b1 good a1 b2 a2 a3]:
  //update for rule 11
  if [edge b1 a1 good b2 a2] notin R:
     W with := [edge b1 a1 good b2 a2];
  //update for rule 6
  W +:= {[gtrans2e b1 a a2 b4 g1 g3] : [g1 b4] in erase12_34{[b3 a1]}
                             | [gtrans2e b1 a a2 b4 g1 g3] notin R};
  //update to auxiliary map used in rule 6
  gtrans245_1236 with:= [[b3 a1], [b1 g3 a a2]];


case x of [erase b3 a1 g1 b2]:
  //update for rule 3
  W +:= {[gtrans1e b1 a b2 g1 g2] : [b1 g2 a] in gtrans145_123{[b3 a1]}
                              | [gtrans1e b1 a b2 g1 g2] notin R};
  //update for rule 6
  W +:= {[gtrans2e b1 a a2 b2 g1 g3] : [b1 g3 a a2] in gtrans245_1236{[b3 a1]}
                              | [gtrans2e b1 a a2 b2 g1 g3] notin R};
  //update for rule 7
  W +:= {[gtrans2ee b1 a b2 good g1 g3]
                              : [b1 a good g3] in gtrans2e34_1256{[a1 b3]}
                              | [gtrans2ee b1 a b2 good g1 g3] notin R};
  //update to auxiliary map used in rules 3,6 and 7
  erase12_34 with:= [[a1 b3], [g1 b2]];
```

```
case x of [gtrans1e b1 a b2 g1 g2]:
  //update for rule 4
  W +:= {[erase b1 a good b2] : [good] in or23_1{[g1 g2]}
                              | [erase b1 a good b2] notin R};
  //update to auxiliary map used in rule 4
  gtrans1e45_123 with:= [[g1 g2], [b1 a b2]];

case x of [gtrans2e b1 a a2 b4 g1 g3]:
  //update for rule 7
  W +:= {[gtrans2ee b1 a b2 g1 g2 g3]
                              : [g2 b2] in erase12_34{[a2 b4]}
                              | [gtrans2ee b1 a b2 g1 g2 g3] notin R};
  //update to auxiliary map used in rule 7
  gtrans2e34_1256 with:= [[a2 b4], [b1 a g1 g3]];

  //update for rule 12
  W +:= {[edge b1 a1 good b4 a2] : [good] in or23_1{[g1 g3]}
                              | [edge b1 a1 good b4 a2] notin R};
  //update to auxiliary map used in rule 12
  gtrans2e56_1234 with:= [[g1 g3], [b1 a1 a2 b4]];

case x of [gtrans2ee b1 a b2 g1 g2 g3]:
  //update for rule 8
  W +:= {[gtrans2ee_or b1 a b2 g3 gt] : [gt] in or23_1{[g1 g2]}
                              | [gtrans2ee_or b1 a b2 g3 gt] notin R};
  //update to auxiliary map used in rule 8
  gtrans2ee45_1236 with:= [[g1 g2], [b1 a b2 g3]];

case x of [gtrans2ee_or b1 a b2 g3 gt]:
  //update for rule 9
  W +:= {[erase b1 a good b2] : [good] in or23_1{[g3 gt]}
                              | [erase b1 a good b2] notin R};
  //update to auxiliary map used in rule 9
  gtrans2ee_or45_123 with:= [[g3 gt], [b1 a b2]];


W less:= x;
R with:= x;
```

A.2 Pseudocodes for Type Inference for Secure Information Flow

A.2.1 Pseudocode for inferring minimum types of expressions:

```
W := input;
locenv1_2={}; arrlen2_1 := {}; arrenv1_23 := {}; type1_2 := {};
arraccess2_13 := {}; arraccessArrenv2_13 := {};
arith2_13 := {}; arithType2_13 := {};
assign2_13 := {}; assignIdType1_23 := {}; assignIdTypeType2_134 := {};
assignLoc2_13 := {}; assignLocType3_124 := {}; assignLocType1_234 := {};
arralloc2_13 := {}; arrallocArrenv3_124 := {};
arrassign2_134 := {}; arrassignArrenv2_134 := {}; arrassignArrenv1_234 := {};
sequence1_23 := {}; if2_134 := {}; ifType1_234 := {};
while2_13 := {}; whileType1_23 := {}; letid3_124 := {}; letid1_234 := {};
O := {};

while exists x in W:

  case x of [root x]:
    //update for rule 1
    if [type x bottom] not in O:
      W with:= [type x bottom];

  case x of [literal n]:
    //update for rule 2
    if [type n bottom] not in O:
      W with:= [type n bottom];

  case x of [loc l]:
    //update for rule 3
    W U:= {[type l t] : [t] in locenv1_2{[l]} | [type l t] not in O};
    //update for rules 8 and 9
    W U:= {[assignLoc c l e] : [c e] in assign2_13{[l]} | [assignLoc c l e] not in O};

  case x of [locenv l t]:
    //update for rule 3
```

```
    if [loc l] in W and [type l t] not in O:
      W with:= [type l t];
    locenv1_2 with:=[[l],[t]];

  case x of [arrlen e a]:
    //updates for rule 4
    W U:= {[type e t2] : [t1 t2] in arrenv1_23{[a]} | [type e t2] not in O};
    //auxiliary map used for rule 4
    arrlen2_1 with:= [[a], [e]];

  case x of [arrenv a t1 t2]:
    //update for rule 4
    W U:= {[type e t2] : [e] in arrlen2_1{[a]} | [type e t2] not in O};
    //update for rule 5
    W U:= {[arraccessArrenv e e1 t1] : [e e1] in arraccess2_13{[a]} |
                                       [arraccessArrenv e e1 t1] not in O};
    //update for rule 10
    W U:= {[arrallocArrenv c a e1 t1] : [c e1] in arralloc2_13{[a]} |
                                        [arrallocArrenv c a e1 t1] not in O};
    //update for rules 11 and 12
    W U:= {[arrassignArrenv c a e1 t1] : [c e1 e2] in arrassign2_134{[a]} |
                                         [arrassignArrenv c a e1 t1] not in O};
    //update to auxiliary map used for rules 10, 11 and 12
    arrenv1_23 with:= [[a], [t1 t2]];

  case x of [arith e e1 e2]:
    //update for rule 6
    W U:= {[arithType e e2 t1] : [t1] in type1_2{[e1]} | [arithType e e2 t1] not in O};
    //update to auxiliary map used for rule 6
    arith2_13 with:= [[e1], [e e2]];

  case x of [arraccess e a e1]:
    // update for rule 5
    W U:= {[arraccessArrenv e e1 t1] : [t1 t2] in arrenv1_23{[a]} |
                                       [arraccessArrenv e e1 t1] not in O};
    // update to auxiliary map used for rule 5
    arraccess2_13 with:= [[a], [e e1]];

  //arraccessArrenv is an auxiliary relation for the first two hypotheses of rule 5
```

```
case x of [arraccessArrenv e e1 t1]:
  // update for rule 5
  W U:= {[type e t1] : [t3] in type1_2{[e1]} | [type e t1] not in O};
  // update to auxiliary map mapping e1 to e and t1 where arraccessArrenv(e,e1,t1)
  arraccessArrenv2_13 with:= [[e1], [e t1]];

// arithType is an auxiliary relation for the first two hypothese of rule 6
case x of [arithType e e2 t1]:
  // update for rule 6
  W U:= {[type e t1] : [t2] in type1_2{[e2]} | [type e Join(t1,t3)] not in O};
  // update to auxiliary map used for rule 6
  arithType2_13 with:= [[e2], [e t1]];

case x of [assign c var e]:
  // update for rule 7
  W U:= {[assignId c var e] | [assignId c var e] not in O};
  // update for rules 8 and 9
  W U:= {[assignLoc c var e] | [assignLoc c var e] not in O};
  assign2_13 with:= [[var], [c e]];

case x of [Id x]:
  // update for rule 7
  W U:= {[assignId c x e] : [c e] in assign2_13{[x]} | [assignId c x e] not in O};

// assignId is an auxiliary relation for the first two hypotheses of rule 7
case x of [assignId c x e]:
  // update for rule 7
  W U:= {[assignIdType c x t1] : [t1] in type1_2{[e]} | [assignIdType c x t1] not in O};
  //auxiliary map - maps e to c and x where assignId(c,x,e)
  assignId3_12 with:= [[e], [c x]];

// assignIdType is an auxiliary relation for assignId and the third hypothesis of rule 7
case x of [assignIdType c x t1]:
  // update for rule 7
  W U:= {[assignIdTypeType c x t1 t2] : [t2] in type1_2{[c]} | [assignIdTypeType c x t1 t2] not in O};
  //auxiliary map used for rule 7
  assignIdType1_23 with:= [[c], [x t1]];

// assignIdTypeType is an auxiliary relation for assignIdType and the fourth hypothesis of rule 7
```

```
case x of [assignIdTypeType c x t1 t2]:
  // update for rule 7
  W U:= {[type x t1] : [t3] in type1_2{[x]} | [type x Join(t1,t2,t3)] not in O};
  //auxiliary map used for rule 7
  assignIdTypeType2_134 with:= [[x], [c t1 t2]];

// assignLoc is an auxiliary relation for the first two hypotheses of rules 8 and 9
case x of [assignLoc c l e]:
  //update for rules 8 and 0
  W U:= {[assignLocType c l e t1] : [t1] in type1_2{[l]} | [assignLocType c l e t1] not in O};
  assignLoc2_13 with:= [[l], [c e]];

// assignLocType is an auxiliary relation for assingLon and the third hypotheses of rules 8 and 9
case x of [assignLocType c l e t1]:
  // update for rule 8
  W U:= {[error c] : [t1] in type1_2{[c]} | not t2 <= t1, [error c] not in O};
  assignLocType3_124 with:= [[e], [c l t1]];
  // update for rule 9
  W U:= {[error c] : [t2] in type1_2{[c]} | not t2 <= t1, [error c] not in O};
  assignLocType1_234 with:= [[c], [l e t1]];

case x of [arralloc c a e1]:
  //update for rule 10
  W U:= {[arrallocArrenv c a e1 t1] : [t1 t2] in arrenv1_23{[a]} |
                                      [arrallocArrenv c a e1 t1] not in O};
  arralloc2_13 with:= [[a], [c e1]];

// arrallocArrenv is an auxiliary relation for the first two hypotheses of rule 10
case x of [arrallocArrenv c a e1 t1]:
  // update for rule 10
  W U:= {[error c] : [t3] in type1_2{[e1]} | [error c] not in O};
  arrallocArrenv3_124 with:= [[e1], [c a t1]];

case x of [arrassign c a e1 e2]:
  // update for rules 11 and 12
  W U:= {[arrassignArrenv c a e1 t1] : [t1 t2] in arrenv1_23{[a]} |
                                       [arrassignArrenv c a e1 t1] not in O};
  arrassign2_134 with:= [[a], [c e1 e2]];
```

```
// arrassignArrenv is an auxiliary relation for the first two hypotheses of rules 11 and 12
case x of [arrassignArrenv c a e1 t1]:
  // updates for rule 11
  W U:= {[error c] : [t3] in type1_2{[e1]} | not t3 <= t1, [error c] not in O};
  // update auxiliary map for rule 11
  arrassignArrenv2_134 with:= [[e1], [c a t1]];
  // updates for rule 12
  W U:= {[error c] : [t4] in type1_2{[e1]} | not t4 <= t1, [error c] not in O};
  // update auxiliary map for rule 12
  arrassignArrenv1_234 with:= [[e2], [c a t1]];

case x of [sequence c c1 c2]:
  // update for rule 13
  W U:= {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in O};
  // update for rule 12
  W U:= {[type c2 t] : [t] in type1_2{[c]} | [type c2 t] not in O};
  // update auxiliary map for rules 13 and 14
  sequence1_23 with:= [[c], [c1 c2]];

case x of [if c e c1 c2]:
  // update for rules 15 and 16
  W U:= {[ifType c c1 c2 t1] : [t1] in type1_2{[e]} | [ifType c c1 c2 t1] not in O};
  // update auxiliary map for rules 15 and 16
  if2_134 with:= [[e], [c c1 c2]];

// ifType is an auxiliary relation for the first two hypotheses of rules 15 and 16
case x of [ifType c c1 c2 t1]:
  // update for rules 15 and 16
  W U:= {[type c1 t1] : [t2] in type1_2{[c]} | [type c1 Join(t1,t2)] not in O};
  // update auxiliary map for rules 15 and 16
  ifType1_234 with:= [[c], [c1 c2 t1]];

case x of [while c e c1]:
  // update for rule 17
  W U:= {[whileType c c1 t1] : [t1] in type1_2{[e]} | [whileType c c1 t1] not in O};
  // update auxiliary map for rule 17
  while2_13 with:= [[e], [c c1]];

// whileType is an auxiliary relation for the first two hypotheses of rule 17
```

```
case x of [whileType c c1 t1]:
  // update for rule 17
  W U:= {[type c1 t1] : [t2] in type1_2{[c]} | [type c1 Join(t1,t2)] not in O};
  // auxiliary map for rule 17
  whileType1_23 with:= [[c], [c1 t1]];

case x of [letid c x e c1]:
  // update for rule 18
  W U:= {[type x t] : [t] in type1_2{[e]} | [type x t] not in O};
  letid3_124 with:= [[e], [c x c1]];
  // update for rule 19
  W U:= {[type c1 t] : [t] in type1_2{[c]} | [type c1 t] not in O};
  letid1_234 with:= [[c], [x e c1]];

case x of [type node t]:
  // update for auxiliary map used in rules 4 through 19
  type1_2 with:= [[node], [t]];
  //update for rule 4
  W U:= {[arithType e e2 t] : [e e2] in arith2_13{[node]} | [arithType e e2 t] not in O};
  //update for rule 5
  W U:= {[type e t1] : [e t1] in arraccessArrenv2_13{[node]} | [type e Join(t1,t)] not in O};
  //update for rule 6
  W U:= {[type e t1] : [e t1] in arithType2_13{[node]} | [type e Join(t1,t)] not in O};
  //updates for rule 7
  W U:= {[assignIdType c x t] : [c x] in assignId3_12{[node]} | [assignIdType c x t] not in O};
  W U:= {[assignIdTypeType node x t1 t] : [x t1] in assignIdType1_23{[node]} |
                                          [assignIdTypeType node node t1 t] not in O};
  W U:= {[type node t1] : [c t1 t2] in assignIdTypeType2_134{[node]} |
                          [type node Join(t1,t2,t)] not in O};
  // update for rules 8 and 9
  W U:= {[assignLocType1 c node e t] : [c e] in assignLoc2_13{[node]} |
                                       [assignLocType1 c node e t] not in O};
  // update for rule 8
  W U:= {[error c] : [c l t1] in assignLocType3_124{[node]} | not t <= t1, [error c] not in O};
  // update for rule 9
  W U:= {[error c] : [l e t1] in assignLocType1_234{[node]} | [error c] not in O};
  //updates for rules 11 and 12
  W U:= {[error c] : [c a t1] in arrallocArrenv3_124{[node]} | not t <= t2, [error c] not in O};
  W U:= {[error c] : [c a t1] in arrassignArrenv2_134{[node]} | not t <= t1, [error c] not in O};
```

```
    W U:= {[error c] : [c a t1] in arrassignArrenv1_234{[e1]} | not t <= t1, [error c] not in O};
    // update for rule 13
    W U:= {[type c1 t] : [c1 c2] in sequence1_23{[node]} | [type c1 t] not in O};
    // update for rule 14
    W U:= {[type c2 t] : [c1 c2] in sequence1_23{[node]} | [type c2 t] not in O};
    // updates for rules 15 and 16
    W U:= {[ifType1 c c1 c2 t] : [c c1 c2] in if2_134{[node]} | [ifType1 c c1 c2 t] not in O};
    W U:= {[type c1 t1] : [c1 c2 t1] in ifType1_234{[node]} | [type c1 Join(t1,t)] not in O};
    W U:= {[type c2 t1] : [c1 c2 t1] in ifType1_234{[node]} | [type c2 Join(t1,t)] not in O};
    //updates for rule 17
    W U:= {[whileType1 c c1 t] : [c c1] in while2_13{[node]} | [whileType1 c c1 t] not in O};
    W U:= {[type c1 t1] : [c1 t1] in whileType1_23{[node]} | [type c1 Join(t1,t)] not in O};
    //updates for rules 18 and 19
    W U:= {[type x t] : [c x c1] in letid3_124{[node]} | [type x t] not in O};
    W U:= {[type c1 t] : [x e c1] in letid3_124{[node]} | [type c1 t] not in O};

  W less:= x;
  if x not in input:
    O with:= x;
```

A.2.2 Pseudocode for inferring maximum types of commands:

```
assign2_13 := {}; assignId2_13 := {}; assignLoc2_13 := {};
arralloc2_13 := {}; arrenv1_23 := {}; arrassign2_134 := {};
sequence2_13 := {}; sequenceHtype1_23 := {}; if3_124 := {}; ifHtype3_124 := {};
while3_12 := {}; letid4_123 := {}; htype1_2 := {}; type1_2 := {};
W:= input U MIN;
O' := {};

while exists x in W:

  case x of [assign c var e]:
    //update for rule 20
    W U:= {[assignId c var e] | [assignId c var e] not in O'};
    //update for rule 21
    W U:= {[assignLoc c var e] | [assignLoc c var e] not in O'};
    //update to auxiliary map used in rules 20 and 21
    assign2_13 with:= [[var], [c e]];
```

```
case x of [id x]:
  //update for rule 20
  W U:= {[assignId c x e] : [c e] in assign2_13{[x]} | [assignId c x e] not in O'};

//assignId is an auxiliary relation for the first two hypotheses of rule 20
case x of [assignId c x e]:
  //update for rule 20
  W U:= {[htype c t] : [t] in type1_2{[x]} | [htype c t] not in O'};
  //update to auxiliary map used in rule 20
  assignId2_13 with:= [[x], [c e]];

case x of [loc l]:
  //update for rule 21
  W U:= {[assignLoc c l e] : [c e] in assign2_13{[l]} | [assignLoc c l e] not in O'};

//assignLoc is an auxiliary relation for the first two hypotheses of rule 21
case x of [assignLoc c l e]:
  //update for rule 21
  W U:= {[htype c t] : [t] in r31typelt{[l]} | [htype c t] not in O'};
  //update to auxiliary map used in rule 21
  assignLoc2_13 with:= [[l], [c e]];

case x of [type node t]:
  //update for rule 20
  W U:= {[htype c t] : [c e] in assignId2_13{[node]} | [htype c t] not in O'};
  //update for rule 21
  W U:= {[htype c t] : [c e] in assignLoc2_13{[node]} | [htype c t] not in O'};
  //update to auxiliary map used in rules 20 and 21
  type1_2 with:= [[node], [t]];

case x of [arralloc c a e1]:
  // update for rule 22
  W U:= {[htype c t2] : [t1 t2] in arrenv1_23{[a]} | [htype c t2] not in O'};
  // update to auxiliary map used in rule 22
  arralloc2_13 with:= [[a], [c e1]];

case x of [arrenv a t1 t2]:
  // update for rule 22
```

```
  W U:= {[htype c t2] : [c e1] in arralloc2_13{[a]} | [htype c t2] not in O'};
  //update for rule 23
  W U:= {[htype c t1] : [c e1 e2] in arrassign2_134{[a]} | [htype c t1] not in O'};
  //update to auxiliary map used in rules 22 and 23
  arrenv1_23 with:= [[a], [t1 t2]];

case x of [arrassign c a e1 e2]:
  //update for rule 23
  W U:= {[htype c t1] : [t1 t2] in arrenv1_23{[a]} | [htype c t1] not in O'};
  //update to auxiliary map used in rule 23
  arrassign2_134 with:= [[a], [c e1 e2]];

case x of [sequence c c1 c2]:
  //update for rule 24
  W U:= {[sequenceHtype c c2 t1] : [t1] in htype1_2{[c1]} | [sequenceHtype c c2 t1] not in O'};
  //update to auxiliary map used in rule 24
  sequence2_13 with:= [[c1], [c c2]];

//sequenceHtype is an auxiliary relation for the first two hypotheses of rule 24
case x of [sequenceHtype c c1 t1]:
  //update for rule 24
  W U:= {[htype c t1] : [c2 t2] in htype1_2{[c1]} | [htype c Meet(t1,t2)] not in O'};
  //update to auxiliary map used in rule 24
  sequenceHtype1_23 with:= [[c1], [c t1]];

case x of [if c e c1 c2]:
  //update for rule 25
  W U:= {[ifHtype c e c2 t1] : [t1] in htype1_2{[c1]} | [ifHtype c e c2 t1] not in O'};
  //update to auxiliary map used in rule 25
  if3_124 with:= [[c1], [c e c2]];

//ifHtype is an auxiliary relation for the first two hypotheses of rule 25
case x of [ifHtype c e c2 t1]:
  //update for rule 25
  W U:= {[htype c t1] : [t2] in htype1_2{[c2]} | [htype c Meet(t1,t2)] not in O'};
  //update to auxiliary map used in rule 25
  ifHtype3_124 with:= [[c2], [c e t1]];

case x of [while c e c1]:
```

```
  //update for rule 26
  W U:= {[htype c t] : [t] in htype1_2{[c1]} | [htype c t] not in O'};
  //update to auxiliary map used in rule 26
  while3_12 with:= [[c1], [c e]];

case x of [letid c e x c1]:
  //update for rule 27
  W U:= {[htype c t] : [t] in htype1_2{[c1]} | [htype c t] not in O'};
  //update to auxiliary map used in rule 27
  letid4_123 with:= [[c1], [c e x]];

case x of [htype node t]:
  //updates for rule 24
  W U:= {[sequenceHtype c c2 t] : [c c2] in sequence2_13{[node]}
                     | [sequenceHtype c c2 t] not in O'};
  W U:= {[htype c t1] : [c t1] in sequenceHtype1_23{[node]} | [htype c Meet(t1,t)] not in O'};
  //updates for rule 25
  W U:= {[ifHtype c e c2 t] : [c e c2] in if3_124{[node]} | [ifHtype c e c2 t] not in O'};
  W U:= {[htype c t1] : [c e t1] in ifHtype3_124{[node]} | [htype c Meet(t1,t)] not in O'};
  //update for rule 26
  W U:= {[htype c t] : [c e] in while3_12{[node]} | [htype c t] not in O'};
  //update for rule 27
  W U:= {[htype c t] : [c e x] in letid4_123{[node]} | [htype c t] not in O'};
  //update to auxiliary map used in rules 24 through 27
  htype1_2 with:= [[node], [t]];

W less:= x;
if x not in input:
   O' with:= x;
```