Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Mining Meaningful Role-Based and Attribute-Based Access Control Policies

A Dissertation Presented

by

Zhongyuan Xu

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2014

Stony Brook University

The Graduate School

Zhongyuan Xu

We, the dissertation committee for the above candidate for

the degree of Doctor of Philosophy,

hereby recommend the acceptance of this dissertation.

Scott D. Stoller – Dissertation Advisor

Professor, Department of Computer Science

I. V. Ramakrishnan – Chairperson of Defense

Professor, Department of Computer Science

Rob Johnson – Committee Member

Assistant Professor, Department of Computer Science

Ian M. Molloy – External Committee Member

Information Security Group, IBM T. J. Watson Research Center

This dissertation is accepted by the Graduate School.

Charles Taber

Dean of the Graduate School

Abstract of the Dissertation

Mining Meaningful Role-Based and Attribute-Based Access Control Policies

by
Zhongyuan Xu

Doctor of Philosophy
in
Computer Science
Stony Brook University
2014

Advanced models of access control, such as role-based access control (RBAC) and attribute-based access control (ABAC), offer important advantages over lower-level access control policy representations, such as access control lists (ACLs). However, the effort required for a large organization to migrate from ACLs to RBAC or ABAC can be a major obstacle to adoption of RBAC or ABAC. Policy mining algorithms partially automate the construction of advanced access control policies from ACL policies and possibly other information, such as user and resource attributes. These algorithms can greatly reduce the cost of migration to RBAC or ABAC. This dissertation presents several new policy mining algorithms.

First, this dissertation considers mining of role-based policies from ACL policies and possibly other information. The dissertation presents new and flexible algorithms for this problem. The algorithms can easily be used to optimize a variety of RBAC policy quality metrics, including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability. In experiments with publicly available access control policies, one of our algorithms achieves significantly better results than previous work.

Next, this dissertation considers mining of parameterized role-based policies. Parameterization significantly enhances the scalability of RBAC, by allowing more concise policies. This dissertation defines a parameterized RBAC (PRBAC) framework, in which users and permissions have attributes that are implicit parameters of roles and can be used in role definitions. Algorithms are presented for mining PRBAC policies from ACLs and attribute data. To the best of our knowledge, this is the first PRBAC policy mining algorithm. Evaluation on three small but non-trivial case studies demonstrates the effectiveness of our algorithm.

Finally, this dissertation considers mining of attribute-based policies. ABAC allows policies to be written in a concise, flexible, and high-level way. Three versions of the ABAC policy mining problem are considered, differing in the input: (1) mining ABAC policies from ACLs and attribute data, (2) mining ABAC policies from RBAC policies and attribute data, and (3) mining ABAC policies from operation logs and attribute data. Algorithms are presented for all three versions of the problem. Extensions of the algorithms to identify suspected noise in the input data are also described. To the best of our knowledge, these are the first ABAC policy mining algorithms. Evaluations on sample policies and synthetic policies demonstrate the effectiveness of our algorithms.

Contents

List of Figures vii

List of Tables x

1 Introduction 1
1.1 Motivation . . . 1
1.2 Research Contributions . . . 4

2 Mining Meaningful Roles 8
2.1 Problem Definition . . . 8
2.2 Algorithms . . . 10
2.2.1 Elimination Algorithm . . . 10
2.2.2 Selection Algorithm . . . 15
2.2.3 Complete Algorithm . . . 15
2.3 Datasets . . . 17
2.4 Experimental Results . . . 19
2.5 Related Work . . . 23

3 Mining Parameterized Role-Based Policies 26
3.1 Parameterized RBAC (PRBAC) . . . 26
3.2 The Problem . . . 28
3.3 Algorithms . . . 29
3.3.1 Mining Core PRBAC Policies: Elimination Algorithm . . . 29
3.3.2 Mining Core PRBAC Policies: Selection Algorithm . . . 33
3.3.3 Mining Hierarchical PRBAC Policies: Elimination Algorithm . . . 34
3.3.4 Mining Hierarchical PRBAC Policies: Selection Algorithm . . . 36
3.3.5 Complexity Analysis . . . 36
3.4 Case Studies . . . 38
3.5 Evaluation . . . 41
3.6 Related Work . . . 44

4 Mining Attribute-based Access Control Policies from ACLs 46
4.1 ABAC policy language . . . 46
4.2 The ABAC Policy Mining Problem . . . 48
4.3 Policy Mining Algorithm . . . 50

4.3.1 Noise Detection 55
4.4 Evaluation 56
4.4.1 Evaluation on Sample Policies 56
4.4.2 Evaluation on Synthetic Policies 58
4.4.3 Generalization 62
4.4.4 Noise 62
4.4.5 Comparison with Inductive Logic Programming 65
4.5 Related Work 65

5 Mining Attribute-Based Access Control Policies from Role-Based Policies 71
5.1 Problem Definition 71
5.2 Policy Mining Algorithm 73
5.3 Evaluation 77
5.3.1 Experiments with Full Attribute Data 77
5.3.2 Experiments with Incomplete Attribute Data 80
5.3.3 Experiments with Varying Policy Structure 81
5.4 Related Work 81

6 Mining Attribute-Based Access Control Policies from Logs 84
6.1 Problem Definition 84
6.2 Algorithm 85
6.2.1 Example 87
6.3 Algorithm Based on Generative Model 90
6.4 Evaluation Methodology 92
6.4.1 ABAC Policies 92
6.4.2 Log Generation 95
6.4.3 Metrics 95
6.5 Experimental Results 96
6.6 Related Work 98

7 Future Work 100

Appendices 103

A Supplemental Material on ABAC Mining from ACLs 104
A.1 Proof of NP-Hardness 104
A.2 Asymptotic Running Time 105
A.3 Processing Order 106
A.4 Optimizations 106
A.5 Details of Sample Policies 107
A.6 Example: Processing of a user-permission tuple 112
A.7 Syntactic Similarity 113
A.8 ROC Curves for Noise Detection Parameters 114
A.9 Graphs of Results from Experiments with Permission Noise and Attribute Noise 115
A.10 Translation to Inductive Logic Programming 115

B Supplemental Material on ABAC Mining from Logs **119**
B.1 Rule Quality Metric Based On Inductive Logic Programming 119

Bibliography **121**

List of Figures

2.1 Role generation, step 1: compute candidate roles. 11
2.2 Role generation, step 2: construct role hierarchy, based on R and PA from step 1. 12
2.3 Role elimination. 14
2.4 Restore role r to policy π. 16
2.5 Create direct user-permission assignment. 16
2.6 Information about datasets. N_a is the number of attributes. AF is the attribute fit. 17
2.7 Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is prohibited. 20
2.8 Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is permitted. 20
2.9 Comparison of elimination algorithm and Attribute Miner (AM). Names of datasets are abbreviated, e.g., fw1 abbreviates “firewall-1”. The upper and lower graphs use the high-fit and low-fit user attribute data, respectively. . . 22
2.10 Results for elimination algorithm and selection algorithm, with policy quality metric INT-WSC. The clusters of points correspond, from left to right in the order they are connected, to the datasets in the following order: firewall-2 healthcare, domino, firewall-1, emea, americas-small, apj. 22
2.11 Comparison of two different policy quality metrics in elimination algorithm. “rdn” and “max” denote $\langle \text{redun}, \text{clsSz} \rangle$ and $\langle \max(\text{attrFit}, \text{clsSz}), \text{redun} \rangle$, respectively. 23

3.1 Algorithm to compute minExpU(s), where s is a set of users, and U is the set of all users. $f[x \mapsto y]$ denotes (a copy of) function f modified so that $f(x) = y$. $f_\emptyset$ denotes the empty function, i.e., the function whose domain is the empty set. 31
3.2 Step 4 (Merge Candidate Roles) of elimination algorithm for core PRBAC policy mining. 32
3.3 Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for core PRBAC policy mining. 34
3.4 Algorithm for $\text{minExpU}_\text{H}(r, U, R, RH)$. 37

3.5 Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for hierarchical PRBAC policy mining. 38
3.6 Step 6 (Select Roles) of selection algorithm for hierarchical PRBAC policy mining. 39
3.7 University case study 40
3.8 Healthcare case study 42
3.9 Engineering department case study 43
3.10 Running times and size metrics for case studies. 43

4.1 Policy mining algorithm. 52
4.2 Compute a candidate rule ρ' and add ρ' to candidate rule set *Rules* 52
4.3 Generalize rule ρ by adding some formulas from cc to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i. 53
4.4 Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$. 54
4.5 Sizes of the sample policies. "Type" indicates whether the attribute data in the policy is manually written ("man") or synthetic ("syn"). N is the number of departments for the university and project management sample policies, and the number of wards for the health care sample policy. $\widehat{[\![\rho]\!]}$ is the average number of user-permission tuples that satisfy each rule. An empty cell indicates the same value as the cell above it. 57
4.6 Running time (log scale) of the algorithm on synthetic attribute datasets for sample policies. The horizontal axis is N_{dept} for university and project management sample policies and N_{ward} for health care sample policy. 59
4.7 Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level. Curve names ending with _o and _u are for over-assignments and under-assignments, respectively. The curves for University_u and Synthetic_o are nearly the same and overlap each other. 64
4.8 Semantic similarity of the original policy and the mined policy, as a function of permission noise level. 64

5.1 Left: Top-level pseudocode for policy mining algorithm. Right: computeUAE(s) computes a user-attribute expression that characterizes set s of users. . . . 74
5.2 Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$. 76
5.3 Functions used to simplify rules. 78

5.4 Functions used to simplify rules (continued) and function useRoleAttribute. . 79
5.5 Department/university server example. 81
5.6 Gradebook permissions in university case study, version 1. 82
5.7 Gradebook permissions in university case study, version 2. 82

6.1 Policy mining algorithm. The pseudocode starts in column 1 and continues in column 2. 88
6.2 Compute a candidate rule ρ' and add ρ' to candidate rule set *Rules* 88
6.3 Left: Generalize rule ρ by adding some formulas from cc to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i. Right: Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$. . 89
6.4 Discretization algorithm. random(x, y) returns a random number in the range $[x, y)$. randomInt(i, j) returns a random integer in the range $[i, j]$. $t[i \to v]$ denotes the tuple obtained from tuple t by changing the value of the i'th component to v. 93
6.5 Construct an ABAC policy based on an author-topic assignment and a topic-word assignment. 94
6.6 Top: Syntactic similarity and semantic similarity of original and mined ABAC policies, as a function of log completeness. Bottom: Fractions of over-assignments and under-assignments in mined ABAC policy, as a function of log completeness. 98

A.1 University case study. 108
A.2 Health care case study. 110
A.3 Project management case study. 111
A.4 Online video case study. 112
A.5 Diagram representing the processing of one user-permission tuple selected as a seed, in the university sample policy. Rules are depicted as rectangles with four compartments, corresponding to the four components of a rule tuple. . . 113
A.6 ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on α and τ. 115
A.7 ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for over-assignments on α and τ. 115
A.8 Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level due to permission noise and attribute noise. 116
A.9 Semantic similarity of the original policy and the mined policy, as a function of permission noise level due to permission noise and attribute noise. 116

List of Tables

4.1 Experimental results for synthetic policies with varying $N_{\text{cnj}}^{\text{min}}$. "Synt. Sim." is syntactic similarity. "Compression" is the compression factor. μ is mean, σ is standard deviation, and CI is half-width of 95% confidence interval using Student's t-distribution. An empty cell indicates the same value as the cell above it. 68
4.2 Experimental results for synthetic policies with varying $N_{\text{cns}}^{\text{min}}$. 69
4.3 Experimental results for synthetic policies with varying P_{over}. 70

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Scott D. Stoller, for his support and guidance in my study and research. It is my honor to be one of his Ph.D. students. His patience, knowledge and enthusiasm inspired and motivated me through my entire Ph.D. study. I could not imagine any of my research accomplishments could have been done without his help. I am also thankful for the excellent example he has set up as a successful scholar.

My sincere thanks also goes to Prof. I.V. Ramakrishnan, Prof. Rob Johnson and Dr. Ian M. Molloy, for their suggestions and feedback that helped greatly in improving this dissertation.

I would also like to thank my labmates and friends, who made my Ph.D. study so joyful: Puneet Gupta, Bo Lin, Jon Brandvein, Xuetian Weng, Xiang Gao, and Yifan Peng.

Last, but not least, I would like to thank my family: my parents Jianhua Xu and Xuelian Zhu, for giving birth to me and teaching me the values I live by.

Chapter 1

Introduction

1.1 Motivation

Access control addresses the challenges of protecting resources and data maintained by a computer system against unauthorized accesses while ensuring their availability for authorized accesses. A variety of access control models have been proposed in the literature and applied to numerous application domains. Advanced models of access control, such as *role-based access control* (RBAC), *parameterized role-based access control* (PRBAC), and *attribute-based access control* (ABAC), offer important advantages, such as smaller policy size and higher interpretability, over lower-level access control policy representations, such as the widely used *access control lists* (ACLs).

However, the effort required for a large organization to migrate from ACLs to an advanced access control model can be a major obstacle to adoption of an advanced access control model. Policy mining algorithms partially automate the construction of advanced access control policies from existing information, such as ACL policies, access logs, and user and resource attributes. These algorithms can greatly reduce the cost of migration.

There are many interesting and practical problems in the research area of access control policy mining. An access control policy mining problem can be characterized mainly using four dimensions: the target policy language, the source of the user-permission data, the availability of attribute information, and the presence of noise. We briefly discuss each of these dimensions in turn.

The most popular target policy language in access control policy mining is RBAC. Mining RBAC policies is also called "role mining." PRBAC and ABAC are also attractive target policy languages, because they are usually more concise and high-level than ACLs and RBAC.

The most commonly considered source of user-permission data is ACLs. When the target language is ABAC, another potential source of user-permission data is an existing RBAC policy. In some cases, user-permission data is not available as a declarative policy, for example, if the policy is encoded in a program. In such cases, user-permission data may be available from operation logs, also called "traces", which record the history of users exercising permissions. Algorithms to mine policies from logs must take into account that logs generally provide incomplete information about permissions. An advantage of mining policies from operation logs is that logs reflect the usage of entitlements, i.e. the frequencies of users

exercising permissions. This information can help guide policy mining. For example, users with the same entitlements but different usage can be placed in different roles. A specific example of this is that users who infrequently function as backup administrators, when the primary administrators are unavailable, can be placed in a different role than the primary administrators [MPC12a].

Attribute information usually consists of user attribute information (such as job title, department, and areas of expertise) and resource attribute information (such as location, owner, and security level). Attribute information can be used to determine the interpretability of RBAC policies and is essential for mining Parameterized RBAC policies and ABAC policies.

Another dimension is whether the given data is clean (i.e., accurate) or noisy (i.e., contains errors). Noisy data is common in practice. For example, if a user changed job positions, but the security administrator forgot to revoke the user's permissions corresponding to his old job position, then those permissions are considered to be noise. This dimension has less impact on the design of policy mining algorithms than the other dimensions, but it is still important.

Among these dimensions, the first dimension—target policy language—has, in our experience, the greatest impact on the design of the policy mining algorithm. Therefore, we categorize policy mining problems based on the target policy language, and within each category, we discuss variants of the problem based on the other dimensions.

RBAC Policy Mining In this category, we consider policy mining problems where the target policy language is RBAC. In RBAC, permissions are assigned to roles instead of directly assigned to users. Users acquire permissions through role membership. Therefore, the management of user privileges is simplified primarily to the assignment of appropriate roles to users. This is significantly easier than assigning permissions to users, especially when adding a user to an organization or changing a user's position within an organization. RBAC dramatically decreases policy administration effort, by reducing the number of relationships that need to be maintained. Suppose there are n users and m permissions. With direct user-permission assignment (e.g., ACLs), $O(n * m)$ relationships need to be maintained. If the same user-permission assignment is expressed as an RBAC policy with k roles, then $O(k * n + k * m)$ relations need to be maintained. In a typical RBAC policy, the number of roles is much smaller than the number of users or permissions, which implies $O(k*n+k*m)$ is much smaller than $O(n * m)$.

Several versions of the RBAC policy mining problem have been proposed. In the most studied version, user-permission data is available from ACLs, and attribute information is unavailable. The problem usually involves finding an RBAC policy that is consistent with (i.e., grants the same permissions as) given ACLs and minimizes some measure of policy size. The policy size metric might be the number of roles, the number of edges (i.e., the sum of the number of tuples in the user-role assignment and the permission-role assignment), or more generally a linear combination of the sizes of the components of the RBAC policy. When user and resource attribute data are unavailable to guide policy mining, minimizing policy size is a reasonable criterion for finding good descriptive roles, because a major advantage of RBAC is reduced management effort, and a small RBAC policy is easier to manage than a large one.

However, the resulting roles might be difficult for administrators to understand in organizational terms. In other words, the roles might have low interpretability, also called "semantic meaning". Interpretability is crucial, because typically, a role produced by a role mining algorithm will be adopted by security administrators only if they can identify a reasonable interpretation of the role, in which case the role is said to be "meaningful". Indeed, researchers at HP Labs wrote that "the biggest barrier we have encountered to getting the results of role mining to be used in practice" is that "customers are unwilling to deploy roles that they can't understand." [EHM+08].

Interpretability is addressed in another important version of the problem, in which user attribute data is available, in addition to ACLs, and the goal is to find an RBAC policy that is consistent with the ACLs, has small size, and has high interpretability with respect to the attribute data. Informally, a role has high interpretability (i.e., is meaningful) if the role's membership (i.e., the set of users in the role) can be characterized accurately by an expression involving user attributes. Similarly, if permissions have attributes, interpretability of the set of permissions granted to each role can also be used to help identify meaningful roles.

Parameterized RBAC Policy Mining In this category, we consider policy mining problems where the target policy language is a form of Parameterized RBAC (PRBAC). Allowing roles to have parameters significantly enhances the scalability of RBAC, by allowing much more concise policies. For example, consider a policy for a university. To grant different permissions to users (*e.g.*, faculty or students) in different classes or departments, in an RBAC model without parameters, a separate role and corresponding permission assignments needs to be created for each course or department, leading to a large and unwieldy policy. In a parameterized RBAC model, this policy can be expressed using a few policy statements parameterized by the class identifier or department name.

To the best of our knowledge, we are the first to study PRBAC policy mining. We consider mining PRBAC policies from ACLs when user and resource attribute information are available. The availability of user and resource attribute information is essential, because user and resource attributes are used to parameterize roles. Similar as for RBAC policy mining, the problem is to find a PRBAC policy that is consistent with given ACLs and has small size. In our formulation of the problem, only parameterized roles whose members and permissions can be characterized using attributes are considered, so explicit maximization of interpretability is unnecessary.

ABAC Policy Mining In this category, we consider mining of ABAC policies. ABAC allows policies to be written in a concise, flexible, high-level way. Compared to ACLs, RBAC, and PRBAC, ABAC can reduce management effort and management errors by reducing the number of relationships that need to be maintained. For example, consider the policy "A user working on a project can read and request to work on a non-proprietary task whose required areas of expertise are among his/her areas of expertise." In ABAC, this policy can be expressed with a single rule, regardless of the number of users, projects, and tasks. With ACLs, a separate entry is needed for each combination of a user and a task for which the user has permissions. In RBAC, this policy requires creation of a role for each task, creation of user-role and permission-role assignments for each of those roles, and updates

to those assignments when relevant attribute values change (e.g., a user gains an area of expertise). In PRBAC, a separate role is still needed for each task, because use of attributes as role parameters allows only equalities between user attributes and resource attributes to be expressed; it does not allow set relationships, such as the subset relationship used in this example, to be expressed. The ACL, RBAC, and PRBAC policies are significantly larger, require significantly more management effort, and are more prone to management errors than the ABAC policy.

Several ABAC frameworks have been proposed, varying in administrative model, flexibility, and expressiveness. However, to be best of our knowledge, we are the first to study ABAC policy mining. We consider three versions of the problem.

We first consider mining ABAC policies from ACLs and attribute data. The ABAC policy language that we consider contains most of the common ABAC policy language constructs and is significantly more complex than policy languages (such as RBAC and PRBAC) handled in previous work on security policy mining.

Second, we consider mining ABAC policies from RBAC policies and attribute data. In addition to the usual requirement of minimizing policy size, we require that some aspects of the structure of the RBAC policy be preserved in the ABAC policy, because the structure of the RBAC policy may reflect expert design decisions by the policy author.

Third, we consider mining ABAC policies from operation logs and attribute data. The major challenge for policy mining from operation logs is that logs provide only a lower bound on the granted privileges, not an exact characterization of them. Therefore, the resulting policy should be allowed to include overassignments, i.e., user-permission tuples that are not reflected in the logs, in a reasonable way.

1.2 Research Contributions

This section summarizes our research contributions on each of the above problems.

RBAC Policy Mining In Chapter 2, we developed RBAC policy mining algorithms that can easily be used to optimize a variety of policy quality metrics, including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability.

All of the algorithms begin with a phase that constructs a set of candidate roles. We consider two strategies for the second phase: start with an empty policy and repeatedly add candidate roles, or start with the entire set of candidate roles and repeatedly remove (eliminate) roles. In experiments with publicly available access control policies, we found that the elimination approach produces better results, and that, for a previously proposed policy quality metric that reflects size and interpretability, our elimination algorithm achieves significantly better results than previous work that aims to optimize that metric, even though our algorithm is not specifically tuned for that metric. We also investigated the effect of varying the order in which roles are considered for removal in the elimination algorithm. Due to the lack of publicly available real user attribute data, we developed an algorithm for generating synthetic user attribute data, and we use it in our experiments.

PRBAC Policy Mining In Chapter 3, we start our contributions on PRBAC policy mining with the definition of an expressive PRBAC framework that supports a simple form of ABAC. In our PRBAC framework, (1) users and permissions have attributes that are implicit parameters of roles, (2) the set of users assigned to a role is specified by an expression over user attributes, and (3) the set of permissions granted to a role is specified by an expression over permission attributes. We make role parameters implicit, rather than explicit, because it makes the framework and algorithms slightly simpler; our approach can easily be adapted to handle roles with explicit parameters. Every user and permission has an "id" attribute containing a unique name, so specifying the users and permissions associated with a role by enumeration, as in traditional RBAC, is a simple case of (2) and (3), respectively.

We developed two algorithms for mining PRBAC policies from ACLs, user attributes, and permission attributes. To the best of our knowledge, these are the first policy mining algorithms for any parameterized RBAC framework. At a high level, both algorithms work as follows. First, a conventional role mining algorithm is used to generate a set of candidate roles; attributes and parameterization are not considered in this step. For a policy like the example policy in Section 1.1, this step would produce a separate role granting appropriate permissions to the users of each course or department. Second, the algorithm attempts to form parameterized roles by merging sets of candidate roles from the first step; the resulting parameterized roles are added to the set of candidate roles. Continuing the example, this step would form a parameterized role from the set of roles containing the role for each course or department. Third, the algorithm decides which of the candidate roles generated in the first two steps to include in the final policy. Similar to our RBAC policy mining approaches, we consider two strategies for this. The *elimination* strategy repeatedly removes low-quality roles from the set of candidate roles, until no more roles can be removed without losing some of the permissions granted in the given ACL policy. The *selection* strategy repeatedly selects the highest-quality candidate role for inclusion in the PRBAC policy, until all permissions granted in the given ACL policy are granted by the PRBAC policy. For each of these two algorithms, we first present a simpler version that does not consider role hierarchy, and then present a version that generates hierarchical policies.

To evaluate whether these algorithms can successfully generate meaningful parameterized roles, we wrote three small but non-trivial PRBAC policies, generated ACL policies and attribute data from them, ran our algorithms on the resulting ACL policies and attribute data, and compared the mined PRBAC policies with the original policies. One of our algorithms successfully reconstructs the original PRBAC policies for all three case studies.

ABAC Policy Mining In Chapter 4, we start our contributions on ABAC policy mining with the definition of an expressive ABAC framework. Our ABAC framework is similar to our PRBAC framework, but supports a richer form of ABAC. Most importantly, our ABAC framework supports multi-valued (also called "set-valued") attributes and allows attributes to be compared using set membership, subset, and equality; in contrast, our PRBAC framework does not support multi-valued attributes, and it allows attributes to be compared using only equality. Multi-valued attributes and set relationships are very common in real policies.

We developed three algorithms for mining ABAC policies, which differ in the source of

user-permission data: ACLs, RBAC, and operation logs, respectively. To the best of our knowledge, these are the first policy mining algorithms for any ABAC framework.

Our algorithm for mining ABAC policies from ACLs and attribute data works as follows. It iterates over tuples in the given user-permission relation, uses selected tuples as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover additional tuples in the user-permission relation by replacing conjuncts in attribute expressions with constraints. After constructing candidate rules that together cover the entire user-permission relation, it attempts to improve the policy by merging and simplifying candidate rules. Finally, it selects the highest-quality candidate rules for inclusion in the generated policy. It can be used to mine an ABAC policy from an RBAC policy and attribute data, by expanding the RBAC policy into ACLs and then applying our algorithm. We also developed an extension of the algorithm to identify suspected noise in the input.

To evaluate the effectiveness of our algorithm, we wrote relatively small but non-trivial hand-written sample policies, created suitable attribute data for each of them, generated ACL policies from the ABAC policies and attribute data, ran our algorithm on the resulting ACL policies and attribute data, and compared the mined ABAC policies with the original policies. With minor exceptions, our algorithm successfully "discovers" the original ABAC policies for all case studies. The user can optionally supply some guidance to our algorithm, by indicating that some attributes are important. In our case studies, appropriate guidance can easily be determined based on the obvious importance of some attributes, or from examination of the policy generated with no guidance. A little bit of guidance eliminates the minor exceptions, leading to exact "discovery" of the original ABAC policies. To evaluate our algorithm on larger attribute datasets and the effectiveness of our noise detection technique, we generated synthetic attribute datasets of varying size for three of the case studies and synthetic policies of varying size and structures.

In Chapter 5, we present our algorithm for mining an ABAC policy from an RBAC policy and attribute data, which works as follows. First, it splits the roles in the given RBAC policy so that each role's assigned permissions are the Cartesian product of a set of resources and a set of operations. Second, it constructs an ABAC policy rule corresponding to each role (the splitting in the first step is necessary to ensure that each role can be translated into a single rule). Finally, it attempts to improve the policy by merging and simplifying rules. Merging is essential to produce a high-quality ABAC policy, because many roles can be expressed concisely by a single rule, if relevant attribute data is available. For example, the department chair roles for many departments can be expressed by a single rule if the relevant users and resources have an attribute indicating their department. Simplification directly improves policy quality, and it facilitates merging.

To evaluate the effectiveness of our algorithm at producing intuitive, high-level ABAC policies from RBAC policies, we manually wrote case study policies in RBAC and ABAC, applied our algorithm to the RBAC policy and accompanying attribute data, and compared the generated ABAC policy to the manually written one. Our algorithm successfully generates ABAC policies identical or similar to the manually written ABAC policies. Similarly, the user can optionally supply some guidance to our algorithm, by indicating that some attributes are important. And with a little bit of guidance, our algorithm generates ABAC policies identical or very similar to the manually written ones. In practice, the available attribute data is often incomplete. To evaluate the effectiveness of our algorithm in such

cases, we also performed experiments in which we omitted some relevant attribute data, and demonstrated that our algorithm uses role membership information effectively as a substitute for missing attribute data. To demonstrate the significance of preserving the structure of the RBAC policy, we wrote variants of some RBAC policies, with the same semantics (i.e., same user-permission relation) but different structure (i.e., different roles), and showed that our algorithm generates a different ABAC policy with corresponding structure for each variant.

In Chapter 6, we present our algorithm for mining an ABAC policy from logs and attribute data. The main challenge is that logs generally provide incomplete information about entitlements (i.e., granted permissions). Specifically, logs provide only a lower bound on the entitlements. Therefore, the generated policy should be allowed to include *over-assignments*, i.e., entitlements not reflected in the logs. We present an algorithm for mining ABAC policies from logs and attribute data. To the best of our knowledge, it is the first algorithm for this problem. It is based on our algorithm for mining ABAC policies from ACLs. At a high level, the algorithm works as follows. It iterates over tuples in the user-permission relation extracted from the log, uses selected tuples as seeds for constructing candidate rules, and attempts to generalize each candidate rule to cover additional tuples in the user-permission relation by replacing conjuncts in attribute expressions with constraints. After constructing candidate rules that together cover the entire user-permission relation, it attempts to improve the policy by merging and simplifying candidate rules. Finally, it selects the highest-quality candidate rules for inclusion in the generated policy.

We evaluated our algorithm on some relatively small but non-trivial handwritten case studies and on synthetic ABAC policies. The results demonstrate our algorithm's effectiveness even when the log reflects only a fraction of the entitlements. Although the original (desired) ABAC policy is not reconstructed perfectly from the log, the mined policy is sufficiently similar to it that the mined policy would be very useful as a starting point for policy administrators tasked with developing that ABAC policy.

Chapter 2

Mining Meaningful Roles

In this chapter, we formally define the RBAC policy mining problem and then present several RBAC policy mining algorithms that can easily be used to optimize a variety of RBAC policy quality metrics, including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability. We then evaluate the algorithms on publicly available access control policies and synthetic attribute data, and show that our algorithms compare favorably with other role mining algorithms. Finally, we discuss related work.

2.1 Problem Definition

This section defines the role mining problems that we consider. Our definitions are similar to those in [MCL$^+$10].

Policies and Policy Quality An *ACL policy* is a tuple $\langle U, P, UP \rangle$, where U is a set of users, P is a set of permissions, and $UP \subseteq U \times P$ is the user-permission assignment.

An *RBAC policy* is a tuple $\langle U, P, R, UA, PA, RH \rangle$, where R is a set of roles, $UA \subseteq U \times R$ is the user-role assignment, $PA \subseteq R \times P$ is the permission-role assignment, and $RH \subseteq R \times R$ is the role inheritance relation. Specifically, $\langle r, r' \rangle \in RH$ means that r is senior to r', hence all permissions of r' are also permissions of r, and all members of r are also members of r'.

An *RBAC policy with direct assignment* is a tuple $\langle U, P, R, UA, PA, RH, DA \rangle$, which is an RBAC policy extended with a direct user-permission assignment $DA \subseteq U \times P$. Allowing direct assignment of permissions to users provides more flexibility to handle anomalous permissions.

An RBAC policy is *consistent* with an ACL policy if $UA \circ PA = UP$, where $\circ$ is composition of relations. An RBAC policy with direct assignment is *consistent* with an ACL policy if $(UA \circ PA) \cup DA = UP$.

User-attribute data is a tuple $\langle A, f \rangle$, where A is a set of attributes, and f is a function such that $f(u, a)$ is the value of attribute a for user u. For simplicity, we assume that all attribute values are natural numbers.

A *policy quality metric* is a function from RBAC policies (or RBAC policies with direct assignment) to a totally-ordered set, such as the natural numbers. The ordering is chosen

so that small values indicate high quality; this might seem counter-intuitive at first glance, but it is natural for metrics such as policy size. We define two basic policy quality metrics and then consider combinations of them.

Weighted Structural Complexity (WSC) is a generalization of policy size [MCL+10]. For an RBAC policy π of the above form, we define weighted structural complexity by $\mathrm{WSC}(\pi) = w_1|R| + w_2|UA| + w_3|PA| + w_4|RH|$, where $|s|$ is the size (cardinality) of set s, and the w_i are user-specified weights. For an RBAC policy with direct assignment, the definition is the same except with an additional summand $w_5|DA|$.

Interpretability is a policy quality metric measures how well the roles in the policy can be characterized (interpreted) in terms of user attributes. Specifically, we quantify policy interpretability as *attribute mismatch*, which measures how well the sets of members of the roles can be characterized using expressions over user attributes. An *attribute expression* e is a function from the set A of attributes to sets of values. A user u *satisfies* an attribute expression e iff $(\forall a \in A.\ f(u,a) \in e(a))$. For example, if $A = \{\mathit{dept}, \mathit{level}\}$, the function e with $e(\mathit{dept}) = \{\mathrm{CS}\}$ and $e(\mathit{level}) = \{2,3\}$ is an attribute expression, which can be written with syntactic sugar as $\mathit{dept} \in \{\mathrm{CS}\} \wedge \mathit{level} \in \{2,3\}$. We refer to the set $e(a)$ as the conjunct for attribute a. Let $s_{\mathrm{u}}(e)$ denote the set of users that satisfy e. For an attribute expression e and a set U' of users, the *mismatch* of e and U', denoted $\mathrm{mismatch}(e, U')$, is the size of the symmetric difference of $s_{\mathrm{u}}(e)$ and U', where the symmetric difference of sets s_1 and s_2 is $s_1 \ominus s_2 = (s_1 \setminus s_2) \cup (s_2 \setminus s_1)$. The *attribute mismatch* of a role r, denoted $\mathrm{AM}(r)$, is $\min_{e \in E} \mathrm{mismatch}(e, \mathrm{assignedU}(r))$, where E is the set of all attribute expressions, and $\mathrm{assignedU}(r) = \{u \mid \langle u, r\rangle \in UA\}$. The *attribute mismatch* of an RBAC policy π (with or without direct assignment) is $\mathrm{AM}(\pi) = \sum_{r \in R} \mathrm{AM}(r)$. We define policy interpretability INT as attribute mismatch, i.e., $\mathrm{INT}(\pi) = \mathrm{AM}(\pi)$.

Compound policy quality metrics take multiple aspects of policy quality into account. One approach is to combine multiple policy quality metrics using a weighted sum; however, the choice of weights may be difficult or arbitrary. We combine metrics by Cartesian product, with lexicographic ordering on the tuples. Let $\text{INT-WSC}(\pi) = \langle \mathrm{INT}(\pi), \mathrm{WSC}(\pi)\rangle$ and $\text{WSC-INT}(\pi) = \langle \mathrm{WSC}(\pi), \mathrm{INT}(\pi)\rangle$.

Role Mining from ACLs The problem of *role mining from ACLs* is: given an ACL policy π_a and a policy quality metric Q, find an RBAC policy π_r that is consistent with π_a and has the best quality, according to Q, among policies consistent with π_a. The problem of *role mining with direct assignment from ACLs* is the same except that π_r is an RBAC policy with direct assignment.

Role Mining from ACLs and User Attributes The problem of *role mining from ACLs and user attributes* (with or without direct assignment) is the same as for role mining from ACLs, except that the input also includes user-attribute data, which may be used in the policy quality metric.

Our algorithms produce RBAC policies in which role membership is always defined by explicit user-role assignment, even when the current membership of a role can be characterized exactly by an attribute expression. In practice, assigning users to roles fully automatically

based on user attributes might be risky; requiring explicit user-role assignments by an administrator is safer. The administrator's effort can be reduced by an algorithm that suggests appropriate roles for new users, based on their attributes. For example, we can compute and store a best-fit attribute expression e_r for each role r, i.e., an attribute expression that minimizes the attribute mismatch for r. When a new user u is added to the access control system, the system suggests that u be made a member of the roles for which u satisfies the best-fit attribute expression, and it presents these suggested roles for u in descending order of the attribute mismatch. This allows good suggestions even in the presence of noise.

2.2 Algorithms

This section presents our role mining algorithms. In general, they compute only approximate solutions to the role-mining problem: the generated RBAC policy is always consistent with the given ACL policy, but it does not always have the best possible quality. This is a common limitation of role mining algorithms, because computing an optimal solution is NP-hard for policy quality metrics of interest [MCL+10].

2.2.1 Elimination Algorithm

Our *elimination algorithm* has three phases. Phase 1, role generation, generates a candidate role hierarchy that contains all "interesting" candidate roles. Phase 2, role elimination, removes roles from the candidate role hierarchy if the removal preserves consistency with the given ACL policy and improves policy quality. Phase 3, role restoration, adds some removed roles back to the policy, if this improves policy quality.

Phase 1: Role Generation Our algorithm for role generation is based closely on CompleteMiner [VAW06], although for increased scalability, we could easily substitute FastMiner [VAW06] or the FP-Tree approach [HPY00, MLL+09]. Roles are characterized primarily by the set of permissions assigned to the role. An *initial role* has a set of permissions that contains all permissions assigned to some user. A *candidate role* has a set of permissions obtained by intersecting the permission sets of an arbitrary number of initial roles. As argued in [VAW06], in the absence of other information on which to base the construction of candidate roles, this method generates all interesting candidate roles. Pseudo-code for this construction appears in Figure 2.1. It is essentially the same as the pseudo-code for CompleteMiner in [VAW06]. It uses the functions assignedP$(r) = \{p \in P \mid \langle r, p\rangle \in PA\}$ and assignedU$(r) = \{u \in U \mid \langle u, r\rangle \in UA\}$.

CompleteMiner does not produce a role hierarchy. Our algorithm computes a role inheritance relation with the maximum amount of inheritance: a candidate role r_p inherits from another role r_c whenever the permissions of r_p are a superset of the permissions of r_c. Furthermore, when that inheritance relation is introduced, the permissions inherited by r_p from r_c are removed from the permissions explicitly assigned to r_p by PA, and the members inherited by r_c from r_p are removed from the members explicitly assigned to r_c by UA. Pseudo-code appears in Figure 2.2. It uses functions authP$(r) = \{p \in P \mid \exists r' \in R.\ \langle r, r'\rangle \in$

```
    // Create initial roles.
1:  InitRole = ∅
2:  permSets = ⋃_{u∈U} {p ∈ P | ⟨u, p⟩ ∈ UP}
3:  for ps in permSets \ {∅}
4:     r = new Role()
5:     InitRole = InitRole ∪ {r}
6:     PA = PA ∪ ({r} × ps)
7:  end for

    // Compute all intersections of initial roles.
8:  R = ∅
9:  for r in InitRole
10:    InitRole = InitRole \ {r}
11:    for r′ in InitRole
12:       P = assignedP(r) ∩ assignedP(r′)
13:       if ¬empty(P)∧ ∄r″ ∈ R. assignedP(r″) = P
14:          r″ = new Role()
15:          PA = PA ∪ ({r″} × P)
16:          R = R ∪ {r″}
17:       end if
18:    end for
19:    for r′ in R
20:       P = assignedP(r) ∩ assignedP(r′)
21:       if ¬empty(P)∧ ∄r″ ∈ R. assignedP(r″) = P
22:          r″ = new Role()
23:          PA = PA ∪ ({r″} × P)
24:          R = R ∪ {r″}
25:       end if
26:    end for
27: end for
28: R = R ∪ InitRole
```

Figure 2.1: Role generation, step 1: compute candidate roles.

$RH^* \wedge \langle r', p\rangle \in PA\}$ and $\mathrm{authU}(r) = \{u \in U \mid \exists r' \in R.\ \langle r', r\rangle \in RH^* \wedge \langle u, r'\rangle \in UA\}$, where RH^* is the reflective transitive closure of RH.

A role hierarchy has *full inheritance* if every two roles that can be related by the inheritance relation are related by it, i.e., $\forall r, r' \in R.\ \mathrm{authP}(r) \supseteq \mathrm{authP}(r') \wedge \mathrm{authU}(r) \subseteq \mathrm{authU}(r') \Rightarrow \langle r, r'\rangle \in RH^*$. Guo *et al.* call this property *completeness* [GVA08].

All of our algorithms generate RBAC policies with full inheritance. Although relaxing this requirement would allow our algorithms to achieve better policy quality in some cases, we impose this requirement, because in the absence of other information, all of these possible inheritance relationships are equally plausible, so removing any of them risks removing some that are semantically meaningful and desirable.

```
   // Initialize variables. Assign users to roles.
1:  UA = ∅; RH = ∅
2:  for u in U
3:      P = {p ∈ P | ⟨u, p⟩ ∈ UP}
4:      for r in R
5:          if authP(r) ⊆ P
6:              UA = UA ∪ {⟨u, r⟩}
7:          end if
8:      end for
9:  end for

   // Add inheritance edges, and eliminate inherited permissions and members from
   // UA and PA.
10: for r in R
11:     parents = {r′ ∈ R | ⟨r, r′⟩ ∈ RH}      // parents of r
12:     for r′ in R \ {r}
13:         if authP(r′) ⊆ authP(r) ∧ ∀r′′ ∈ parents. authP(r′) ⊈ authP(r′′)
14:             RH = RH ∪ {⟨r, r′⟩}
15:             for ⟨r, p⟩ in PA
16:                 if p ∈ authP(r′)
17:                     PA = PA \ {⟨r, p⟩}
18:                 end if
19:             end for
20:             for ⟨u, r′⟩ in UA
21:                 if u ∈ assignedU(r)
22:                     UA = UA \ {⟨u, r′⟩}
23:                 end if
24:             end for
25:             for r′′ in parents
26:                 if authP(r′′) ⊈ authP(r′)
27:                     RH = RH \ {⟨r, r′′⟩}
28:                 end if
29:             end for
30:         end if
31:     end for
32: end for
```

Figure 2.2: Role generation, step 2: construct role hierarchy, based on R and PA from step 1.

Phase 2: Role Elimination Roughly, the role elimination phase removes roles from the candidate role hierarchy if the removal preserves consistency with the given ACL policy and improves policy quality. When a role r is removed, the role hierarchy is adjusted to preserve inheritance relations between parents and children of r, and the user assignment

and permission assignment are adjusted to explicitly assign to other roles the members and permissions that they previously inherited from r.

The order in which roles are considered for removal is important, because it may lead to different RBAC policies in the end. We control this ordering with a *role quality metric* Q_{role}, which maps roles to an ordered set, with the interpretation that large values denote high quality (note: this is opposite to the interpretation of the ordering for policy quality metrics). Low-quality roles are considered for removal first. The algorithm is parameterized by the choice of role quality metric. We consider three basic role quality metrics and then consider combinations of them.

Clustered size measures how well user permissions are clustered in the role. A first attempt at formulating such a metric might simply be the total number of *UP* pairs (i.e., elements of the *UP* relation) that are covered by the role, or, equivalently but with the metric normalized to be in the range $[0, 1]$, the fraction of all *UP* pairs covered by the role. However, such a metric would give the same rating to a role r_1 that covers one permission for each of 10 users and a role r_2 that covers 5 permissions for each of 2 users, even though r_2 is preferable; for example, if all of the users have exactly 5 permissions, then the two users in r_2 would not need to belong to any other roles, while all of the users in r_1 would need to belong to other roles as well. To take this into account, we define the clustered size metric to be equal to the fraction of the permissions of the role's members that are covered by this role; formally,

$$\begin{aligned}
\text{assignedUP}(r) &= \{\langle u, p\rangle \in \mathit{UP} \mid u \in \text{assignedU}(r) \\
&\qquad\qquad \wedge\, p \in \text{assignedP}(r)\} \\
\text{clsSz}(r) &= |\text{assignedUP}(r)| \div |\{\langle u, p\rangle \in \mathit{UP} \mid u \in \text{assignedU}(r)\}|
\end{aligned}$$

The numerator considers assigned users and permissions, instead of authorized users and permissions, so that a role gets credit only for the *UP* pairs that it covers by itself, not for *UP* pairs covered by its ancestors or descendants.

Attribute fitness measures how well the set of members of a role can be characterized (interpreted) in terms of user attributes. It is based on attribute mismatch, defined in Section 2.1, normalized to be in the range $[0, 1]$ and subtracted from 1 so that higher values of the metric indicate higher quality; formally, $\text{attrFit}(r) = 1 - \frac{\text{AM}(r)}{|\text{assignedU}(r)|}$.

Redundancy measures how many other roles also cover the *UP* pairs covered by a role. Removing a role with higher redundancy is less likely to prevent subsequent removal of other roles, so we eliminate roles with higher redundancy first. Values of the redundancy metric are pairs, with lexicographic order. The redundancy of role r is the negative of the minimum, over *UP* pairs $\langle u, p\rangle$ covered by r, of the number of other removable roles that cover $\langle u, p\rangle$ (we take the negative so that roles with more redundancy have lower quality and hence get considered for removed first).

$$\begin{aligned}
&\text{authUP}(r) = \{\langle u, p\rangle \in \mathit{UP} \mid u \in \text{authU}(r) \,\wedge\, p \in \text{authP}(r)\} \\
&\text{redun}(\langle u, p\rangle) = |\{r \in R \mid \langle u, p\rangle \in \text{authUP}(r) \wedge \text{removable}(r)\}| \\
&\text{redun}(r) = -\min_{\langle u,p\rangle \in \text{authUP}(r)}(\text{redun}(\langle u, p\rangle))
\end{aligned}$$

Compound role quality metrics can be formed in the same ways as compound policy

```
1:  π = policy produced by role
        generation
2:  q = Q_pol(π)
3:  workList = list containing
        removable roles in π
4:  changed = true
5:  while ¬empty(workList) ∧ changed
6:      sort workList in ascending
        order by Q_role
7:      changed = false
8:      for r in workList
9:          if ¬removable(r)
10:             remove r from workList
11:         else
12:             π′ = removeRole(π, r)
13:             q′ = Q_pol(π′)
14:             if q′ < δq
15:                 π = π′
16:                 q = q′
17:                 changed = true
18:                 remove r from
                    workList
19:             end if
20:         end if
21:     end for
22: end while

function removeRole(π, r)
23: ⟨U, P, R, UA, PA, RH⟩ = π
24: R = R \ {r}
25: for ⟨r1, r⟩ in RH
26:     RH = RH \ {⟨r1, r⟩}
27:     for ⟨r, r2⟩ in RH
28:         if ⟨r1, r2⟩ ∉ RH*
29:             RH = RH ∪ {⟨r1, r2⟩}
30:         end if
31:     end for
32:     for ⟨r, p⟩ in PA
33:         if p ∉ authP(r1)
34:             PA = PA ∪ {⟨r1, p⟩}
35:         end if
36:     end for
37: end for
38: for ⟨r, r2⟩ in RH
39:     RH = RH \ {⟨r, r2⟩}
40:     for ⟨r, u⟩ in UA
41:         if u ∉ authU(r2)
42:             UA = UA ∪ {⟨r2, u⟩}
43:         end if
44:     end for
45: end for
46: return ⟨U, P, R, UA, PA, RH⟩
```

Figure 2.3: Role elimination.

quality metrics, e.g., max(clsSz, attrFit).

Our algorithm may remove a role even if the removal worsens policy quality slightly. Specifically, we introduce a *quality change tolerance* δ, with $\delta \geq 1$, and we remove a role if the quality Q' of the RBAC policy resulting from the removal is related to the quality Q of the current RBAC policy by $Q' < \delta Q$ (recall that, for policy quality metrics, smaller values are better). Choosing $\delta > 1$ partially compensates for the fact that a purely greedy approach to policy quality improvement is not an optimal strategy.

Pseudo-code for role elimination appears in Figure 2.3. It is parameterized by a policy quality metric Q_{pol}, a role quality metric Q_{role}, and a quality change tolerance δ. A role is *removable* if every *UP*-pair covered by r is covered by at least one other role currently in the policy; formally,

$$\begin{aligned} \text{removable}(r) = \forall \langle u, p\rangle \in \text{authUP}(r).\ \exists r' \in R.\\ r' \neq r \ \wedge\ \langle u, p\rangle \in \text{authUP}(r') \end{aligned}$$

A removable role can be removed while preserving consistency with the given ACL policy. The removeRole function removes a role r, adjusts the role hierarchy to preserve inheritance relations between parents and children of r, and adjusts the user assignment and permission

assignment to explicitly assign to other roles the members and permissions that they previously inherited from r. The removability test in line 9 is necessary because a role that is initially removable might become unremovable, due to other removals. The quality of each role is computed only in line 6, immediately before sorting the worklist. Role quality metrics may change as roles are removed and hence are re-computed each time line 6 is executed.

Phase 3: Role Restoration Phase 3 restores removed roles when this improves policy quality. Specifically, it considers each removed role r, in the same order that the roles were removed, and restores r if this improves the policy quality. Pseudo-code to restore a role appears in Figure 2.4. It uses the relation $\prec$ defined by $r \prec r' = \mathrm{authP}(r) \subset \mathrm{authP}(r')$. It makes r a child of roles r' such that $r \prec r' \wedge \neg\exists r'' \in R.\ r \prec r'' \prec r'$, makes r a parent of roles r' such that $r' \prec r \wedge \neg\exists r'' \in R.\ r' \prec r'' \prec r$, and adjusts the permission assignment, user assignment, and inheritance relations of roles related to r to eliminate redundancy.

Direct User-Permission Assignment If direct user-permission assignment is allowed, we add a final phase that replaces roles with direct assignment if that improves policy quality. Pseudo-code appears in Figure 2.5; variable π initially contains the policy produced by phase 3, which contains no direct assignments, i.e., $DA = \emptyset$.

Determining Algorithm Parameters Different choices of role quality metric Q_{role} and quality change tolerance δ may give the best results for different datasets, so we enclose the algorithm in a loop that tries all combinations of the following values for those parameters and returns the result from the best combination: Q_{role} in $\{\langle \mathrm{redun}, \mathrm{clsSz}\rangle, \langle \max(\mathrm{attrFit}, \mathrm{clsSz}), \mathrm{redun}\rangle\}$, and δ in $\{1, 1.001, 1.002\}$. We also experimented with $\mathrm{sum}(\mathrm{clsSz}, \mathrm{attrFit})$ for Q_{role}, and with larger values for δ, but that did not improve the results.

2.2.2 Selection Algorithm

Our *selection algorithm* works in the opposite way as the elimination based algorithm. Specifically, it starts with an empty policy and repeatedly adds candidate roles to the policy. The selection algorithm is parameterized by a role quality metric. In phase 1, candidate roles are generated as in the elimination algorithm (see Figure 2.1). In phase 2, candidate roles are added to the RBAC policy in order of descending role quality, until the RBAC policy is consistent with the given ACL policy. Phase 3 performs pruning: for each role r in the policy in the reverse order that the roles were added, checks whether the role is removable, and if so, whether removing it improves policy quality, and if so, removes it.

2.2.3 Complete Algorithm

Our *complete algorithm* has two phases. Phase 1 generates a hierarchical RBAC policy in exactly the same way as the elimination algorithm. Phase 2 is role removal. While the elimination algorithm heuristically takes a greedy approach to removals, the complete algorithm considers all subsets of the set of removable roles, to find the set of removals that produces the policy with the highest quality.

```
function restoreRole(π, r)
1:  ⟨U, P, R, UA, PA, RH⟩ = π
2:  for r′ in R
3:      if r ≺ r′ ∧ ¬∃r″ ∈ R. r ≺ r″ ≺ r′
            // make r a child of r′
4:          assignedP(r′) = assignedP(r′) \ authP(r)
5:          assignedU(r) = assignedU(r) \ authU(r′)
6:          RH = RH ∪ {⟨r′, r⟩}
7:          for r″ in R such that ⟨r′, r″⟩ ∈ RH   // children of r′
8:              if r″ ≺ r
                    // remove r″ as a child of r′. r″ will be
                    // a child of r and a grandchild of r′
9:                  RH = RH \ {⟨r′, r″⟩}
10:             end if
11:         end for
12:     end if
13:     if r′ ≺ r ∧ ¬∃r″ ∈ R. r′ ≺ r″ ≺ r
            // make r a parent of r′
14:         assignedP(r) = assignedP(r) \ authP(r′)
15:         assignedU(r′) = assignedU(r′) \ authU(r)
16:         RH = RH ∪ {⟨r, r′⟩}
17:         for r″ in R such that ⟨r″, r′⟩ ∈ RH   // parents of r′
18:             if r ≺ r″
                    // remove r″ as a parent of r′. r″ will be
                    // a parent of r and a grandparent of r′
19:                 RH = RH \ {⟨r″, r′⟩}
20:             end if
21:         end for
22:     end if
23: end for
24: R = R ∪ {r}
25: return ⟨U, P, R, UA, PA, RH⟩
```

Figure 2.4: Restore role r to policy π.

```
1: for r in R
2:     π1 = removeRole(r)
3:     π2 = π1 with all UP pairs in the given ACL policy
           that are not covered in π1 added to DA
4:     if Q_pol(π2) < δQ_pol(π)
5:         π = π2
6:     end if
7: end for
```

Figure 2.5: Create direct user-permission assignment.

Dataset	$\|U\|$	$\|P\|$	$\|UP\|$	high-fit N_a	high-fit AF	low-fit N_a	low-fit AF
healthcare	46	46	1486	20	1	5	0.79
domino	79	231	730	20	1	12	0.48
emea	35	3046	7220	20	1	6	0.56
apj	2044	1146	6841	40	0.94	10	0.57
firewall-1	365	709	31951	40	0.997	15	0.58
firewall-2	325	590	36428	40	1	10	0.50
americas-small	3477	1587	105205	50	0.95	9	0.36

Figure 2.6: Information about datasets. N_a is the number of attributes. AF is the attribute fit.

To avoid explicitly storing the set of sets of removable roles that have been explored so far, our role removal algorithm is expressed as a recursive search. Removal of one role may prevent subsequent removal of another role, but removals commute in the sense that, if it is possible to remove r_1 and then remove r_2, then it is also possible to remove r_2 and then remove r_1, and these two sequences of removals lead to the same policy. To ensure that the algorithm does not unnecessarily explore the same removals in multiple orders, we impose an arbitrary ordering on the removable roles, by storing them in a list R_{rmv}, and the algorithm considers only sequences of removals consistent with that ordering; in other words, it considers sequences of removals that correspond to subsequences (not necessarily contiguous) of R_{rmv}. The algorithm is parameterized by a policy quality metric Q_{pol}. The algorithm is complete in the following sense: if Q_{pol} is WSC, then the complete algorithm computes a policy that minimizes WSC among policies consistent with the given ACL policy; for other policy quality metrics Q_{pol}, the complete algorithm computes a policy that minimizes Q_{pol} among policies that are consistent with the given ACL policy and have full inheritance.

2.3 Datasets

We know of no publicly available real ACL policies with user attribute data, so we use publicly available real ACL policies, described next, together with synthetic user attribute data, generated as described below.

The ACL policies are listed in Figure 2.6. They originate from Hewlett-Packard (HP) Labs [EHM+08]. The healthcare dataset was obtained by HP Labs from the U.S. Veteran's Administration, which has developed a comprehensive list of the healthcare permissions that may be assigned to licensed or certified providers. The domino data is from a set of user and access profiles for a Lotus Domino server. americas-small is a network access control policy from Cisco firewalls used to manage external business partner's access to HP's network. apj and emea are similar but smaller datasets. HP Labs produced the firewall-1 and firewall-2 datasets based on analysis of network connectivity permitted by Checkpoint firewall rules.

Generation of User Attribute Data Molloy *et al.* provide summary information about non-public user attribute data and ACL policies from three customers [MLC11]; we exploit this to make our synthetic attribute data have some approximately realistic characteristics. Based on the information in the paper, we construct the following distributions: (a) for each customer i, we fit an exponential distribution $card_i$ to the distribution of cardinalities of user attributes for that customer. (b) for each attribute of each customer, we fit a Zipf distribution to the distribution of values of that attribute (based on the information in [MLC11, Figures 3-5]), to obtain a Zipf-distribution exponent for each attribute, and then we fit a Weibull distribution *zipfExp* to the resulting distribution of Zipf-distribution exponents. The individual Zipf-exponents obtained from our measurements of the charts in [MLC11, Figures 3-5] have considerable uncertainty, due to the limited information in those charts, but these uncertainties might average out to some extent, making the parameters of the Weibull distribution *zipfExp* somewhat more robust.

Our algorithm for generating user attribute data is parameterized by an ACL policy and the desired number N_a of attributes. The algorithm has two phases. Phase 1 generates user attribute data for each attribute separately, independent of the ACLs. Phase 2 modifies the user attribute data to improve its fit with the ACLs. In more detail, phase 1 starts by identifying the customer i in [MLC11] for which the number of users is closest to the number $|U|$ of users in the given ACL policy, and then, for each of the desired attributes, select a cardinality c_a from $card_i$ and a Zipf-exponent s_a from *zipfExp*. Next, the value of attribute a for each user is selected from a Zipf distribution with c_a elements and exponent s_a. We take all attribute values to be natural numbers interpreted as ranks in the Zipf distribution (0 is the most common value, 1 is the second most common value, etc.).

Phase 2 tries to reduce the attribute mismatch for each permission. Let U_p denote the set of users with permission p, i.e., $U_p = \{u \in U \mid \langle u, p\rangle \in \mathit{UP}\}$. For each permission p, we first compute an attribute expression e_p representing the least superset of U_p expressible as an attribute expression; e_p is given by $e_p(a) = \{f(u,a) \mid u \in U_p\}$. e_p may be a very loose upper bound on U_p, so we convert it to a lower bound on U_p by repeatedly removing an attribute value from a conjunct of e_p until $s_{\mathrm{u}}(e_p) \subseteq U_p$; in each iteration, we remove the attribute value with the largest value of the metric m, where, for a value v in the conjunct for attribute a

$$\begin{aligned} m(v,a) = \quad & |\{u \in U \mid f(u,a) = v \wedge u \notin U_p\}| \\ & - |\{u \in U \mid f(u,a) = v \wedge u \in U(p)\}| \end{aligned}$$

Finally, we try to make the lower bound tighter as follows: for each user u in $U_p \setminus s_{\mathrm{u}}(e_p)$, for each attribute a such that $f(u,a) \notin e_p(a)$, if adding $f(u,a)$ to $e_p(a)$ preserves the fact that $s_{\mathrm{u}}(e_p) \subseteq U_p$, then add $f(u,a)$ to $e_p(a)$, otherwise try to modify f so that $f(u,a) \in e_p(a)$, by swapping the values of $f(u,a)$ and $f(u',a)$ for some other user u', provided the swap does not affect whether u' satisfies the attribute expressions already constructed for other permissions. Note that swapping values of attributes between users preserves the distribution of values of each attribute.

The *attribute fit* of the resulting attribute assignment is defined as $1 - \frac{1}{|\mathit{UP}|}\sum_{p \in P} \mathrm{mismatch}([\![e_p]\!], U_p)$. For each dataset, we start with $N_a = 10$, generate user attribute data, and compute the attribute fit. If it is above 0.9, we stop, otherwise we increment the number of attributes by 10 and try again, until the attribute fit is above 0.9.

We call the resulting user attribute data the *high-fit* user attribute data.

In practice, the available user attribute data will often have a lower attribute fit than 0.6, e.g., because some relevant user attributes are unavailable. Therefore, we also produce a version of the user attribute data with fewer attributes; specifically, we discard attributes one at a time, until the attribute fit drops below 0.6 (except we use a higher threshold of 0.8 for healthcare, otherwise N_a is very low). We call this the *low-fit* user attribute data.

Figure 2.6 contains information about the generated user attribute data. Generation of user attribute data takes only a few minutes for small datasets, and it takes less than an hour for the largest dataset.

2.4 Experimental Results

This section compares our algorithms with each other, compares the elimination algorithm (which is best among our algorithms) with prior work, and explores the effects of different policy quality metrics and role quality metrics.

Comparison of Elimination Algorithm with Hierarchical Miner and Graph Optimisation Figure 2.7 shows the WSC and interpretability (using the high-fit attribute data) of policies produced by the elimination algorithm and Hierarchical Miner (HM) [MCL+10] with policy quality metric WSC-INT and the WSC of policies produced Graph Optimisation (GO) [ZRE07] (modified slightly by Molloy *et al.* to use WSC as the policy quality metric). The weight vector for WSC contains all ones except that the weight for direct assignment is infinity (in other words, direct assignment is prohibited). In the comparison of eight role mining algorithms in [MLL+09] and the comparison of four role mining algorithms in [MCL+10], for this weight vector, the best WSC for every dataset is achieved by either HM or GO. *Figure 2.7 shows that the elimination algorithm achieves smaller or equal WSC than HM and GO on every dataset, while simultaneously achieving good policy interpretability* (Figure 2.11 shows that the elimination algorithm simultaneously achieves good results for both components of the policy quality metric). The WSC from HM and GO are 2.7% worse and 14.0% worse, respectively, averaged over the datasets, compared to the WSC from the elimination algorithm. The INT from HM is 46.3% worse, averaged over the datasets, compared to the INT from the elimination algorithm; this is not surprising, because HM does not consider user attributes or policy interpretability. The results for HM are computed from policies produced by HM that Molloy sent to us. The results for GO are from [MCL+10, Table VI] for all datasets except americas-small, which is not used in [MCL+10]; the results for GO for americas-small are from [MLL+09, Table 4].

On a PC with an Intel Core 2 Quad 2.66 GHz CPU (the processor has 4 cores, but our code is purely sequential), the elimination algorithm terminates in 30 seconds or less for all datasets except americas-small, which takes about 3.5 minutes. Running times for HM and GO are not reported in [ZRE07, MLL+09, MCL+10], and the implementations of HM and GO described in those papers are not publicly available. We fit curves to a graph of running time *vs.* $|UP|$ for the datasets in Figure 2.6 and found that a quadratic function fits well.

Figure 2.8 shows the result of our elimination algorithm when allowing direct assignments, with a WSC weight vector containing all ones. The results for HM are computed from

	Elimination		HM		GO
Dataset	INT	WSC	INT	WSC	WSC
healthcare	14	144	16	149	168
domino	21	404	30	418	413
emea	32	3709	92	3795	3888
apj	392	4248	411	4282	4600
firewall-1	48	1385	59	1426	1543
firewall-2	7	945	7	945	960
americas-small	214	6330	324	6710	9721

Figure 2.7: Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is prohibited.

	Elimination		HM		GO
Dataset	INT	WSC	INT	WSC	WSC
healthcare	9	140	10	142	168
domino	7	371	9	379	413
emea	36	3644	39	3693	3888
apj	130	3827	164	3862	4600
firewall-1	17	1340	21	1349	1543
firewall-2	4	944	4	944	960
americas-small	182	6214	198	6468	9721

Figure 2.8: Comparison of elimination algorithm with policy quality metric WSC-INT, Hierarchical Miner, and Graph Optimisation, when direct user-permission assignment is permitted.

policies producd by HM that Molloy sent us. The results for GO are from [MCL+10, Table VII] for all datasets except americas-small, which is not used in [MCL+10]; the results for GO for americas-small are from [MLL+09, Table 4]. The original GO does not consider direct assignment, but Molloy *et al.* extended GO to support it. *Figure 2.8 shows that the elimination algorithm achieves smaller WSC than HM and GO on every dataset, while simultaneously achieving good policy interpretability.* The WSC from HM and GO are 1.5% worse and 18.8% worse, respectively, averaged over the datasets, compared to the WSC from the elimination algorithm. The INT from HM is 15.2% worse, averaged over the datasets, compared to the INT from the elimination algorithm.

Comparison of Elimination Algorithm with Attribute Miner Among prior work on role mining that takes policy interpretability into account, the most closely related is Molloy *et al.*'s work on Attribute Miner [MCL+10]. Figure 2.9 compares the elimination algorithm (using the redundancy role quality metric and $\delta = 1.001$) with Attribute Miner [MCL+10]. Molloy *et al.*'s implementation of Attribute Miner is not publicly available, so the results for Attribute Miner are from our own implementation of it. Attribute Miner is designed to optimize the policy quality metric Weighted Structural Complexity with Attributes (WSCA)

[MCL+10]. WSCA differs from WSC in how the size of the user-role assignment is measured. In WSC, it is simply $|UA|$ or equivalently $\sum_{r \in R} |U(r)|$, where $U(r)$ is the membership (assigned users) of role r. In WSCA, if $U(r)$ can be characterized exactly by an attribute expression $D(r)$, the size of $D(r)$ (i.e., the number of conjuncts) is used instead of $|U(r)|$; otherwise, the geometric mean of $|U(r)|$ and $|s_{\mathrm{u}}B(r)|$ is used instead of $|U(r)|$, where $B(r)$ is the attribute expression that is the least upper bound for $U(r)$. We have some reservations about WSCA: (1) use of the geometric mean of $|U(r)|$ and $|s_{\mathrm{u}}B(r)|$ seems unintuitive, since it does not directly measure either the size or the interpretability of the role; (2) WSCA is very sensitive to whether a role can be characterized exactly by an attribute expression—a small change to the input data can significantly change the WSCA associated with a role, because $|D(r)|$ is often much smaller than $|U(r)|$; (3) as discussed at the end of Section 2.1, it might be safer to use attribute expressions to suggest role membership than to define role membership. Nevertheless, we use WSCA for this comparison, because Attribute Miner is designed to optimize WSCA and would probably fare poorly in a comparison based on INT-WSC.

Attribute Miner, as described in [MCL+10] uses attribute expressions that are conjunctions of positive literals over Boolean attributes. We implemented a generalized version of Attribute Miner that uses attribute expressions of the form described in Section 2.1. This involves straightforward changes to the code that computes least upper bounds and to the definition of the size of an attribute expression, which is used in the definition of WSCA [MCL+10, Definition 13] and in the definition of the cost of an attribute role [MCL+10, Table III]. We define the size of an attribute expression e to be $\sum_{a \in A} |e(a)|$. Attribute Miner takes user attribute data and a set of candidate roles as input; we generate the set of candidate roles using Phase 1 of the elimination algorithm.

Figure 2.9 shows that the elimination algorithm achieves better WSCA than Attribute Miner on every dataset. With the high-fit attribute data, Attribute Miner is 78% worse, averaged over the datasets, i.e., the average of the ratios of the WSCA values obtained using the two algorithms is 1.78; the median of the ratios is 1.38. With the low-fit attribute data Attribute Miner is 57% worse, averaged over the datasets, i.e., the average of the ratios of the WSCA values obtained using the two algorithms is 1.57; the median of the ratios is 1.36.

Comparison of Our Algorithms Figure 2.10 contains results for the elimination algorithm with the redundancy role quality metric and the selection algorithm with role quality metric max(attrFit, clsSz). We use INT-WSC as the policy quality metric for both algorithms. The weight vector for WSC contains all ones except that the weight for direct assignment is infinity (in other words, direct assignment is prohibited). Figure 2.10 shows that the elimination algorithm achieves the same or better results than the selection algorithm on both components of the policy quality metric for every dataset. We ran the complete algorithm on the smallest dataset, healthcare, with Q_{pol}=WSC. The result has WSC = 141, which is better than elimination algorithm (WSC = 144) and HM (WSC = 149). We started to run the complete algorithm on the second smallest dataset, domino, but we aborted it after 30 hours.

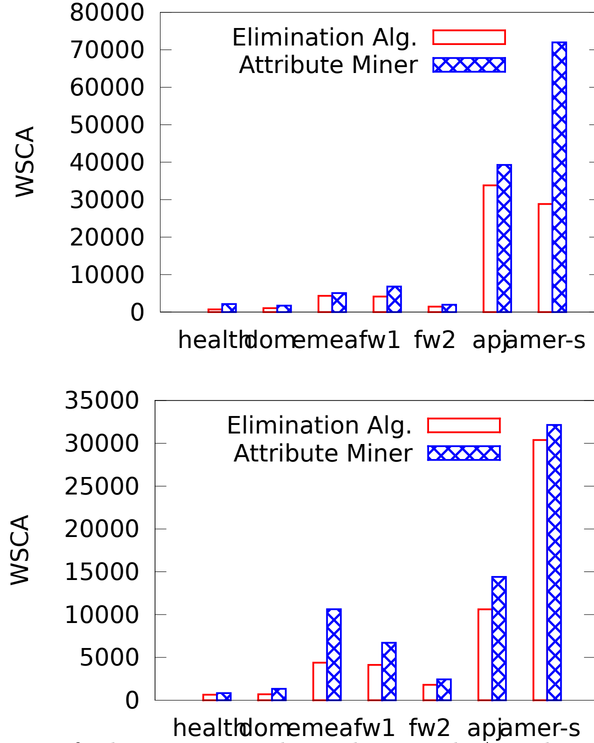

Figure 2.9: Comparison of elimination algorithm and Attribute Miner (AM). Names of datasets are abbreviated, e.g., fw1 abbreviates "firewall-1". The upper and lower graphs use the high-fit and low-fit user attribute data, respectively.

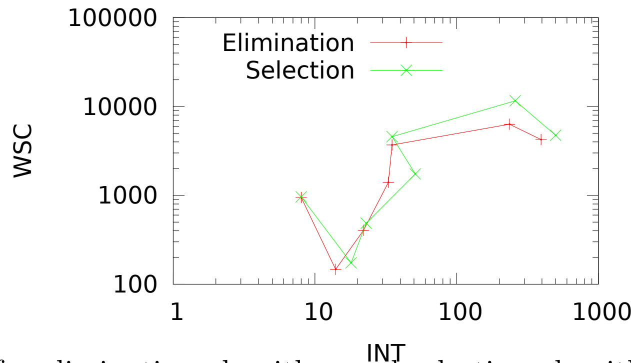

Figure 2.10: Results for elimination algorithm and selection algorithm, with policy quality metric INT-WSC. The clusters of points correspond, from left to right in the order they are connected, to the datasets in the following order: firewall-2 healthcare, domino, firewall-1, emea, americas-small, apj.

Effect of Policy Quality Metric in Elimination Algorithm Figure 2.11 compares the quality of policies produced by the elimination algorithm with policy quality metrics WSC-INT and INT-WSC, using the high-fit user attribute data. Recall that the elimination algorithm tries multiple role quality metrics Q_{role} and quality change tolerances δ; the tables also show the best combination of those parameters for each policy quality metric and each dataset. Surprisingly, for all of these datasets, it makes little or no difference whether priority is given to WSC or interpretability.

	WSC-INT				INT-WSC			
Dataset	INT	WSC	Q_{role}	δ	INT	WSC	Q_{role}	δ
healthcare	14	144	rdn	1.001	14	144	rdn	1.001
domino	21	404	max	1.001	21	404	max	1.001
emea	32	3709	max	1.000	32	3709	max	1.000
apj	392	4248	rdn	1.000	384	4331	rdn	1.002
firewall-1	48	1385	max	1.000	44	1419	max	1.003
firewall-2	7	945	max	1.000	7	945	max	1.000
amer-small	214	6330	max	1.000	180	6912	red	1.003

Figure 2.11: Comparison of two different policy quality metrics in elimination algorithm. "rdn" and "max" denote $\langle \text{redun}, \text{clsSz} \rangle$ and $\langle \max(\text{attrFit}, \text{clsSz}), \text{redun} \rangle$, respectively.

Effect of Role Quality Metric and Quality Change Tolerance in Elimination Algorithm We compared the results of the elimination algorithm with policy quality metric INT-WSC and four role quality metrics: redundancy, max(attrFit, clsSz), and the "reverse" of each of these, obtained by taking the negative of the value. The reverse orders exemplify a bad choice of role quality metric. We used $\delta = 1.0$ and policy quality metric WSC-INT with all four role quality metrics. Averaged over the datasets, using reverse-max(attrFit, clsSz) instead of max(attrFit, clsSz) worsens policy interpretability by 5.0% and WSC by 0.9%, and using reverse-redundancy instead of redundancy worsens policy interpretability by 3.9% and WSC by 1.0%. This shows that the order in which roles are considered for removal has a small but non-negligible effect.

We also compared the results of the elimination algorithm using all six combinations of the two role quality metrics and three quality change tolerances specified in Section 2.2. We found that the combination Q_{role} = redun and $\delta = 1.001$ gives the best result or close to it—within 2% for WSC and interpretability—for every dataset in our experiments.

2.5 Related Work

The literature on role mining is sizable, so we discuss only the most closely related work.

Vaidya *et al.*'s RoleMiner algorithm has two phases [VAW06]. Phase 1 produces a set of candidate roles, each represented by a set of permissions. They give two algorithms for this: CompleteMiner, which we adopt as the first step in Phase 1 of our elimination algorithm, and FastMiner, which is similar to CompleteMiner but more scalable, because it considers only pairwise intersections of initial roles. Phase 2 prioritizes the candidate roles produced by Phase 1. The prioritized list of roles is the final result of the algorithm. The algorithm does not attempt to determine which candidate roles to include in such an RBAC policy, to produce a role inheritance relation, or to assign users to roles. In contrast, our algorithm addresses these issues in order to produce an RBAC policy. Vaidya *et al.* also developed algorithms for computing an RBAC policy with minimal $|R|$ that is consistent with a given ACL policy [VAG07]. Lu *et al.* [LVA08] present role mining algorithms that minimize either $|R|$ or $|UA| + |PA|$. None of these papers considers more general policy size metrics (such as WSC), role hierarchy, or interpretability of roles with respect to user attribute data.

Zhang *et al.*'s Graph Optimisation (GO) algorithm starts with each user's permission set as a candidate role, and repeatedly splits or merges roles when the transformation improves policy quality [ZRE07]. They do not consider interpretability of roles with respect to user attribute data. The data in Figures 2.7 and 2.8 show that the elimination algorithm achieves better WSC than GO does. The main reasons are: (1) GO performs role generation and role selection in a single phase, considering new candidate roles lazily according to a greedy heuristic, instead of eagerly generating all candidate roles in an initial phase; as a result, GO is faster, but it might fail to consider some useful roles; (2) it appears from the paper that GO does not explicitly control the order in which roles are considered for splitting and merging; and (3) GO never tries to eliminate roles.

Ene *et al.*'s role mining algorithms aim to minimize either $|R|$ or $|UA| + |PA|$ [EHM+08]. They do not consider policy interpretability with respect to user attribute data. Molloy *et al.* generalized the algorithm that aims to minimize $|UA| + |PA|$ so that it aims to minimize WSC instead, and they found that the modified algorithm performs well when the weight vector corresponds to the algorithm's original metric (i.e., when WSC equals $|UA| + |PA|$) but performs worse than GO and HM with other weight vectors [MCL+10], including the weight vectors used in our experiments.

Li *et al.*'s Dynamic Miner [LLM+07, MLL+09] has three phases. Phase 1 generates a set of candidate roles. Phase 2 selects candidate roles to include in the RBAC policy, adding them to the policy in descending order of the estimated decrease in WSC achieved by adding the role (it is an estimate because the user-role assignment and role hierarchy are not known yet). Phase 3 constructs the user-role assignment and role hierarchy. Our selection algorithm is similar to Dynamic Miner, but more general, because it is parameterized by the role quality metric that controls the order in which roles are considered for selection, and, more importantly, it allows the role quality metric to take the role hierarchy and user-role assignment into account, because they are computed during the role selection phase. Molloy *et al.* found that Dynamic Miner generally produces worse WSC than HM and GO [MLL+09]. This is consistent with our finding that the selection algorithm generally produces worse results than the elimination algorithm.

Molloy *et al.*'s Hierarchical Miner (HM) has two phases. Phase 1 uses formal concept analysis to create a candidate role hierarchy consistent with a given ACL policy; phase 1 of the elimination algorithm is equivalent to phase 1 of HM. Phase 2 eliminates roles, removes their inheritance edges, or replaces them with direct user-permission assignment when this preserves consistency with the given ACL policy and lowers the WSC. The elimination algorithm achieves slightly better results than HM in our experiments. We believe this is mainly because the elimination algorithm uses a role quality metric to control the order in which roles are considered; the order in which roles are considered in HM is not explicitly controlled and depends on implementation details of a hashset library [Mol11]. The use of a quality change tolerance and a role restoration phase also help the elimination algorithm achieve better results. Although phase 1 of HM produces a candidate role hierarchy with full inheritance, phase 2 of HM does not preserve this property; we plan to experiment with allowing similar deviations from full inheritance in the elimination algorithm, which should allow better results for policy quality. HM does not consider policy interpretability with respect to user attribute data.

Molloy *et al.*'s Attribute Miner (AM) has two phases. Phase 1 produces a set of candidate normal roles and a set of candidate attribute roles (i.e., roles whose membership is defined by an attribute expression). Phase 2 greedily selects normal roles and attribute roles for inclusion in the policy in descending order of the role's benefit-to-cost ratio, which is an estimate of the role's effect on the policy's WSCA. The elimination algorithm is more flexible than AM, since it can easily be used with any policy quality metric, and it achieves significantly better results than AM even for AM's target policy quality metric, namely, WSCA. We believe the main reason for this is that the elimination approach (i.e., repeatedly remove roles) generally yields better results than the selection approach (i.e., repeatedly add roles), as we saw in the comparison of the elimination algorithm with our selection algorithm in Section 2.4, and as noted above in the discussion of Dynamic Miner.

Colantonio *et al.* propose two metrics to measure the interpretability of roles [CDPOV09]. Their approach relies on an *activity tree*, describing the hierarchical structure of business activities (business processes), and an *organization unit tree*, describing the hierarchical structure of the organization. It also assumes knowledge of which permissions are required for each activity and of the assignment of users to organizational units. The *activity-spread* of a role measures the dispersion within the activity tree of the activities enabled by the role's permissions. The *organization-unit-spread* of a role measures the "dispersion" within the organization unit tree of the role's members. Roles with low activity-spread and low organization-unit-spread are considered to be more meaningful. These metrics are intuitively appealing and could be combined with metrics based on user attributes in our algorithms when the required information is available.

Colantonio *et al.* propose an approach to taking user attributes into account during role mining [CDV12]. They first partition the set of users based on the values of selected attributes, and then perform role mining separately for each set of users in the partition (using the corresponding slice of the *UP* relation). Note that the role mining in the second step does not explicitly consider user attributes. They propose metrics that are used to select a set of attributes that provides the most meaningful partition of the users. Their paper does not consider metrics to directly evaluate the interpretability of the resulting roles or RBAC policies.

Chapter 3

Mining Parameterized Role-Based Policies

In this chapter, we first define an expressive parameterized RBAC (PRBAC) framework that supports a simple form of ABAC. Next we present two algorithms for mining PRBAC policies from ACLs, user attributes, and permission attributes. To the best of our knowledge, it is the first policy mining algorithm for any parameterized RBAC framework or ABAC framework. We then evaluate these algorithms on three small but non-trivial case studies. Finally we discuss related work.

3.1 Parameterized RBAC (PRBAC)

PRBAC policies refer to attributes of users and permissions. User-attribute data is represented as a tuple $\langle U, A_U, f_U \rangle$, where U is a set of users, A_U is a set of user attributes, and f_U is a function such that $f_U(u, a)$ is the value of attribute a for user u. There is a special user attribute called uid that has a unique value for each user. This allows traditional identity-based roles to be represented in the same way as other roles. Similarly, permission-attribute data is represented as a tuple $\langle P, A_P, f_P \rangle$, where P is a set of permissions, A_P is a set of permission attributes, and f_P is a function such that $f_P(p, a)$ is the value of attribute a for permission p. Informally, a permission may be regarded as involving a resource and an operation, and a permission attribute may be an attribute of the resource or an attribute (i.e., argument) of the operation. There is a special permission attribute called pid that has a unique value for each permission. Let AttrVal be the set of all legal attribute values. We assume AttrVal includes a special value "$\perp$" that indicates that the value of an attribute is unknown.

Attribute expressions are used to express the sets of users and permissions associated with roles. A *conjunctive user-attribute expression* e_c is a function from user attributes A_U to $\mathrm{Set}(\mathrm{AttrVal} \setminus \{\perp\}) \cup \{\top\}$. The symbol $\top$ denotes the set of all legal values for an attribute. We say that expression e_c *uses* an attribute a if $e_c(a) \neq \top$. We refer to the set $e_c(a)$ as the conjunct for attribute a. A user u satisfies expression e_c, denoted $u \models e_c$, iff $(\forall a \in A_U : f_U(u, a) \in e_c(a))$. For example, if $A_U = \{\mathrm{dept}, \mathrm{level}\}$, the function e_c with $e_c(\mathrm{dept}) = \{\mathrm{CS}\}$ and $e_c(\mathrm{level}) = \{\mathrm{undergrad}, \mathrm{grad}\}$ is a conjunctive user-attribute

expression, which we write with syntactic sugar as dept = CS $\wedge$ level $\in$ {undergrad, grad} (note that, when $e_c(a)$ is a singleton set $\{v\}$, we may write the conjunct as $a \in \{v\}$ or $a = v$). An *user-attribute expression* is a set, representing a disjunction, of conjunctive user-attribute expressions. A user u satisfies a user attribute expression e, denoted $u \models e$, iff $(\exists e_c \in e : u \models e_c)$. The meaning of a user-attribute expression e, denoted $[\![e]\!]_U$ is the set of users that satisfy it: $[\![e]\!]_U = \{u \in U \mid u \models e\}$. We say that a user-attribute expression e *characterizes* $[\![e]\!]_U$. We say that e *uses* an attribute a if some conjunctive user-attribute expression in e uses a. The definitions of *conjunctive permission-attribute expression* and *permission-attribute expression* are similar, except using the set A_P of permission attributes instead of the set A_U of user attributes. The meaning of a permission-attribute expression e, denoted $[\![e]\!]_P$ is the set of permissions that satisfy it: $[\![e]\!]_P = \{p \in P \mid p \models e\}$.

Constraints are used to express parameterization. Traditional PRBAC frameworks use explicit role parameters to indirectly express equalities between user attributes and permissions attributes; in our framework, such equalities are expressed directly, as constraints. For example, consider the policy that the chair of a department can update the course schedule for the department. This can be expressed using explicit role parameters by introducing a role chair(dept) and using a permission assignment rule such as PA(chair(dept), ⟨write, courseSchedule(dept)⟩). In our framework, we would define a chair role with the chairs of all departments as members, with permissions to write all course schedules, and with the constraint that the user's department equals the permission's department. The constraint ensures that each member of the role gets only the appropriate permissions. Informally, attributes used in the constraint act as role parameters.

A *constraint* is a set of equalities of the form $a_u = a_p$, where a_u is a user attribute and a_p is a permission attribute. User u and permission p satisfy constraint c, denoted $u, p \models c$, if for each equality $a_u = a_p$ in c, $f_U(u, a_u) = f_P(p, a_p)$.

A *core PRBAC policy* is a tuple $\langle U, P, R\rangle$ where U is a set of users, P is a set of permissions, and R is a set of roles, each represented as a tuple $\langle e_u, e_p, c\rangle$, , where e_u is a user-attribute expression, e_p is a permission-attribute expression, and c is a constraint. For a role $r = \langle e_u, e_p, c\rangle$, let $\mathrm{uae}(r) = e_u$, $\mathrm{pae}(r) = e_p$, and $\mathrm{con}(r) = c$.

For example, the role ⟨uid = {Alice, Bob}, operation = write$\wedge$resource = courseSchedule, dept = dept⟩ has members Alice and Bob, has permissions to write course schedules for all departments (because the department attribute of the course schedule is not restricted by the permission-attribute expression), and has constraint dept = dept. If $f_U(\mathrm{Alice}, \mathrm{dept}) = \mathrm{CS}$ and $f_U(\mathrm{Bob}, \mathrm{dept}) = \mathrm{EE}$, the constraint ensures that Alice only gets permission to write the CS course schedule, and Bob only gets permission to write the EE course schedule.

The user-permission assignment UPA(π) induced by a policy π is defined by

$$\begin{aligned}
\mathrm{assignedU}(r, U) &= \{u \in U \mid u \models \mathrm{uae}(r)\}\\
\mathrm{assignedP}(r, P) &= \{p \in P \mid p \models \mathrm{pae}(r)\}\\
\mathrm{assignedUP}(r, U, P) &= \{\langle u, p\rangle \in \mathrm{assignedU}(r, U) \times \mathrm{assignedP}(r, P) \mid \\
&\qquad u, p \models \mathrm{con}(r)\}\\
\mathrm{UPA}(\langle U, P, R\rangle) &= \textstyle\bigcup_{r \in R} \mathrm{assignedUP}(r, U, P)
\end{aligned}$$

A *hierarchical PRBAC policy* is a tuple $\pi = \langle U, P, R, RH\rangle$, where U, P, and R are the

same as in a core PRBAC policy, and the role hierarchy RH is an acyclic transitive binary relation on roles. A tuple $\langle r, r' \rangle$ in RH means that r is junior to r' (or, equivalently, r' is senior to r). This means that r inherits members from r', and r' inherits permissions from r. This is captured in the equations

$$\begin{aligned}
\text{ancestors}(r, R, RH) &= \{r' \in R \mid \langle r, r' \rangle \in RH\} \\
\text{descendants}(r, R, RH) &= \{r' \in R \mid \langle r', r \rangle \in RH\} \\
\text{authU}(r, U, R, RH) &= \text{assignedU}(r, U) \cup \\
&\quad \bigcup_{r' \in \text{ancestors}(r,R,RH)} \text{assignedU}(r', U) \\
\text{authP}(r, P, R, RH) &= \text{assignedP}(r, P) \cup \\
&\quad \bigcup_{r' \in \text{descendants}(r,R,RH)} \text{assignedP}(r', P)
\end{aligned}$$

The user-permission assignment $\text{UPA}(\pi)$ induced by a hierarchical PRBAC policy π is defined by:

$$\begin{aligned}
&\text{authUP}(r, U, P, R, RH) = \\
&\quad \{\langle u, p \rangle \in \text{authU}(r, U, R, RH) \times \text{authP}(r, P, R, RH) \mid \\
&\qquad u, p \models \text{con}(r)\} \\
&\text{UPA}(\langle U, P, R, RH \rangle) = \textstyle\bigcup_{r \in R} \text{authUP}(r, U, P, R, RH)
\end{aligned}$$

In the definition of $\text{authUP}(r, U, P)$, all authorized users and permissions of r, including the inherited ones, are subject to the constraint associated with r. However, constraints are not "inherited"; in particular, the constraint associated with a role r affects only r's contribution to the user-permission relation induced by the policy.

3.2 The Problem

A core PRBAC policy $\pi = \langle U, P, R \rangle$ is *consistent* with an ACL policy $\pi' = \langle U', P', UP' \rangle$ if $U = U'$, $P = P'$, and $\text{UPA}(\pi) = UP'$. A hierarchical PRBAC policy $\pi = \langle U, P, R, RH \rangle$ is *consistent* with an ACL policy $\pi' = \langle U', P', UP' \rangle$ if $U = U'$, $P = P'$, and $\text{UPA}(\pi) = UP'$.

A *policy quality metric* is a function from PRBAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this might seem counter-intuitive at first glance but is natural for metrics based on policy size.

The *core PRBAC policy mining problem* is: given an ACL policy π' and policy quality metric Q_{pol}, find a core PRBAC policy π that is consistent with π' and has the best quality, according to Q_{pol}, among policies consistent with π'. The *hierarchical PRBAC policy mining problem* is the same except that π is a hierarchical PRBAC policy.

Our algorithms aim to optimize the policy's *weighted structural complexity* (WSC), which is a generalization of policy size [MCL+10]. The weighted structural complexity of a core PRBAC policy is defined by

$$\text{WSC}(e_c) = \sum_{a \in \text{domain}(e_c)} e_c(a) = \top \; ? \; 0 : |e_c(a)|$$

$$\begin{aligned}
\text{WSC}(c) &= |c| \\
\text{WSC}(\langle e_u, e_p, c\rangle) &= w_1 \sum_{e_c \in e_u} \text{WSC}(e_c) + w_2 \sum_{e_c \in e_p} \text{WSC}(e_c) \\
&\quad + w_3 \text{WSC}(c) \\
\text{WSC}(\langle U, P, R\rangle) &= \sum_{r \in R} \text{WSC}(r),
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights. The weighted structural complexity WSC_H of a hierarchical PRBAC policy is defined in the same way, except with an additional term $w_4|RH|$, where the size of the role hierarchy RH is the number of tuples in it.

3.3 Algorithms

This section presents our algorithms for the problems defined in Section 3.2.

3.3.1 Mining Core PRBAC Policies: Elimination Algorithm

Step 1: Generate Candidate Roles This step uses a traditional role mining algorithm to generate a set R_{can} of un-parameterized candidate roles without role hierarchy. Each role r in R_{can} is associated with a set assignedU(r) of assigned users and a set assignedP(r) of assigned permissions. We use CompleteMiner [VAW06, VAWG10] to generate candidate roles. Briefly, CompleteMiner generates a candidate role for every set of permissions that can be obtained by intersecting the sets of permissions granted to some users by the ACL policy. Note that CompleteMiner's goal is to include every reasonable candidate role in its output; CompleteMiner does not attempt to produce a minimum-sized policy.

We assume that no two candidate roles have exactly the same set of assigned users, and that no two candidate roles have exactly the same set of assigned permissions. This is true for the result of CompleteMiner and other standard role mining algorithms, because two roles with the same set of users or permissions can easily be merged into a single role.

Step 2: Generate Attribute Expressions for Candidate Roles This step computes minimum-sized attribute expressions that characterize the assigned users and assigned permissions of each candidate role, with preference given to (1) attribute expressions that do not use uid or pid, since attribute-based policies are generally preferable to identity-based policies, even when they have higher WSC, because attribute-based generalize better, and (2) conjunctive attribute expressions, because they are simpler than attribute expressions that use disjunction (in addition to conjunction).

Given a set s of users and the set U of all users, let minExpU(s, U) be a minimum-sized user-attribute expression that characterizes s, subject to the preferences described above. Given a set s of permissions and the set P of all permissions, let minExpP(s, P) be a minimum-sized permission-attribute expression that characterizes s, subject to the preferences described above. In both cases, at least one such attribute expression exists, because attributes uid and pid are present and have a unique value for each user or permission,

respectively. For each $r \in R_{can}$, this step sets $\mathrm{uae}(r) = \mathrm{minExpU}(\mathrm{assignedU}(r), U)$ and $\mathrm{pae}(r) = \mathrm{minExpP}(\mathrm{assignedP}(r), P)$.

Our algorithm to compute $\mathrm{minExpU}(s, U)$ appears in Figure 3.1; the algorithm for minExpP is the same, except that A_U and f_U are replaced with A_P and f_P, respectively. The pseudocode for minExpU simply embodies the preferences described above. It uses an auxiliary function $\mathrm{simplifyExp}(e)$ that simplifies an attribute expression e by repeatedly looking for pairs of conjunctions c_1 and c_2 in e that differ in the value of a single attribute a and replacing c_1 and c_2 with a single conjunction c that agrees with c_1 and c_2 for all attributes except a and that maps a to $c_1(a) \cup c_2(a)$.

The pseudocode for minExpU also uses an auxiliary function minConjExpU that computes a minimum-sized conjunctive user-attribute expression that characterizes s, with preference given to attribute expressions that do not use uid. The first for-loop computes a conjunctive user-attribute expression e that attempts to characterize s without using uid. If this fails, then uid is needed to characterize s, and the algorithm returns a user-attribute expression that uses only uid. Otherwise, the algorithm uses e as a starting point for computation of a minimum-sized user-attribute expression for s that does not use uid. How could a smaller user attribute expression e' for s differ from e? It cannot be that some conjunct of e' is a strict subset of the corresponding conjunct of e, because then some user in s will not satisfy that conjunct. The only way that e' could differ from e is by replacement of some conjuncts with $\top$. The second for-loop considers all expressions that differ from e in this way.

Step 3: Generate Constraints for Candidate Roles We take $\mathrm{con}(r)$ to contain every equality that holds between every assigned user and every assigned permission of r. In other words, for each attribute a_u in A_U and each attribute a_p in A_P, we add the equality $a_u = a_p$ to $\mathrm{con}(r)$ iff $\forall u \in \mathrm{assignedU}(r).\ \forall p \in \mathrm{assignedP}(r).\ u, p \models a_u = a_p$. This is the strictest constraint that can be associated with r, because any stricter constraint would incorrectly eliminate some user-permission pairs in $\mathrm{assignedUP}(r, U, P)$. Using the strictest constraint for each role facilitates merging of roles in the next step.

Step 4: Merge Candidate Roles This step creates additional candidate roles by merging sets of candidate roles. A set s of roles is *mergeable* if there exists a role r' with the same assigned users, same assigned permissions, and same or larger user-permission assignment as the roles in s collectively, and $\mathrm{assignedUP}(r', U, P) \subseteq UP'$, i.e., if there exists r' such that $\mathrm{assignedU}(r') = \bigcup_{r \in s} \mathrm{assignedU}(r)$ and $\mathrm{assignedP}(r') = \bigcup_{r \in s} \mathrm{assignedP}(r)$ and $\bigcup_{r \in s} \mathrm{assignedUP}(r, U, P) \subseteq \mathrm{assignedUP}(r', U, P) \subseteq UP'$. Assuming that all roles have distinct sets of assigned users and permissions (as mentioned in Step 1), s is mergeable only if there exists a constraint that can be associated with r' that prevents users assigned to one of the roles in s from incorrectly gaining permissions assigned to one of the other roles in s. Thus, to maximize the chance of a successful merge, we identify the strictest constraint that can be associated with r' and then check whether it prevents such gaining of permissions. The constraint associated with r' must not eliminate any user-permission assignment associated with any role in s. From Step 3, for each role r, $\mathrm{con}(r)$ is the strictest constraint that does not eliminate any user-permission assignment associated with r, so $\mathrm{con}(r')$ must

```
function minExpU(s, U):
   // compute conjunctive and
   // disjunctive user-attribute
   // expressions for s, and then
   // compare them.
1: e_c = {minConjExpU(s, U)}
2: e_d = simplifyExp(
        ⋃_{u∈s} minConjExpU(u, U))
3: if e_c does not use uid and e_d
        uses uid
4:      return e_c
5: end if
6: if e_d does not use uid and e_c
        uses uid
7:      return e_d
8: end if
9: if WSC(e_c) ≤ WSC(e_d)
10:     return e_c
11: else
12:     return e_d
13: end if
```

```
function minConjExpU(s, U):
   // check whether uid is needed to
   // characterize s in a conjunctive
   // user-attribute expression.
14: for a in A_U \ {uid}
15:     e(a) = ⋃_{u∈s} f_U(u, a)
16:     if ⊥ ∈ e(a)
17:         e(a) = ⊤
18:     end if
19: end for
20: e(uid) = ⊤
21: if [[e]]_U ≠ s
        // uid is necessary (and sufficient) to
        // characterize s in a conjunctive
        // user-attribute expression.
22:     return f_∅[uid ↦ ⋃_{u∈s} f_U(u, uid)]
23: end if
   // e characterizes s. check if there's a
   // smaller conjunctive user-attribute
   // expression that characterizes s.
24: for each non-empty subset A of A_U \ {uid}
25:     e′ = e[a ↦ ⊤ for a in A]
26:     if [[e′]]_U = s
27:         if WSC(e′) < WSC(e)
28:             e = e′
29:         end if
30:     end if
31: end for
32: return e
```

Figure 3.1: Algorithm to compute minExpU(s), where s is a set of users, and U is the set of all users. $f[x \mapsto y]$ denotes (a copy of) function f modified so that $f(x) = y$. $f_\emptyset$ denotes the empty function, i.e., the function whose domain is the empty set.

be weaker (i.e., less strict) than or equal to con(r) for each r in s, so the strictest constraint that can be associated with r' is $\mathrm{con}(r') = \bigcap_{r \in s} \mathrm{con}(r)$. If the role r' with the assigned users, assigned permissions, and constraint specified above satisfies $\mathrm{assignedUP}(r', U, P) \subseteq UP'$ (note that $\bigcup_{r \in s} \mathrm{assignedUP}(r, U, P) \subseteq \mathrm{assignedUP}(r', U, P)$ holds by construction), then s is mergeable, and we set $\mathrm{uae}(r') = \mathrm{minExpU}(\mathrm{assignedU}(r'), U)$ and $\mathrm{pae}(r') = \mathrm{minExpP}(\mathrm{assignedP}(r'), P)$ and then add r' to R_{can} (note that we leave the roles in s in R_{can}); if not, s is not mergeable. Let merge(s) denote the role r' defined above.

A simple algorithm for this step attempts to merge every subset s of R_{can}. We optimize the algorithm by exploiting a monotonicity property of merge, namely: $s \subseteq s'$ implies $\mathrm{assignedUP}(\mathrm{merge}(s), U, P) \subseteq \mathrm{UPA}(\mathrm{merge}(s'), U, P)$. Thus, if $\mathrm{UPA}(\mathrm{merge}(s)) \not\subseteq UP'$, the

```
1:  R′ = R_can
    // R_mrg contains roles produced by merging.
2:  R_mrg = {}
    // for each r in R′, remove r from R′, then attempt to merge
    // r′ with each remaining role in R′ and each role in R_mrg.
3:  for each r in R′
4:    R′ = R′ \ {r}
5:    for each r′ in R′ ∪ R_mrg
6:      r″ = merge({r, r′})
7:      if UPA(r″) ⊆ UP′
8:        R_mrg = R_mrg ∪ {r″}
9:      end if
10:   end for
11: end for
12: R_can = R_can ∪ R_mrg
```

Figure 3.2: Step 4 (Merge Candidate Roles) of elimination algorithm for core PRBAC policy mining.

algorithm does not attempt to merge supersets s' of s, because $\mathrm{UPA}(\mathrm{merge}(s')) \not\subseteq UP'$ holds and hence s' is not mergeable. The algorithm also exploits the property $\mathrm{merge}(s \cup \{r\}) = \mathrm{merge}(\{\mathrm{merge}(s), r\})$, which implies that arbitrary merges can be realized by merging in one role at a time. Pseudo-code for this step appears in Figure 3.2. The general structure of the code is similar to CompleteMiner [VAW06, VAWG10]. Our implementation incorporates another optimization, not shown in Figure 3.2. The order in which roles are merged does not affect the result, so we extend the algorithm to avoid merging the same roles in different orders. We define an arbitrary total order on roles. For each role r produced by merging, let $\mathrm{maxMerge}(r)$ be the largest role used in the merges that produced r. In line 6, if r' was produced by merging, r is merged with r' only if $r > \mathrm{maxMerge}(r')$.

Step 5 (Optional): Eliminate Unnecessary Constraints For a role r, an equality in $\mathrm{con}(r)$ is *unnecessary* if removing it from $\mathrm{con}(r)$ leaves $\mathrm{assignedUP}(r)$ unchanged. This optional step removes each unnecessary equality from the constraint of each role in R_{can}.

Informally, one cannot tell from the given input whether to include unnecessary constraints in the PRBAC policy, because they do not affect consistency with the given ACL policy. Note that these "unnecessary" constraints may have been useful during merging, because they may help to prevent user-permission assignments from growing when roles are merged, but they provide no benefit after merging. The argument in favor of removing them (after merging) is to reduce policy size and hence increase policy quality. The argument in favor of keeping them is to be more conservative from the security perspective, specifically, to minimize the risk that the policy grants to a new user a permission that the new user should not have. This is related to how well the policy generalizes.

We consider this step as optional; in other words, the user decides whether unnecessary constraints should be removed.

Step 6: Eliminate Low-Quality Removable Candidate Roles This step eliminates low-quality removable candidate roles; the remaining roles form the generated PRBAC policy.

A *role quality metric* is a function $Q(r, U, P, R)$, where r is a new role whose quality is returned, U and P are the sets of users and permissions, respectively, R is the set of existing roles, and the range is a totally-ordered set. The ordering is chosen so that large values indicate high quality (note: this is opposite to the interpretation of the ordering for policy quality metrics).

Based on our goal of minimizing the generated policy's WSC, we define a role quality metric that assigns higher quality to roles with smaller WSC that cover more uncovered user-permission pairs; "uncovered" means that the user-permission pairs are not covered by roles in R. We capture this notion of quality using the ratio $\frac{|\mathrm{uncovUP}(r,U,P,R)|}{\mathrm{WSC}(r)}$ as the first component of the role quality metric, where $\mathrm{uncovUP}(r, U, P, R)$ is the set of user-permission pairs covered by r and not covered by roles in R. Among roles with the same value of this ratio, we assign higher quality to roles that cover more user-permission pairs. We capture this by using $|\mathrm{uncovUP}(r, U, P, R)|$ as the second component of the role quality metric, and ordering values of the role quality metric lexicographically. In summary, the role quality metric Q is defined as follows.

$$\begin{aligned}
&\mathrm{uncovUP}(r, U, P, R) = \\
&\quad \mathrm{assignedUP}(r, U, P) \setminus \textstyle\bigcup_{r' \in R} \mathrm{assignedUP}(r', U, P) \\
&Q(r, U, P, R) = \langle \tfrac{|\mathrm{uncovUP}(r,U,P,R)|}{\mathrm{WSC}(r)}, |\mathrm{uncovUP}(r, U, P, R)| \rangle
\end{aligned}$$

A role r is *removable*, denoted $\mathrm{removable}(r, U, P, R)$, if every user-permission pair covered by r is also covered by another role in R. Formally,

$$\begin{aligned}
&\mathrm{removable}(r, U, P, R) = \\
&\quad \mathrm{assignedUP}(r, U, P) \subseteq \textstyle\bigcup_{r' \in R \setminus \{r\}} \mathrm{assignedUP}(r', U, P)
\end{aligned}$$

Pseudo-code for step 6 appears in Figure 3.3. In each iteration, R contains roles currently known to be in the result policy, and R_{can} contains roles that might later get added to the result policy. The algorithm evaluates removability of a role with respect to $R_{can} \cup R$ (instead of R), in order to minimize the set of roles considered unremovable; this leaves more roles in R_{can}, eligible for removal, and therefore leads to better policy quality. The algorithm evaluates role quality with respect to R, because this provides a better estimate of the role's quality in the final policy.

3.3.2 Mining Core PRBAC Policies: Selection Algorithm

Steps 1–5 of the selection algorithm for mining core PRBAC policies are the same as in the elimination algorithm for mining core PRBAC policies in Section 3.3.1. Step 6 is as follows.

Step 6: Select Roles This step selects candidate roles for inclusion in the generated policy. It selects roles from highest quality to lowest, until every pair in the user-permission relation in the given ACL policy is covered. It uses the same role quality metric as Step 6 of the elimination algorithm in Section 3.3.1. Pseudo-code for this step is as follows.

```
1:  R = ∅
2:  while UPA(⟨U, P, R⟩) ≠ UP
      // move unremovable roles from candidates R_can to result R.
3:    R_unrm = {r ∈ R_can | ¬removable(r, U, P, R_can ∪ R)}
4:    R_can = R_can \ R_unrm
5:    R = R ∪ R_unrm
      // discard the lowest-quality candidate role
6:    if ¬empty(R_can)
7:      r_min = a role in R_can with minimal quality, i.e.,
8:              ∀r ∈ R_can. Q(r_min, U, P, R) ≤ Q(r, U, P, R).
9:      R_can = R_can \ {r_min}
10:   end if
11: end while
12: return ⟨U, P, R⟩
```

Figure 3.3: Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for core PRBAC policy mining.

```
1: R = ∅
2: while UPA(⟨U, P, R⟩) ≠ UP
3:   r_max = a role in R_can with maximal quality, i.e.,
4:           ∀r ∈ R_can. Q(r_max, U, P, R) ≥ Q(r, U, P, R).
5:   R = R ∪ {r_max}
6:   R_can = R_can \ {r_max}
7: end while
8: return ⟨U, P, R⟩
```

3.3.3 Mining Hierarchical PRBAC Policies: Elimination Algorithm

Steps 1–5 of this algorithm are the same as in the core PRBAC policy mining algorithm in Section 3.3.1. The remaining steps are as follows.

Step 6: Compute Role Hierarchy This step computes all possible role hierarchy relations between candidate roles. Let $r_1 \prec r_2$ if $r_1 \neq r_2 \wedge \text{assignedP}(r_1) \subseteq \text{assignedP}(r_2) \wedge \text{assignedU}(r_1) \supseteq \text{assignedU}(r_2)$. Let RH_{all} be the transitive reduction of $\prec$.

Step 7: Generate Result Policy This step starts by storing some current information about each candidate role in auxiliary data structures. Specifically, let $\text{authU}_0(r) = \text{assignedU}(r)$ and $\text{authP}_0(r) = \text{assignedP}(r)$ and $\text{authUP}_0(r) = \text{assignedUP}(r)$. Note that the assigned users and permissions might change as we generate the hierarchical policy, because some assigned users and permissions might become inherited instead. In contrast, the authorized users and permissions of each role r never change, always remaining

equal to $\text{authU}_0(r)$ and $\text{authP}_0(r)$, respectively. Similarly, $\text{assignedUP}(r)$ might change, but $\text{authUP}(r)$ always remains equal to $\text{authUP}_0(r)$.

Our algorithm always generates policies with full inheritance [XS12]. This implies that a role hierarchy edge in RH_{all} is included in the result policy whenever the roles that it connects are included in the policy. Therefore, we associate with each candidate role the cost (WSC) of the edges that will be added to the policy if that role is added. We define a size metric on roles that reflects this: for a role r in R_{can}, and a set R of roles that have already been selected to be in the result policy,

$$\begin{aligned}&\text{sizeof}(r, R, RH_{\text{all}}) = \\ &\quad \text{WSC}(r) + \tfrac{w_4}{2}|\{r' \in R \mid \langle r, r' \rangle \in RH_{\text{all}} \vee \langle r', r \rangle \in RH_{\text{all}}\}|\end{aligned}$$

The coefficient $\frac{w_4}{2}$ (recall that w_4 is introduced in Section 3.2) reflects that half of the cost of each inheritance relationship is attributed to each of the involved roles.

We define a role quality metric $Q_{\text{H}}(r, R, RH)$ similar to the metric in the non-hierarchical case, except using sizeof instead of WSC and using $\text{authUP}_0(r)$ instead of $\text{assignedUP}(r)$.

$$\begin{aligned}\text{uncovUP}_{\text{H}}(r, R) &= \text{authUP}_0(r) \setminus \bigcup_{r' \in R} \text{authUP}_0(r') \\ Q_{\text{H}}(r, R, RH) &= \frac{|\text{uncovUP}_{\text{H}}(r, R)|}{\text{sizeof}(r, RH)}\end{aligned}$$

We define a function removable similar to the one in the non-hierarchical case, except using $\text{authUP}_0(r)$ instead of $\text{assignedUP}(r)$.

$$\text{removable}_{\text{H}}(r, R) = \text{authUP}_0(r) \subseteq \bigcup_{r' \in R \setminus \{r\}} \text{authUP}_0(r')$$

Pseudo-code for this step appears in Figure 3.5. It is similar to the pseudo-code in Figure 3.3 for Step 6 of the elimination algorithm for mining core PRBAC policies. The main difference is that this algorithm calls minExpU and minExpP to update $\text{uae}(r)$ and $\text{pae}(r)$, respectively, after roles have been added to the set R of roles that will be included in the generated policy. This reflects the fact that, in the presence of role hierarchy, we are free to choose $\text{assignedU}(r)$ in a way that minimizes the WSC of the policy, provided $\text{authU}(r, U, R, RH)$ remains equal to $\text{authU}_0(r)$, and similarly for $\text{assignedP}(r)$.

$\text{minExpU}_{\text{H}}(r, U, R, RH)$ returns a minimum-sized user attribute expression for r that excludes users that can be inherited from other roles in R along edges in RH if excluding those users reduces $\text{WSC}(\text{uae}(r))$; this is legitimate, because the sets of users assigned to and inherited by a role may overlap. Let $\text{inheritedU}(r, R, RH)$ denote the set of users that r inherits, i.e.,

$$\text{inheritedU}(r, R, RH) = \bigcup_{r' \in \text{ancestors}(r, R, RH)} \text{authU}_0(r').$$

Ideally, $\text{minExpU}_{\text{H}}(r, U, R, RH)$ would find a subset s of $\text{inheritedU}(r)$ that minimizes $\text{WSC}(\text{minExpU}(\text{authU}_0(r) \setminus s))$ and return $\text{minExpU}(\text{authU}_0(r) \setminus s)$. In practice, trying all subsets s of $\text{inheritedU}(r, R, RH)$ would be too slow. An obvious heuristic approximation is to try only the extrema—in other words, only $s = \emptyset$ and $s = \text{inheritedU}(r, R, RH)$.

We adopt a heuristic approximation, shown in Figure 3.4 that is somewhat more thorough and correspondingly more expensive: it is exponential in the number of attributes, but polynomial in the number of users. Similar to our algorithm for minExpU in Figure 3.1, it starts by constructing an upper-bound expression e (for $\mathrm{authU}_0(r)$) without using uid and then considers the expressions obtained setting to $\top$ the conjuncts of e corresponding to each subset of the attributes. However, instead of simply checking whether the resulting expression now denotes a larger set or still denotes the same set (namely, $\mathrm{authU}_0(r)$) as in Fig 3.1, it exploits the flexibility that the expression may characterize any set between $\mathrm{authU}_0(r) \setminus \mathrm{inheritedU}(r)$ and $\mathrm{authU}_0(r)$, by removing values from conjuncts of e (in order to make the denoted set smaller, partially counteracting the effect of setting some conjuncts to $\top$), provided the resulting expression still represents a superset of $\mathrm{authU}_0(r) \setminus \mathrm{inheritedU}(r)$, and then checks whether the resulting expression characterizes a set in the required range. If this fails to produce an expression representing a set in the required range, then an expression using uid is constructed.

$\mathrm{minExpP_H}(r, P, R, RH)$ is defined similarly, except using $\mathrm{inheritedP}(r, R, RH)$ instead of $\mathrm{inheritedU}(r, R, RH)$, where $\mathrm{inheritedP}(r, R, RH) = \bigcup_{r' \in \mathrm{descendants}(r,R,RH)} \mathrm{authP}_0(r')$.

Because we minimize $\mathrm{WSC}(\mathrm{uae}(r))$ and $\mathrm{WSC}(\mathrm{pae}(r))$ instead of $\mathrm{assignedU}(r)$ and $\mathrm{assignedP}(r)$, some inheritance relationships might become useless, if the users and permissions inherited by a role r through those relationships are also in $\mathrm{assignedU}(r)$ and $\mathrm{assignedP}(r)$, respectively. Such inheritance relationships could be eliminated without changing $\mathrm{authUP}(r)$. We leave such relationships in the policy, because we want to generate policies with complete inheritance, as mentioned above. To illustrate the benefits of this approach, consider a problem instance in which there are user attributes indicating which users are employees (e.g., isEmployee = true) and which users are faculty (e.g., position = faculty), and that all faculty are employees. Suppose role mining produces roles corresponding to employee and faculty. If the assigned users of the employee role are characterized by isEmployee = true, then users in the faculty role are assigned users of the employee role, so an inheritance relationship between these roles is useless and could be eliminated, but this inheritance relationship is semantically meaningful and natural, so it is better to keep it in the policy.

3.3.4 Mining Hierarchical PRBAC Policies: Selection Algorithm

Steps 1–5 of this algorithm are the same as in the elimination algorithm for mining hierarchical PRBAC policies in Section 3.3.3. Step 6 is as follows.

Step 6: Select Roles Pseudo-code for this step appears in Figure 3.6. It is similar to the pseudocode for Step 6 of the selection algorithm for mining core PRBAC policies in Section 3.3.2. The main differences are the addition of the for-loop to adjust the user-attribute expressions and permission-attribute expressions of previously selected roles when another role is selected, and the addition of the call to finalizePolicy at the end.

3.3.5 Complexity Analysis

This complexity analysis applies to all four of the above algorithms. The running time of CompleteMiner in Step 1 is worst-case exponential in $|P|$ but acceptable in practice, based

```
    function minExpU_H(r, U, R, RH):
    // try to construct an expression representing a set
    // in the required range, without using uid
1:  for a in A_U \ {uid}
2:      e(a) = ⋃_{u∈authU_0(r)} f_U(u, a)
3:      if ⊥ ∈ e(a)
4:          e(a) = ⊤
5:      end if
6:  end for
7:  e(uid) = ⊤
8:  for each non-empty subset A of A_U
9:      e′ = e[a ↦ ⊤ for a in A]
10:     // try to remove values from conjuncts of e′
11:     for a in A_U \ A
12:         for v in e(a)
13:             e″ = e′[a ↦ e′(a) \ {v}]
14:             if isInRange(e″, r, R, RH)
15:                 e′ = e″
16:             end if
17:         end for
18:     end for
19:     if isInRange(e′, r, R, RH) ∧ WSC(e′) < WSC(e)
20:         e = e′
21:     end if
22: end for
23: if isInRange(e, r, R, RH)
24:     return e
25: end if
    // uid is need to represent a set in the required range. Choose
    // the smallest set in that range, to get the smallest expression.
26: return f_∅[uid ↦ ⋃_{u∈authU_0(r)\inheritedU(r,R,RH)} f_U(u, uid)]

    // check whether [[e]]_U is in the required range
    function isInRange(e,r,R,RH):
27: return authU_0(r) \ inheritedU(r, R, RH) ⊆ [[e]]_U ∧ [[e]]_U ⊆ authU_0(r)
```

Figure 3.4: Algorithm for $\mathrm{minExpU_H}(r, U, R, RH)$.

on our experience applying it to small inputs in this work and larger inputs in previous work [XS12]. Let $R_{can}(i)$ denote the value of R_{can} after Step i; note that $|R_{can}(i)|$ is worst-case exponential in the size of the input policy. The running time of Step 2 is $O(|R_{can}(1)| \times (2^{|A_U|} + 2^{|A_P|}))$. The running time of Step 3 is $O(|R_{can}(2)| \times |A_U| \times |A_P|)$. The running time of Step 4 is $O(2^{|R_{can}(3)|})$, since every subset of $R_{can}(3)$ is explored in the worst case; however, the optimizations in Step 4 greatly reduce the number of explored subsets in our case studies.

1: $R = \emptyset$
2: **while** $\text{UPA}(\langle U, P, R, RH\rangle) \neq UP$
    // move unremovable roles from candidates R_{can} to result R.
3: $R_{unrm} = \{r \in R_{can} \mid \neg\text{removable}_{\text{H}}(r, R_{can} \cup R)\}$
4: $R_{can} = R_{can} \setminus R_{unrm}$
5: $R = R \cup R_{unrm}$
    // update uae and pae of candidate roles, based on updated R
6: **for** r **in** R_{can}
7: $\text{uae}(r) = \text{minExpU}_{\text{H}}(r, R, RH_{\text{all}})$
8: $\text{pae}(r) = \text{minExpP}_{\text{H}}(r, R, RH_{\text{all}})$
9: **end for**
    // discard the lowest-quality candidate role
10: **if** $\neg\text{empty}(R_{can})$
11: $r_{\min}$ = a role in R_{can} with minimal quality, *i.e.*,
12: $\forall r \in R_{can}.\ Q_{\text{H}}(r_{\min}, R, RH) \leq Q_{\text{H}}(r, R, RH)$.
13: $R_{can} = R_{can} \setminus \{r_{\min}\}$
14: remove tuples containing $r_{\min}$ from RH_{all}
15: **end if**
16: **end while**
17: finalizePolicy(R, RH_{all})

function finalizePolicy(R, RH_{all}):
// adjust uae and pae of roles in policy, based on final role hierarchy, to reduce WSC.
18: **for** r **in** R
19: $\text{uae}(r) = \text{minExpU}_{\text{H}}(r, R, RH_{\text{all}})$
20: $\text{pae}(r) = \text{minExpP}_{\text{H}}(r, R, RH_{\text{all}})$
21: **end for**
22: $RH = \{\langle r, r'\rangle \in RH_{\text{all}} \mid r \in R \wedge r' \in R\}$
23: **return** $\langle U, P, R, RH\rangle$

Figure 3.5: Step 6 (Eliminate Low-Quality Removable Candidate Roles) of elimination algorithm for hierarchical PRBAC policy mining.

Steps 5, 6, and 7 are polynomial in $|R_{can}(4)|$, $|R_{can}(5)|$, and $|R_{can}(6)|$, respectively.

3.4 Case Studies

This section describes the PRBAC policies we developed as case studies to illustrate our policy language and evaluate our algorithms.

The policies are written in a concrete syntax with the following kinds of statements. $\text{uae}(r, e)$ associates conjunctive user-attribute expression e with role r. $\text{pae}(r, e)$ associates conjunctive permission-attribute expression e with role r. The overall user attribute expression associated with role r is the disjunction of the expressions in the uae statements for r;

```
1:  R = ∅
2:  while UPA(⟨U, P, R, RH⟩) ≠ UP
      // select the highest quality candidate role
3:    r_max = a role in R_can with maximal quality, i.e.,
4:          ∀r ∈ R_can. Q_H(r_max, R, RH_all) ≥ Q_H(r, R, RH_all).
5:    R = R ∪ {r_max}
6:    R_can = R_can \ {r_max}
      // update uae and pae of candidate roles, based on updated R
7:    for r in R_can
8:      uae(r) = minExpU_H(r, U, R, RH_all)
9:      pae(r) = minExpP_H(r, U, R, RH_all)
10:   end for
11: end while
12: finalizePolicy(R, RH_all)
```

Figure 3.6: Step 6 (Select Roles) of selection algorithm for hierarchical PRBAC policy mining.

similarly for the permission-attribute expression. $\text{con}(r, c)$ associates constraint c with role r. $\text{rh}(r, r')$ means that r is junior to r'. $\text{userAttrib}(u, a_1 = v_1, a_2 = v_2, \ldots)$ means that, for user u, attribute a_1 has value v_1, attribute a_2 has value v_2, etc.; uid $= u$ is implicit. $\text{permAttrib}(p, a_1 = v_1, a_2 = v_2, \ldots)$ is the analogous statement for permissions.

In each policy, we included only a few users in each "role instance", e.g., two or three users in each department. This provides sufficient data for the algorithm to discover the patterns, i.e., the parameterization. Increasing the number of users in each role instance only helps the algorithm, by providing stronger evidence for each pattern.

These case studies are small in size but non-trivial in structure. They includes roles with membership specified using uid, roles with membership specified using other attributes, roles with overlapping membership, roles with disjoint membership, roles with multiple pae statements, roles with constraints with multiple conjuncts, linear role hierarchy, diamond-shaped role hierarchy, etc.

University Case Study Our university case study controls access to gradebooks and course schedules. The policy appears in Figure 3.7, except that most of the userAttrib and permAttrib statements are omitted, to save space. For convenience, we give users names such as csStu1 and eeStu1, instead of Alice and Bob. User attributes include: position (student, faculty, or staff), dept (the user's academic or administrative department), crsTaken (course number of course being taken by a student; to keep the example small, we assume the student is taking at most one course, and it is in the student's department), crsTaught (course number of course taught by a faculty or TA; same assumptions as for crsTaken). Permission attributes include: resource (resource to which the operation is applied), operation (requested operation), dept (department to which the resource belongs), crsNum (number of the course that the resource is for), and student (student whose scores are read, for operation=readScoreStudent). The conjunct crsTaken=crsNum in the constraint for the

Student role ensures that students can apply the readScoreStudent operation only to courses the student is taking. This is not essential, but it is natural and is advisable according to the defense-in-depth principle.

```
// 1. Student Role
uae(Student, position=student)
// Student can read his own scores
// in gradebook for course he is
// taking.
pae(Student,
    operation=readScoreStudent
    and resource=gradebook)
con(Student, dept=dept and
    crsTaken=crsNum and
    uid=student)

// 2. Teaching Assistant (TA) Role
uae(TA, uid in {csStu2, eeStu2,
    csStu3, eeStu3})
// TA can add and read scores for
// any student in gradebook for
// course he/she is teaching.
pae(TA, operation in {addScore,
    readScore} and
    resource=gradebook)
con(TA, dept=dept and
    crsTaught=crsNum)

// 3. Instructor Role
uae(Instructor, uid in {csFac1,
    csFac2, eeFac1, eeFac2})
// Instructor can change a score
// and assign a course grade in
// gradebook for course he/she
// is teaching.
pae(Instructor, operation in
    {changeScore, assignGrade} and
    resource=gradebook)
con(Instructor, dept=dept and
    crsTaught=crsNum)
rh(TA, Instructor)

// 4. Department Chair Role
uae(Chair, uid in {csChair, eeChair})
// Chair can read and write course
// schedule for his/her department.
pae(Chair, operation in {read, write}
    and resource=courseSchedule)
// Chair can assign grades for courses
// in his/her department.
pae(Chair, operation=assignGrade
    and resource=gradebook)
con(Chair, dept=dept)

// 5. Registrar Role
uae(Registrar, dept=registrar)
// Staff in registrar's office can modify
// course schedules for all departments.
pae(Registrar, operation=write and
    resource=courseSchedule)

// User Attribute Data. The userAttrib
// statement for one user is shown here.
// The full policy contains 19 users.
userAttrib(csStu1, position=student,
    dept=cs, crsTaken=101)

// Permission Attribute Data.
// The permAttrib statement for
// one permission is shown here.
// The full policy contains 26 permissions.
permAttrib(cs101addScore, dept=cs,
    crsNum=101, operation=addScore,
    resource=gradebook)
```

Figure 3.7: University case study

Healthcare Case Study Our healthcare case study controls access to items in electronic health records. The policy appears in Figure 3.8, except that most of the userAttrib and

permAttrib statements are omitted, to save space. User attributes include: position (doctor or nurse; for other users, this attribute equals $\perp$); ward (the ward a patient or nurse is in), specialty (the medical specialty of a doctor), team (the medical team a doctor is in), and agentFor (the patient for which a user is an agent). Permissions for access to a health record have resource=HR ("HR" is short for "health record"). Other attributes of permissions for health records include: operation (the requested operation), patient (the patient that the HR is for), topic (the medical specialty to which the HR item is related), treatingTeam (the team of doctors treating the patient the HR is for), and ward (the ward housing the patient that the HR is for).

Engineering Department Case Study Our engineering department case study controls access to project-related documents. It is based on the running example in [SBM99]. The policy appears in Figure 3.9, except that most of the userAttrib and permAttrib statements are omitted, to save space. The role hierarchy is a lattice: it has a diamond shape. User attributes include: dept (the user's department), project (the project the user is involved in; to keep the example small, we assume the user is involved in at most one project, and it is in the user's department), and specialty (the user's specialty, e.g., testing). Permission attributes include: resource (resource to which the operation is applied), operation (requested operation), dept (department to which the resource belongs), and project (project to which the resource belongs).

3.5 Evaluation

This section describes an evaluation of the effectiveness of our algorithms, based on the case studies in Section 3.4. For each case study, we generated an equivalent ACL policy and an attribute data file from the PRBAC policy, ran our hierarchical PRBAC policy mining algorithms on the resulting ACL policy and attribute data, and then compared the generated PRBAC policy to the original PRBAC policy.

The same methodology could be applied starting with synthetic (i.e., pseudo-randomly generated) PRBAC policies. We did not do this, for two reasons. First, it is difficult to generate "realistic" synthetic policies, so effectiveness of our algorithm on synthetic policies might not be representative of its effectiveness on real policies. Second, it is difficult to evaluate the effectiveness of our algorithms on synthetic policies: in case of differences between the synthetic policy and the mined policy, there would be no basis for determining which one is better (for example, the synthetic policy might be unnecessarily complicated, and the mined policy might be better). We could determine which policy has lower WSC, but minimizing WSC is just a heuristic aimed at helping the algorithm discover high-level structure, and we do not know what the best high-level structure is for synthetic policies. Ideally, we would evaluate the algorithms on access control policies in actual use, but we do not know of any publicly available deployed access control policies with accompanying attribute data.

In summary, for all three case studies, the selection algorithm for mining hierarchical PRBAC policies, without optional Step 5 (Eliminate Unnecessary Constraints), successfully reconstructs the original PRBAC policy from the ACLs and attribute data.

```
// 1. Nurse Role
uae(Nurse, position=nurse)
// Nurse can read and add HR
// items with topic=general for
// patients in his/her ward.
pae(Nurse, resource=HR and
    operation in {readItem, addItem}
    and topic=general)
con(Nurse, ward=ward)

// 2. Doctor Role
uae(Doctor, position=doctor)
// Doctor can read and add HR
// items related to his specialty
// for patients being treated
// by his/her team.
pae(Doctor, resource=HR and
    operation in {readItem, addItem})
con(Doctor, team=treatingTeam
    and specialty=topic)

// 3. Patient Role
uae(Patient, uid in {oncPat1,
    oncPat2, carPat1, carPat2})
// A patient can read and add items
// with topic=patientNote in his/her
// HR.
pae(Patient, resource=HR and
    operation in
    {readItem, addItem} and
    topic=patientNote)
con(Patient, uid=patient)

// 4. Agent Role
uae(Agent, uid in {agent1, agent2})
// Agent can add an item with
// topic=agentNote in HR for
// patient whose agent he/she is.
pae(Agent, resource=HR and
    operation=addItem and
    topic=agentNote)
// Agent can read an item with topic
// patientNote or agentNote in HR for
// patient whose agent he/she is.
pae(Agent, resource=HR and
    operation=readItem and
    topic in {patientNote, agentNote})
con(Agent, agentFor=patient)

// User Attribute Data. The userAttrib
// statement for one user is shown here;
// the full policy contains 14 users.
userAttrib(oncNurse1, position=nurse,
    ward=oncWard)

// Permission Attribute Data.
// The permAttrib statement for
// one permission is shown here;
// the full policy contains 24
// permissions.
permAttrib(rdOncItemOncPat1,
    resource=HR, operation=readItem,
    patient=oncPat1, topic=oncology,
    treatingTeam=oncTeam1,
    ward=oncWard)
```

Figure 3.8: Healthcare case study

We implemented the algorithms in Java and ran them on a laptop with an Intel Core i3 2.13 GHz CPU. In our experiments, all weights w_i in the definition of WSC are equal to 1. Table 3.10 shows, for each case study, several size metrics and the running times of both algorithms. The "$|R_{can}|$" column contains the size of R_{can} after Step 4. The "Time" column contains the running times (in seconds) for the elimination and selection algorithms, respectively, for mining hierarchical PRBAC policies and including the optional Step 5. We also measured the running time of each step. In all cases except one, Step 4 is the most expensive step; the one exception is the elimination algorithm on the university case study,

```
// 1. Engineer Role
// In this example, all users are
// engineers.
uae(Engineer, true)
// Engineer can read the project
// plan and test plan for the project
// he/she is working on.
pae(Engineer, operation=read and
    resource in {projectPlan,
    testPlan})
con(Engineer, dept=dept and
    project=project)

// 2. ProductionEngineer Role
uae(ProductionEngineer,
    specialty=production)
// Production Engineer can write
// the project plan for the project
// he/she is working on.
pae(ProductionEngineer,
    operation=write and
    resource=projectPlan)
con(ProductionEngineer, dept=dept
    and project=project)
rh(Engineer, ProductionEngineer)

// 3. QualityEngineer Role
uae(QualityEngineer,
    specialty=testing)
// Quality Engineer can write the
// test plan for the project he/she
// is working on.
pae(QualityEngineer, operation=write
    and resource=testPlan)
con(QualityEngineer, dept=dept
    and project=project)
rh(Engineer, QualityEngineer)

// 4. ProjectLead Role
uae(ProjectLead, specialty=management)
// Project Lead can create a budget for
// the project he/she is leading.
pae(ProjectLead, operation=create
    and resource=budget)
con(ProjectLead, dept=dept and
    project=project)
rh(ProductionEngineer, ProjectLead)
rh(QualityEngineer, ProjectLead)

// User Attribute Data. The userAttrib
// statement for one user is shown here;
// the full policy contains 14 users.
userAttrib(qe1, dept=ads,
    project=alpha, specialty=testing)

// Permission Attribute Data.
// The permAttrib statement for
// one permission is shown here;
// the full policy contains 10 permissions.
permAttrib(rpa1, dept=ads,
    project=alpha, operation=read,
    resource=projectPlan)
```

Figure 3.9: Engineering department case study

Case Study	$\|U\|$	$\|P\|$	$\|UP\|$	$\|A_U\|$	$\|A_P\|$	$\|R_{can}\|$	$\|R\|$	Time
university	19	26	42	4	5	203	5	2.1 1.2
healthcare	14	24	42	5	6	42	4	.55 .43
eng. dept.	14	10	42	3	4	24	4	.21 .19

Figure 3.10: Running times and size metrics for case studies.

for which Step 7 is the most expensive step.

Results of university case study We ran the elimination and selection algorithms on ACLs and attribute data generated from the university case study. Without Step 5 (Eliminate Unnecessary Constraints), the selection algorithm reconstructs the original PRBAC policy. The elimination algorithm does slightly worse, producing two roles, corresponding to TAs for CS101 and CS601, instead of a single parameterized TA role. With Step 5 (Eliminate Unnecessary Constraints), the output of the elimination algorithm stays the same, and the output of the selection algorithm becomes the same as the output of the elimination algorithm.

Results of healthcare case study We ran the elimination and selection algorithms on ACLs and attribute data generated from the healthcare case study. Without Step 5 (Eliminate Unnecessary Constraints), both algorithms reconstruct the original PRBAC policy. With Step 5 (Eliminate Unnecessary Constraints), the elimination algorithm still reconstructs the original PRBAC policy, but the selection algorithm does slightly worse, producing two roles, corresponding to cardiologists and oncologists, instead of a single parameterized Doctor role.

Results of engineering department case study We ran the elimination and selection algorithms on ACLs and attribute data generated from the engineering department case study. The selection algorithm reconstructs the original PRBAC policy. The elimination algorithm reconstructs the ProductionEngineer and ProjectLead roles, but each of the other two roles in the resulting policy contain some general engineers and some quality engineers. For both algorithms, the results are unaffected by Step 5 (Eliminate Unnecessary Constraints).

Limitations Our algorithm does not reconstruct the original policy for some variants of the health care case study, because CompleteMiner does not generate the candidate roles that need to be merged to produce the original roles. For example, suppose we modify the policy so that a patient's agent has all permissions of that patient, plus some agent-specific permissions. As a result, the agent's permissions are a superset of the patient's permissions, and the roles generated by CompleteMiner all have the property that, if the role contains the patient, then it also contains the agent. This prevents subsequent steps of the algorithm from discovering a parameterized patient role, because different constraints are needed for patients and agents, as one can see from the patient and Agent roles in Figure 3.8. To overcome this limitation, Step 1 should be extended to take attribute information into account when generating candidate roles.

3.6 Related Work

We are not aware of any prior work on policy mining for PRBAC or ABAC. Our policy mining algorithms build on two pieces of prior work on role mining for RBAC: Vaidya, Atluri, and

Warner's CompleteMiner algorithm for generating candidate roles [VAW06, VAWG10], and Xu and Stoller's elimination and selection algorithms for deciding which candidate roles to include in the final policy [XS12]. The novel part of our algorithms are the middle steps, in which constraint generation and role merging are used to discover parameterization.

Xu and Stoller's elimination algorithm is partly inspired by Molloy *et al.*'s Hierarchical Miner algorithm for mining roles with semantic meaning based on user-attribute data [MCL+10]. Colantonio *et al.* developed a different method for taking user-attribute data into account during role mining; their method partitions the set of users based on the values of selected attributes, and then performs role mining separately for each of the resulting sets of users [CDV12].

We use role quality and policy quality metrics based on weighted structural complexity [MCL+10]. Other role quality and policy quality metrics have been proposed. Colantonio *et al.* proposed metrics that measure how well roles fit the hierarchical structures of an organization and its business processes [CDPOV09]. These metrics could be incorporated in our algorithm. Qi *al.* proposed a metric for optimality of role hierarchies and an efficient heuristic algorithm for mining role hierarchies based on that metric [GVA08]. Their work could be extended to accommodate parameters and combined with our approach to discovering parameterized roles.

Several access control frameworks that support some form of parameterized roles have been proposed. The earliest ones are by Giuri and Iglio [GI97] and Lupu and Sloman [LS97]; the role templates and policy templates, respectively, in these frameworks support parameterized roles. More recently, Ge and Osborn [GO04] and Li and Mao [LM07] proposed RBAC frameworks with parameterized roles. The most visible difference between parameterization in these frameworks and ours is that role parameters are explicit in these frameworks but implicit in ours. However, this difference is more superficial than significant: our approach to PRBAC policy mining can be adapted to PRBAC frameworks with explicit parameters.

Chapter 4

Mining Attribute-based Access Control Policies from ACLs

In this chapter, we first define an expressive ABAC framework that contains all of the common ABAC policy language constructs. Next we present an algorithm for mining ABAC policies from ACLs, user attributes, and resource attributes. To the best of our knowledge, it is the first policy mining algorithm for any ABAC framework. We also describe extensions of the algorithm to identify suspected noise in the input data. We then evaluate our algorithm on several manually written case studies and randomly generated synthetic policies of varying size. We discuss related work at the end of this chapter.

4.1 ABAC policy language

This section presents our ABAC policy language. We do not consider policy administration, since our goal is to mine a single ABAC policy from the current low-level policy. We present a specific concrete policy language, rather than a flexible framework, to simplify the exposition and evaluation of our policy mining algorithm, although our approach is general and can be adapted to other ABAC policy languages. Our ABAC policy language contains all of the common ABAC policy language constructs, except arithmetic inequalities and negation. Extending our algorithm to handle those constructs is future work. The policy language handled in this paper is already significantly more complex than policy languages handled in previous work on security policy mining.

ABAC policies refer to attributes of users and resources. Given a set U of users and a set A_{u} of user attributes, user attribute data is represented by a function d_{u} such that $d_{\mathrm{u}}(u, a)$ is the value of attribute a for user u. There is a distinguished user attribute uid that has a unique value for each user. Similarly, given a set R of resources and a set A_{r} of resource attributes, resource attribute data is represented by a function d_{r} such that $d_{\mathrm{r}}(r, a)$ is the value of attribute a for resource r. There is a distinguished resource attribute rid that has a unique value for each resource. We assume the set A_{u} of user attributes can be partitioned into a set $A_{\mathrm{u},1}$ of *single-valued user attributes* which have atomic values, and a set $A_{\mathrm{u,m}}$ of *multi-valued user attributes* whose values are sets of atomic values. Similarly, we assume the set A_{r} of resource attributes can be partitioned into a set $A_{\mathrm{r},1}$ of *single-valued*

resource attributes and a set of $A_{\mathrm{r,m}}$ of *multi-valued resource attributes*. Let $\mathrm{Val_s}$ be the set of possible atomic values of attributes. We assume $\mathrm{Val_s}$ includes a distinguished value $\bot$ used to indicate that an attribute's value is unknown. The set of possible values of multi-valued attributes is $\mathrm{Val_m} = \mathrm{Set}(\mathrm{Val_s} \setminus \{\bot\}) \cup \bot$, where $\mathrm{Set}(S)$ is the powerset of set S.

Attribute expressions are used to express the sets of users and resources to which a rule applies. A *user-attribute expression* (UAE) is a function e such that, for each user attribute a, $e(a)$ is either the special value $\top$, indicating that e imposes no constraint on the value of attribute a, or a set (interpreted as a disjunction) of possible values of a excluding $\bot$ (in other words, a subset of $\mathrm{Val_s} \setminus \{\bot\}$ or $\mathrm{Val_m} \setminus \{\bot\}$, depending on whether a is single-valued or multi-valued). We refer to the set $e(a)$ as the *conjunct* for attribute a. We say that expression e *uses* an attribute a if $e(a) \neq \top$. Let $\mathrm{attr}(e)$ denote the set of attributes used by e. Let $\mathrm{attr_1}(e)$ and $\mathrm{attr_m}(e)$ denote the sets of single-valued and multi-valued attributes, respectively, used by e.

A user u *satisfies* a user-attribute expression e, denoted $u \models e$, iff $(\forall a \in A_{\mathrm{u,1}}.\, e(a) = \top \vee \exists v \in e(a).\, d_{\mathrm{u}}(u,a) = v)$ and $(\forall a \in A_{\mathrm{u,m}}.\, e(a) = \top \vee \exists v \in e(a).\, d_{\mathrm{u}}(u,a) \supseteq v)$. For multi-valued attributes, we use the condition $d_{\mathrm{u}}(u,a) \supseteq v$ instead of $d_{\mathrm{u}}(u,a) = v$ because elements of a multi-valued user attribute typically represent some type of capabilities of a user, so using $\supseteq$ expresses that the user has the specified capabilities and possibly more.

For example, suppose $A_{\mathrm{u,1}} = \{\mathrm{dept}, \mathrm{position}\}$ and $A_{\mathrm{u,m}} = \{\mathrm{courses}\}$. The function e_1 with $e_1(\mathrm{dept}) = \{\mathrm{CS}\}$ and $e_1(\mathrm{position}) = \{\mathrm{grad}, \mathrm{undergrad}\}$ and $e_1(\mathrm{courses}) = \{\{\mathrm{CS101}, \mathrm{CS102}\}\}$ is a user-attribute expression satisfied by users in the CS department who are either graduate or undergraduate students and whose courses include CS101 and CS102 (and possibly other courses).

We introduce a concrete syntax for attribute expressions, for improved readability in examples. We write a user attribute expression as a conjunction of the conjuncts not equal to $\top$. Suppose $e(a) \neq \top$. Let $v = e(a)$. When a is single-valued, we write the conjunct for a as $a \in v$; as syntactic sugar, if v is a singleton set $\{s\}$, we may write the conjunct as $a = s$. When a is multi-valued, we write the conjunct for a as $a \supseteq\!\in v$ (indicating that a is a superset of an element of v); as syntactic sugar, if v is a singleton set $\{s\}$, we may write the conjunct as $a \supseteq s$. For example, the above expression e_1 may be written as $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{undergrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq \{\mathrm{CS101}, \mathrm{CS102}\}$. For an example that uses $\supseteq\!\in$, the expression e_2 that is the same as e_1 except with $e_2(\mathrm{courses}) = \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$ may be written as $\mathrm{dept} = \mathrm{CS} \wedge \mathrm{position} \in \{\mathrm{undergrad}, \mathrm{grad}\} \wedge \mathrm{courses} \supseteq\!\in \{\{\mathrm{CS101}\}, \{\mathrm{CS102}\}\}$, and is satisfied by graduate or undergraduate students in the CS department whose courses include either CS101 or CS102.

The *meaning* of a user-attribute expression e, denoted $[\![e]\!]_U$, is the set of users in U that satisfy it: $[\![e]\!]_U = \{u \in U \mid u \models e\}$. User attribute data is an implicit argument to $[\![e]\!]_U$. We say that e *characterizes* the set $[\![e]\!]_U$.

A *resource-attribute expression* (RAE) is defined similarly, except using the set A_{r} of resource attributes instead of the set A_{u} of user attributes. The semantics of RAEs is defined similarly to the semantics of UAEs, except simply using equality, not $\supseteq$, in the condition for multi-valued attributes in the definition of "satisfies", because we do not interpret elements of multi-valued resource attributes specially (e.g., as capabilities).

In ABAC policy rules, constraints are used to express relationships between users and resources. An *atomic constraint* is a formula f of the form $a_{\mathrm{u,m}} \supseteq a_{\mathrm{r,m}}$, $a_{\mathrm{u,m}} \ni a_{\mathrm{r,1}}$, or

$a_{\mathrm{u,1}} = a_{\mathrm{r,1}}$, where $a_{\mathrm{u,1}} \in A_{\mathrm{u,1}}$, $a_{\mathrm{u,m}} \in A_{\mathrm{u,m}}$, $a_{\mathrm{r,1}} \in A_{\mathrm{r,1}}$, and $a_{\mathrm{r,m}} \in A_{\mathrm{r,m}}$. The first two forms express that user attributes contain specified values. This is a common type of constraint, because user attributes typically represent some type of capabilities of a user. Other forms of atomic constraint are possible (e.g., $a_{\mathrm{u,m}} \subseteq a_{\mathrm{r,m}}$) but less common, so we leave them for future work. Let $\mathrm{uAttr}(f)$ and $\mathrm{rAttr}(f)$ refer to the user attribute and resource attribute, respectively, used in f. User u and resource r *satisfy* an atomic constraint f, denoted $\langle u, r\rangle \models f$, if $d_{\mathrm{u}}(u, \mathrm{uAttr}(f)) \neq \bot$ and $d_{\mathrm{r}}(u, \mathrm{rAttr}(f)) \neq \bot$ and formula f holds when the values $d_{\mathrm{u}}(u, \mathrm{uAttr}(f))$ and $d_{\mathrm{r}}(u, \mathrm{rAttr}(f))$ are substituted in it.

A *constraint* is a set (interpreted as a conjunction) of atomic constraints. User u and resource r *satisfy* a constraint c, denoted $\langle u, r\rangle \models c$, if they satisfy every atomic constraint in c. In examples, we write constraints as conjunctions instead of sets. For example, the constraint "specialties $\supseteq$ topics $\wedge$ teams $\ni$ treatingTeam" is satisfied by user u and resource r if the user's specialties include all of the topics associated with the resource, and the set of teams associated with the user contains the treatingTeam associated with the resource.

A *user-permission tuple* is a tuple $\langle u, r, o\rangle$ containing a user, a resource, and an operation. This tuple means that user u has permission to perform operation o on resource r. A *user-permission relation* is a set of such tuples.

A *rule* is a tuple $\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$, where e_{u} is a user-attribute expression, e_{r} is a resource-attribute expression, O is a set of operations, and c is a constraint. For a rule $\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$, let $\mathrm{uae}(\rho) = e_{\mathrm{u}}$, $\mathrm{rae}(\rho) = e_{\mathrm{r}}$, $\mathrm{ops}(\rho) = O$, and $\mathrm{con}(\rho) = c$. For example, the rule $\langle$true, type=task $\wedge$ proprietary=false, {read, request}, projects $\ni$ project $\wedge$ expertise $\supseteq$ expertise$\rangle$ used in our project management case study can be interpreted as "A user working on a project can read and request to work on a non-proprietary task whose required areas of expertise are among his/her areas of expertise." User u, resource r, and operation o *satisfy* a rule ρ, denoted $\langle u, r, o\rangle \models \rho$, if $u \models \mathrm{uae}(\rho) \wedge r \models \mathrm{rae}(\rho) \wedge o \in \mathrm{ops}(\rho) \wedge \langle u, r\rangle \models \mathrm{con}(\rho)$.

An *ABAC policy* is a tuple $\langle U, R, \mathit{Op}, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, \mathit{Rules}\rangle$, where U, R, A_{u}, A_{r}, d_{u}, and d_{r} are as described above, Op is a set of operations, and Rules is a set of rules.

The user-permission relation induced by a rule ρ is $[\![\rho]\!] = \{\langle u, r, o\rangle \in U \times R \times \mathit{Op} \mid \langle u, r, o\rangle \models \rho\}$. Note that U, R, d_{u}, and d_{r} are implicit arguments to $[\![\rho]\!]$.

The user-permission relation induced by a policy π with the above form is $[\![\pi]\!] = \bigcup_{\rho \in \mathit{Rules}} [\![\rho]\!]$.

4.2 The ABAC Policy Mining Problem

An *access control list (ACL) policy* is a tuple $\langle U, R, \mathit{Op}, \mathit{UP}_0\rangle$, where U is a set of users, R is a set of resources, Op is a set of operations, and $\mathit{UP}_0 \subseteq U \times R \times \mathit{Op}$ is a user-permission relation, obtained from the union of the access control lists.

An ABAC policy π is *consistent* with an ACL policy $\langle U, P, \mathit{Op}, \mathit{UP}_0\rangle$ if they have the same sets of users, resource, and operations and $[\![\pi]\!] = \mathit{UP}_0$.

An ABAC policy consistent with a given ACL policy can be trivially constructed, by creating a separate rule corresponding to each user-permission tuple in the ACL policy, simply using uid and rid to identify the relevant user and resource. Of course, such an ABAC policy is as verbose and hard to manage as the original ACL policy. This observation forces us to ask: among ABAC policies semantically consistent with a given ACL policy π_0, which ones are preferable? We adopt two criteria.

One criterion is that policies that do not use the attributes uid and rid are preferable, because policies that use uid and rid are partly identity-based, not entirely attribute-based. Therefore, our definition of ABAC policy mining requires that these attributes are used only if necessary, i.e., only if every ABAC policy semantically consistent with π_0 contains rules that use them.

The other criterion is to maximize a policy quality metric. A *policy quality metric* is a function Q_{pol} from ABAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this is natural for metrics based on policy size. For generality, we parameterize the policy mining problem by the policy quality metric.

The *ABAC policy mining problem* is: given an ACL policy $\pi_0 = \langle U, R, Op, UP_0\rangle$, user attributes A_{u}, resource attributes A_{r}, user attribute data d_{u}, resource attribute data d_{r}, and a policy quality metric Q_{pol}, find a set *Rules* of rules such that the ABAC policy $\pi = \langle U, R, Op, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, Rules\rangle$ that (1) is consistent with π_0, (2) uses uid only when necessary, (3) uses rid only when necessary, and (4) has the best quality, according to Q_{pol}, among such policies.

The policy quality metric that our algorithm aims to optimize is *weighted structural complexity* (WSC) [MCL+10], a generalization of policy size. This is consistent with usability studies of access control rules, which conclude that more concise policies are more manageable [BM13]. Informally, the WSC of an ABAC policy is a weighted sum of the number of elements in the policy. Formally, the WSC of an ABAC policy π with rules *Rules* is $\mathrm{WSC}(\pi) = \mathrm{WSC}(Rules)$, defined by

$$\begin{aligned}
\mathrm{WSC}(e) &= \sum_{a\in \mathrm{attr}_1(e)} |e(a)| + \sum_{a\in \mathrm{attr}_{\mathrm{m}}(e), s\in e(a)} |s| \\
\mathrm{WSC}(\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle) &= w_1\mathrm{WSC}(e_{\mathrm{u}}) + w_2\mathrm{WSC}(e_{\mathrm{r}}) \\
&\quad + w_3|O| + w_4|c| \\
\mathrm{WSC}(Rules) &= \sum_{\rho\in Rules} \mathrm{WSC}(\rho),
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights.

Computational Complexity We show that the ABAC policy mining problem is NP-hard, by reducing the Edge Role Mining Problem (Edge RMP) [LVA08] to it. NP-hardness of Edge RMP follows from Theorem 1 in [MCL+10]. The basic idea of the reduction is that an Edge RMP instance I_R is translated into an ABAC policy mining problem instance I_A with uid and rid as the only attributes. Given a solution π_{ABAC} to problem instance I_A, the solution to I_R is constructed by interpreting each rule as a role. Details of the reduction appear in Section A.1 in the Supplemental Material.

It is easy to show that a decision-problem version of ABAC policy mining is in NP. The decision-problem version asks whether there exists an ABAC policy that meets conditions (1)–(3) in the above definition of the ABAC policy mining problem and has WSC less than or equal to a given value.

4.3 Policy Mining Algorithm

Top-level pseudocode for our policy mining algorithm appears in Figure 4.1. Functions called by the top-level pseudocode are described next. Function names hyperlink to pseudocode for the function, if it is included in the paper, otherwise to a description of the function. An example illustrating the processing of a user-permission tuple by our algorithm appears in Section A.6 in the Supplemental Material. For efficiency, our algorithm incorporates heuristics and is not guaranteed to generate a policy with minimal WSC.

The function addCandidateRule($s_{\mathrm{u}}, s_r, s_{\mathrm{o}}, cc, \mathrm{uncovUP}, \mathit{Rules}$) in Figure 4.2 first calls computeUAE to compute a user-attribute expression e_{u} that characterizes s_{u}, then calls computeRAE to compute a resource-attribute expression e_{r} that characterizes s_r. It then calls generalizeRule($\rho, cc, \mathrm{uncovUP}, \mathit{Rules}$) to generalize the rule $\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, s_{\mathrm{o}}, \emptyset \rangle$ to ρ' and adds ρ' to candidate rule set *Rules*. The details of the functions called by addCandidateRule are described next.

The function computeUAE(s, U) computes a user-attribute expression e_{u} that characterizes the set s of users. The conjunct for each attribute a contains the values of a for users in s, unless one of those values is $\perp$, in which case a is unused (i.e., the conjunct for a is $\top$). Furthermore, the conjunct for uid is removed if the resulting attribute expression still characterizes s; this step is useful because policies that are not identity-based generalize better. Similarly, computeRAE(s, R) computes a resource-attribute expression that characterizes the set s of resources. The attribute expressions returned by computeUAE and computeRAE might not be minimum-sized among expressions that characterize s: it is possible that some conjuncts can be removed. We defer minimization of the attribute expressions until after the call to generalizeRule (described below), because minimizing them before that would reduce opportunities to find relations between values of user attributes and resource attributes in generalizeRule.

The function candidateConstraint(r, u) returns a set containing all the atomic constraints that hold between resource r and user u. Pseudocode for candidateConstraint is straightforward and omitted.

A rule ρ' is *valid* if $[\![\rho']\!] \subseteq UP_0$.

The function generalizeRule($\rho, cc, \mathrm{uncovUP}, \mathit{Rules}$) in Figure 4.3 attempts to generalize rule ρ by adding some of the atomic constraints f in cc to ρ and eliminating the conjuncts of the user attribute expression and the resource attribute expression corresponding to the attributes used in f, i.e., mapping those attributes to $\top$. If the resulting rule is invalid, the function attempts a more conservative generalization by eliminating only one of those conjuncts, keeping the other. We call a rule obtained in this way a *generalization* of ρ. Such a rule is more general than ρ in the sense that it refers to relationships instead of specific values. Also, the user-permission relation induced by a generalization of ρ is a superset of the user-permission relation induced by ρ.

If there are no valid generalizations of ρ, then generalizeRule($\rho, cc, \mathrm{uncovUP}, \mathit{Rules}$) simply returns ρ. If there is a valid generalization of ρ, generalizeRule($\rho, cc, \mathrm{uncovUP}, \mathit{Rules}$) returns the generalization ρ' of ρ with the best quality according to a given rule quality metric. Note that ρ' may cover tuples that are already covered (i.e., are in UP); in other words, our algorithm can generate policies containing rules whose meanings overlap. A *rule quality metric* is a function $Q_{\mathrm{rul}}(\rho, UP)$ that maps a rule ρ to a totally-ordered set, with the

ordering chosen so that larger values indicate high quality. The second argument UP is a set of user-permission tuples. Based on our primary goal of minimizing the generated policy's WSC, and a secondary preference for rules with more constraints, we define

$$Q_{\text{rul}}(\rho, UP) = \langle |[\![\rho]\!] \cap UP| / \text{WSC}(\rho),\ |\text{con}(\rho)| \rangle.$$

The secondary preference for more constraints is a heuristic, based on the observation that rules with more constraints tend to be more general than other rules with the same $|[\![\rho]\!] \cap UP|/\text{WSC}(\rho)$ (such rules typically have more conjuncts) and hence lead to lower WSC. In generalizeRule, uncovUP is the second argument to Q_{rul}, so $[\![\rho]\!] \cap UP$ is the set of user-permission tuples in UP_0 that are covered by ρ and not covered by rules already in the policy. The loop over i near the end of the pseudocode for generalizeRule considers all possibilities for the first atomic constraint in cc that gets added to the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in c that get added to the constraint.

The function mergeRules(*Rules*) in Figure 4.4 attempts to reduce the WSC of *Rules* by removing redundant rules and merging pairs of rules. A rule ρ in *Rules* is *redundant* if *Rules* contains another rule ρ' such that $[\![\rho]\!] \subseteq [\![\rho']\!]$. Informally, rules ρ_1 and ρ_2 are merged by taking, for each attribute, the union of the conjuncts in ρ_1 and ρ_2 for that attribute. If the resulting rule ρ_{merge} is valid, ρ_{merge} is added to *Rules*, and ρ_1 and ρ_2 and any other rules that are now redundant are removed from *Rules*. mergeRules(*Rules*) updates its argument *Rules* in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(*Rules*) attempts to simplify all of the rules in *Rules*. It updates its argument *Rules* in place, replacing rules in *Rules* with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in the following ways. (1) It eliminates sets that are supersets of other sets in conjuncts for multi-valued user attributes. The $\supseteq$-based semantics for such conjuncts implies that this does not change the meaning of the conjunct. For example, a conjunct $\{\{a\}, \{a, b\}\}$ is simplified to $\{\{a\}\}$. (2) It eliminates elements from sets in conjuncts for multi-valued user attributes when this preserves validity of the rule; note that this might increase but cannot decrease the meaning of a rule. For example, if every user whose specialties include a also have specialty b, and a rule contains the conjunct $\{\{a, b\}\}$ for the specialties attribute, then b will be eliminated from that conjunct. (3) It eliminates conjuncts from a rule when this preserves validity of the rule. Since removing one conjunct might prevent removal of another conjunct, it searches for the set of conjuncts to remove that maximizes the quality of the resulting rule, while preserving validity. The user can specify a set of *unremovable attributes*, i.e., attributes for which simplifyRules should not try to eliminate the conjunct, because eliminating it would increase the risk of generating an overly general policy, i.e., a policy that might grant inappropriate permissions when new users or new resources (hence new permissions) are added to the system. Our experience suggests that appropriate unremovable attributes can be identified based on the obvious importance of some attributes and by examination of the policy generated without specification of unremovable attributes. (4) It eliminates atomic constraints from a rule when this preserves validity of the rule. It searches for the set of atomic constraints to remove that maximizes the quality of the resulting rule, while preserving validity. (5) It

// Rules is the set of candidate rules
1: $\mathit{Rules} = \emptyset$
// uncovUP contains user-permission tuples in UP_0
// that are not covered by Rules
2: uncovUP = UP_0.copy()
3: **while** $\neg$uncovUP.empty()
// Select an uncovered user-permission tuple.
4: $\langle u, r, o \rangle$ = some tuple in uncovUP
5: cc = candidateConstraint(r, u)
// s_u contains users with permission $\langle r, o \rangle$ and
// that have the same candidate constraint for r as u
6: $s_\mathrm{u} = \{u' \in U \mid \langle u', r, o \rangle \in UP_0$
7: $\wedge$ candidateConstraint$(r, u') = cc\}$
8: addCandidateRule($s_\mathrm{u}, \{r\}, \{o\}, cc$, uncovUP, Rules)
// s_o is set of operations that u can apply to r
9: $s_\mathrm{o} = \{o' \in \mathit{Op} \mid \langle u, r, o' \rangle \in UP_0\}$
10: addCandidateRule($\{u\}, \{r\}, s_\mathrm{o}, cc$, uncovUP, Rules)
11: **end while**
// Repeatedly merge and simplify rules, until
// this has no effect
12: mergeRules(Rules)
13: **while** simplifyRules(Rules) && mergeRules(Rules)
14: skip
15: **end while**
// Select high quality rules into final result Rules'.
16: $\mathit{Rules}' = \emptyset$
17: Repeatedly select the highest quality rules from
Rules to Rules' until $\sum_{\rho \in \mathit{Rules}'} [\![\rho]\!] = UP_0$,
using $UP_0 \setminus [\![\mathit{Rules}']\!]$ as second argument to Q_rul
18: **return** Rules'

Figure 4.1: Policy mining algorithm.

function addCandidateRule($s_\mathrm{u}, s_r, s_\mathrm{o}, cc$, uncovUP, Rules)
// Construct a rule ρ that covers user-permission
// tuples $\{\langle u, r, o \rangle \mid u \in s_\mathrm{u} \wedge r \in s_r \wedge o \in s_\mathrm{o}\}$.
1: e_u = computeUAE(s_u, U)
2: e_r = computeRAE(s_r, R)
3: $\rho = \langle e_\mathrm{u}, e_\mathrm{r}, s_\mathrm{o}, \emptyset \rangle$
4: ρ' = generalizeRule(ρ, cc, uncovUP, Rules)
5: Rules.add(ρ')
6: uncovUP.removeAll($[\![\rho']\!]$)

Figure 4.2: Compute a candidate rule ρ' and add ρ' to candidate rule set Rules

```
    function generalizeRule(ρ, cc, uncovUP,
                Rules)
    // ρ_best is highest-quality generalization
    // of ρ
1:  ρ_best = ρ
    // cc′ contains formulas from cc that lead
    // to valid generalizations of ρ.
2:  cc′ = new Vector()
3:  // gen[i] is a generalization of ρ
    // using cc′[i]
4:  gen = new Vector()
    // find formulas in cc that lead to valid
    // generalizations of ρ.
5:  for f in cc
      // try to generalize ρ by adding f
      // and eliminating conjuncts for
      // both attributes used in f.
6:    ρ′ = ⟨uae(ρ)[uAttr(f) ↦ ⊤],
           rae(ρ)[rAttr(f) ↦ ⊤],
           ops(ρ), con(ρ) ∪ {f}⟩
      // check if ⟦ρ′⟧ is a valid rule
7:    if ⟦ρ′⟧ ⊆ UP_0
8:      cc′.add(f)
9:      gen.add(ρ′)
10:   else
        // try to generalize ρ by adding f
        // and eliminating conjunct for
        //  one user attribute used in f
11:     ρ′ = ⟨uae(ρ)[uAttr(f) ↦ ⊤],
             rae(ρ), ops(ρ), con(ρ) ∪ {f}⟩
12:     if ⟦ρ′⟧ ⊆ UP_0
13:       cc′.add(f)
14:       gen.add(ρ′)
15:     else
    // try to generalize ρ by adding f and
    // eliminating conjunct for one resource
    // attribute used in f.
16:       ρ′ = ⟨uae(ρ), rae(ρ)[rAttr(f) ↦ ⊤],
               ops(ρ), con(ρ) ∪ {f}⟩
18:       if ⟦ρ′⟧ ⊆ UP_0
19:         cc′.add(f)
20:         gen.add(ρ′)
21:       end if
22:     end if
23:   end if
24: end for
25: for i = 1 to cc′.length
26:   // try to further generalize gen[i]
27:   ρ′′ = generalizeRule(gen[i], cc′[i+1 ..],
28:                         uncovUP, Rules)
29:   if Q_rul(ρ′′, uncovUP) > Q_rul(ρ_best, uncovUP)
30:     ρ_best = ρ′′
31:   end if
32: end for
33: return ρ_best
```

Figure 4.3: Generalize rule ρ by adding some formulas from cc to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i.

eliminates overlapping values between rules. Specifically, a value v in the conjunct for a user attribute a in a rule ρ is removed if there is another rule ρ' in the policy such that (*i*) $\mathrm{attr}(\mathrm{uae}(\rho')) \subseteq \mathrm{attr}(\mathrm{uae}(\rho))$ and $\mathrm{attr}(\mathrm{rae}(\rho')) \subseteq \mathrm{attr}(\mathrm{rae}(\rho))$, (*ii*) the conjunct of $\mathrm{uae}(\rho')$ for a contains v, (*iii*) each conjunct of $\mathrm{uae}(\rho')$ or $\mathrm{rae}(\rho')$ other than the conjunct for a is either $\top$ or a superset of the corresponding conjunct of ρ, and (*iv*) $\mathrm{con}(\rho') \subseteq \mathrm{con}(\rho)$. The condition for removal of a value in the conjunct for a resource attribute is analogous. If a conjunct of $\mathrm{uae}(\rho)$ or $\mathrm{rae}(\rho)$ becomes empty, ρ is removed from the policy. For example, if a policy contains the rules $\langle \mathrm{dept} \in \{d_1, d_2\} \wedge \mathrm{position} = p_1, \mathrm{type} = t_1, \mathrm{read}, \mathrm{dept} = \mathrm{dept}\rangle$ and $\langle \mathrm{dept} \in \{d_1\} \wedge \mathrm{position} = p_1, \mathrm{type} \in \{t_1, t_2\}, \mathrm{read}, \mathrm{dept} = \mathrm{dept}\rangle$, then d_1 is eliminated from the former rule. (6) It eliminates overlapping operations between rules. The details are similar to those for elimination of overlapping values between rules. For example, if a policy

```
    function mergeRules(Rules)
1:  // Remove redundant rules
2:  rdtRules = {ρ ∈ Rules | ∃ ρ' ∈ Rules \ {ρ}. [[ρ]] ⊆ [[ρ']]}
3:  Rules.removeAll(rdtRules)
4:  // Merge rules
5:  workSet = {(ρ_1, ρ_2) | ρ_1 ∈ Rules ∧ ρ_2 ∈ Rules
                             ∧ ρ_1 ≠ ρ_2 ∧ con(ρ_1) = con(ρ_2)}
6:  while not(workSet.empty())
      // Remove an arbitrary element of the workset
7:    (ρ_1, ρ_2) = workSet.remove()
8:    ρ_merge = ⟨uae(ρ_1) ∪ uae(ρ_2), rae(ρ_1) ∪ rae(ρ_2),
                 ops(ρ_1) ∪ ops(ρ_2), con(ρ_1)⟩
9:    if [[ρ_merge]] ⊆ UP_0
        // The merged rule is valid. Add it to Rules,
        // and remove rules that became redundant.
10:     rdtRules = {ρ ∈ Rules | [[ρ]] ⊆ [[ρ_merge]]}
11:     Rules.removeAll(rdtRules)
12:     workSet.removeAll({(ρ_1, ρ_2) ∈ workSet |
                     ρ_1 ∈ rdtRules ∨ ρ_2 ∈ rdtRules})
13:     workSet.addAll({(ρ_merge, ρ) | ρ ∈ Rules
                                  ∧ con(ρ) = con(ρ_merge)})
14:     Rules.add(ρ_merge)
15:   end if
16: end while
17: return true if any rules were merged
```

Figure 4.4: Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$.

contains the rules $\langle \text{dept} = d_1, \text{type} = t_1, \text{read}, \text{dept} = \text{dept} \rangle$ and $\langle \text{dept} = d_1 \wedge \text{position} = p_1, \text{type} = t_1, \{\text{read}, \text{write}\}, \text{dept} = \text{dept} \rangle$, then read is eliminated from the latter rule.

Asymptotic Running Time The algorithm's overall running time is worst-case cubic in $|UP_0|$. A detailed analysis of the asymptotic running time appears in Section A.2 in the Supplemental Material. In the experiments with case studies and synthetic policies described in Section 4.4, the observed running time is roughly quadratic and roughly linear, respectively, in $|UP_0|$.

Attribute Selection Attribute data may contain attributes irrelevant to access control. This potentially hurts the effectiveness and performance of policy mining algorithms

[FSBB09, NLC+09]. Therefore, before applying our algorithm to a dataset that might contain irrelevant attributes, it is advisable to use the method in [FSBB09] or [MLQ+10] to determine the relevance of each attribute to the user-permission assignment and then eliminate attributes with low relevance.

Processing Order The order in which tuples and rules are processed can affect the mined policy. The order in which our algorithm processes tuples and rules is described in Section A.3 in the Supplemental Material.

Optimizations Our implementation incorporates a few optimizations not reflected in the pseudocode. Details of these optimizations appear in Section A.4 in the Supplemental Material. Briefly, the most important optimizations are calling mergeRules periodically (not only after all of UP_0 has been covered) and caching the meanings of rules and related values.

4.3.1 Noise Detection

In practice, the given user-permission relation often contains noise, consisting of over-assignments and under-assignments. An *over-assignment* is when a permission is inappropriately granted to a user. An *under-assignment* is when a user lacks a permission that he or she should be granted. Noise incurs security risks and significant IT support effort [MLQ+10]. This section describes extensions of our algorithm to handle noise. The extended algorithm detects and reports suspected noise and generates an ABAC policy that is consistent with its notion of the correct user-permission relation (i.e., with the suspected noise removed). The user should examine the suspected noise and decide which parts of it are actual noise (i.e., errors in the user-permission relation). If all of it is actual noise, then the policy already generated is the desired one; otherwise, the user should remove the parts that are actual noise from the user-permission relation to obtain a correct user-permission relation and then run the algorithm without the noise detection extension on it to generate the desired ABAC policy.

Over-assignments are often the result of incomplete revocation of old permissions when users change job functions [MLQ+10]. Therefore, over-assignments usually cannot be captured concisely using rules with attribute expressions that refer to the current attribute information, so a candidate rule constructed from a user-permission tuple that is an over-assignment is less likely to be generalized and merged with other rules, and that candidate rule will end up as a low-quality rule in the generated policy. So, to detect over-assignments, we introduce a rule quality threshold τ. The rule quality metric used here is the first component of the metric used in the loop in Figure 4.1 that constructs $Rules'$; thus, τ is a threshold on the value of $Q_{\mathrm{rul}}(\rho, \mathrm{uncovUP})$, and the rules with quality less than or equal to τ form a suffix of the sequence of rules added to $Rules'$. The extended algorithm reports as suspected over-assignments the user-permission tuples covered in $Rules'$ only by rules with quality less than or equal to τ, and then it removes rules with quality less than or equal to τ from $Rules'$. Adjustment of τ is guided by the user. For example, the user might guess a percentage of over-assignments (e.g., 3%) based on experience, and let the system adjust τ until the number of reported over-assignments is that percentage of $|UP_0|$. Note that re-computing over-assignments after a change to τ does not require re-generating the policy.

To detect under-assignments, we look for rules that are almost valid, i.e., rules that would be valid if a relatively small number of tuples were added to UP_0. A parameter α quantifies the notion of "relatively small". A rule is α *almost valid* if the fraction of invalid user-permission tuples in $[\![\rho]\!]$ is at most α, i.e., $|[\![\rho]\!] \setminus UP_0| \div |[\![\rho]\!]| \leq \alpha$. In places where the policy mining algorithm checks whether a rule is valid, if the rule is α almost valid, the algorithm treats it as if it were valid. The extended algorithm reports $\bigcup_{\rho \in Rules'} [\![\rho]\!] \setminus UP_0$ as the set of suspected under-assignments, and (as usual) it returns $Rules'$ as the generated policy. Adjustment of α is guided by the user, similarly as for the over-assignment threshold τ.

4.4 Evaluation

The general methodology used for evaluation is described in Section 1.2. We applied this methodology to sample policies and synthetic policies. Evaluation on policies (including attribute data) from real organizations would be ideal, but we are not aware of any suitable and publicly available policies from real organizations. Therefore, we developed sample policies that, although not based directly on specific real-world case studies, are intended to be similar to policies that might be found in the application domains for which they are named. The sample policies are relatively small and intended to resemble interesting core parts of full-scale policies in those application domains. Despite their modest size, they are a significant test of the effectiveness of our algorithm, because they express non-trivial policies and exercise all features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The synthetic policies are used primarily to assess the behavior of the algorithm as a function of parameters controlling specific structural characteristics of the policies.

We implemented our policy mining algorithm in Java and ran experiments on a laptop with a 2.5 GHz Intel Core i5 CPU. All of the code and data is available at `http://www.cs.sunysb.edu/~stoller/`. In our experiments, the weights w_i in the definition of WSC equal 1.

4.4.1 Evaluation on Sample Policies

We developed four sample policies, each consisting of rules and a manually written attribute dataset containing a small number of instances of each type of user and resource. We also generated synthetic attribute datasets for each sample policy. The sample policies are described very briefly in this section. Details of the sample policies, including all policy rules, some illustrative manually written attribute data, and a more detailed description of the synthetic attribute data generation algorithm appear in Section A.5 in the Supplemental Material.

Figure 4.5 provides information about their size. Although the sample policies are relatively small when measured by a coarse metric such as number of rules, they are complex, because each rule has a lot of structure. For example, the number of well-formed rules built using the attributes and constants in each policy and that satisfy the strictest syntactic size limits satisfied by rules in the sample policies (at most one conjunct in each UAE, at most

Policy	$\lvert Rules\rvert$	$\lvert A_{\mathrm{u}}\rvert$	$\lvert A_{\mathrm{r}}\rvert$	$\lvert Op\rvert$	Type	N	$\lvert U\rvert$	$\lvert R\rvert$	$\lvert \mathrm{Val_s}\rvert$	$\lvert UP\rvert$	$\widehat{[\![\rho]\!]}$
university	10	6	5	9	man	2	22	34	76	168	19
					syn	10	479	997	1651	8374	837
					syn	20	920	1918	3166	24077	2408
health care	9	6	7	3	man	2	21	16	55	51	6.7
					syn	10	200	720	1386	1532	195
					syn	20	400	1440	2758	3098	393
project mgmt	11	8	6	7	man	2	19	40	77	189	19
					syn	10	100	200	543	960	96
					syn	20	200	400	1064	1920	193

Figure 4.5: Sizes of the sample policies. "Type" indicates whether the attribute data in the policy is manually written ("man") or synthetic ("syn"). N is the number of departments for the university and project management sample policies, and the number of wards for the health care sample policy. $\widehat{[\![\rho]\!]}$ is the average number of user-permission tuples that satisfy each rule. An empty cell indicates the same value as the cell above it.

two conjuncts in each RAE, at most two atomic constraints in each constraint, at most one atomic value in each UAE conjunct, at most two atomic values in each RAE conjunct, etc.) is more than 10^{12} for the sample policies with manually written attribute data and is much higher for the sample policies with synthetic attribute data and the synthetic policies.

In summary, our algorithm is very effective for all three sample policies: there are only small differences between the original and mined policies if no attributes are declared unremovable, and the original and mined policies are identical if the resource-type attribute is declared unremovable.

University Sample Policy Our university sample policy controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=transcript" is replaced with the constraint "department=department" in one rule. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Health Care Sample Policy Our health care sample policy controls access by nurses, doctors, patients, and agents (e.g., a patient's spouse) to electronic health records (HRs) and HR items (i.e., entries in health records). If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=HRitem" is eliminated from four rules; that conjunct is unnecessary, because those rules also contain a conjunct for the "topic" attribute, and the "topic" attribute is used only for resources with type=HRitem. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Project Management Sample Policy Our project management sample policy controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. If no attributes are declared unremovable, the generated policy is the same as the original ABAC policy except that the RAE conjunct "type=task" is eliminated from three rules; the explanation is similar to the above explanation for the health care sample policy. If resource type is declared unremovable, the generated policy is identical to the original ABAC policy.

Running Time on Synthetic Attribute Data We generated a series of pseudorandom synthetic attribute datasets for the sample policies, parameterized by a number N, which is the number of departments for the university and project management sample policies, and the number of wards for the health care sample policy. The generated attribute data for users and resources associated with each department or ward are similar to but more numerous than the attribute data in the manually written datasets. Figure 4.5 contains information about the sizes of the policies with synthetic attribute data, for selected values of N. Policies for the largest shown value of N are generated as described in Section A.5 in the Supplemental Material; policies for smaller values of N are prefixes of them. Each row contains the average over 20 synthetic policies with the specified N. For all sizes of synthetic attribute data, the mined policies are the same as with the manually generated attribute data. This reflects that larger attribute datasets are not necessarily harder to mine from, if they represent more instances of the same rules; the complexity is primarily in the structure of the rules. Figure 4.6 shows the algorithm's running time as a function of N. Each data point is an average of the running times on 20 policies with synthetic attribute data. Error bars (too small to see in most cases) show 95% confidence intervals using Student's t-distribution. The running time is a roughly quadratic function of N for all three sample policies, with different constant factors. Different constant factors are expected, because policies are very complex structures, and N captures only one aspect of the size and difficulty of the policy mining problem instance. For example, the constant factors are larger for the university sample policy mainly because it has larger $|UP|$, as a function of N, than the other sample policies. For example, Figure 4.5 shows that $|UP|$ for the university sample policy with $N = 10$ is larger than $|UP|$ for the other sample policies with $N = 20$.

Benefit of Periodic Rule Merging Optimization It is not obvious *a priori* whether the savings from periodic merging of rules outweighs the cost. In fact, the net benefit grows with policy size. For example, for the university policy with synthetic attribute data, this optimization provides a speedup of (67 sec)/(40 sec) = 1.7 for $N_{\mathrm{dept}} = 10$ and a speedup of (1012 sec)/(102 sec) = 9.9 for $N_{\mathrm{dept}} = 15$.

4.4.2 Evaluation on Synthetic Policies

We also evaluated our algorithm on synthetic ABAC policies. On the positive side, synthetic policies can be generated in all sizes and with varying structural characteristics. On the other hand, even though our synthesis algorithm is designed to generate policies with some realistic characteristics, the effectiveness and performance of our algorithm on synthetic

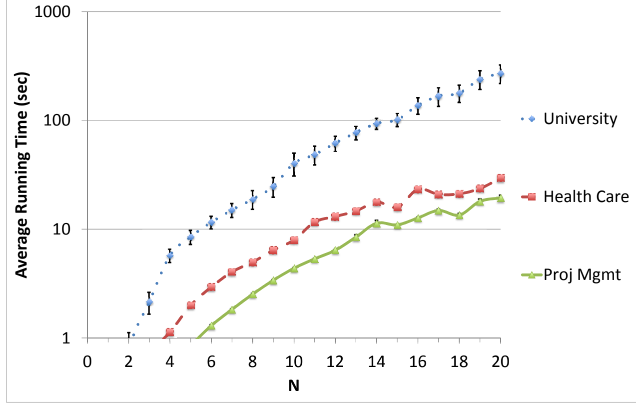

Figure 4.6: Running time (log scale) of the algorithm on synthetic attribute datasets for sample policies. The horizontal axis is N_{dept} for university and project management sample policies and N_{ward} for health care sample policy.

policies might not be representative of its effectiveness and performance on real policies. For experiments with synthetic policies, we compare the syntactic similarity and WSC of the synthetic ABAC policy and the mined ABAC policy. Syntactic similarity of policies measures the syntactic similarity of rules in the policies. It ranges from 0 (completely different) to 1 (identical). The detailed definition of syntactic similarity is in Section A.7 in the Supplemental Material. We do not expect high syntactic similarity between the synthetic and mined ABAC policies, because synthetic policies tend to be unnecessarily complicated, and mined policies tend to be more concise. Thus, we consider the policy mining algorithm to be effective if the mined ABAC policy $\mathit{Rules}_{\text{mined}}$ is simpler (i.e., has lower WSC) than the original synthetic ABAC policy $\mathit{Rules}_{\text{syn}}$. We compare them using the *compression factor*, defined as $\text{WSC}(\mathit{Rules}_{\text{syn}})/\text{WSC}(\mathit{Rules}_{\text{mined}})$. Thus, a compression factor above 1 is good, and larger is better.

Synthetic Policy Generation Our policy synthesis algorithm first generates the rules and then uses the rules to guide generation of the attribute data; this allows control of the number of granted permissions. Our synthesis algorithm takes N_{rule}, the desired number of rules, $N_{\text{cnj}}^{\text{min}}$, the minimum number of conjuncts in each attribute expression, and $N_{\text{cns}}^{\text{min}}$, the minimum number of constraints in each rule, as inputs. The numbers of users and resources are not specified directly but are proportional to the number of rules, since our algorithm generates new users and resources to satisfy each generated rule, as sketched below. Rule generation is based on several statistical distributions, which are either based loosely on our sample policies or assumed to have a simple functional form (e.g., uniform distribution or Zipf distribution). For example, the distribution of the number of conjuncts in each attribute expression is based loosely on our sample policies and ranges from $N_{\text{cnj}}^{\text{min}}$ to $N_{\text{cnj}}^{\text{min}} + 3$, the distribution of the number of atomic constraints in each constraint is based loosely on our sample policies and ranges from $N_{\text{cns}}^{\text{min}}$ to $N_{\text{cns}}^{\text{min}} + 2$, and the distribution of attributes in attribute expressions is assumed to be uniform (i.e., each attribute is equally likely to be selected for use in each conjunct).

The numbers of user attributes and resource attributes are fixed at $N_{\text{attr}} = 8$ (this is

the maximum number of attributes relevant to access control for the datasets presented in [MLC11]). Our synthesis algorithm adopts a simple type system, with 7 types, and with at least one user attribute and one resource attribute of each type. For each type t, the cardinality $c(t)$ is selected from a uniform distribution on the interval $[2, 10N_{\mathrm{rule}} + 2]$, the target ratio between the frequencies of the most and least frequent values of type t is chosen to be 1, 10, or 100 with probability 0.2 0.7, and 0.1, respectively, and a skew $s(t)$ is computed so that the Zipf distribution with cardinality $c(t)$ and skew $s(t)$ has that frequency ratio. When assigning a value to an attribute of type t, the value is selected from the Zipf distribution with cardinality $c(t)$ and skew $s(t)$. Types are also used when generating constraints: constraints relate attributes with the same type.

For each rule ρ, our algorithm ensures that there are at least $N_{\mathrm{urp}} = 16$ user-resource pairs $\langle u, r\rangle$ such that $\langle u, r, o\rangle \models \rho$ for some operation o. The algorithm first checks how many pairs of an existing user and an existing resource (which were generated for previous rules) satisfy ρ or can be made to satisfy ρ by appropriate choice of values for attributes with unknown values (i.e., $\perp$). If the count is less than N_{urp}, the algorithm generates additional users and resources that together satisfy ρ. With the resulting modest number of users and resources, some conjuncts in the UAE and RAE are likely to be unnecessary (i.e., eliminating them does not grant additional permissions to any existing user). In a real policy with sufficiently large numbers of users and resources, all conjuncts are likely be to necessary. To emulate this situation with a modest number of users, for each rule ρ, for each conjunct $e_u(a_u)$ in the UAE e_u in ρ, the algorithm generates a user u' by copying an existing user u that (together with some resource) satisfies ρ and then changing $d_{\mathrm{u}}(u', a_u)$ to some value not in $e_u(a_u)$. Similarly, the algorithm adds resources to increase the chance that conjuncts in resource attribute expressions are necessary, and it adds users and resources to increase the chance that constraints are necessary. The algorithm initially assigns values only to the attributes needed to ensure that a user or resource satisfies the rule under consideration. To make the attribute data more realistic, a final step of the algorithm assigns values to additional attributes until the fraction of attribute values equal to $\perp$ reaches a target fraction $\nu_{\perp} = 0.1$.

Results for Varying Number of Conjuncts To explore the effect of varying the number of conjuncts, we generated synthetic policies with N_{rule} ranging from 10 to 50 in steps of 20, with $N_{\mathrm{cnj}}^{\mathrm{min}}$ ranging from 4 to 0, and with $N_{\mathrm{cns}}^{\mathrm{min}} = 0$. For each value of N_{rule}, synthetic policies with smaller $N_{\mathrm{cnj}}^{\mathrm{min}}$ are obtained by removing conjuncts from synthetic policies with larger $N_{\mathrm{cnj}}^{\mathrm{min}}$. For each combination of parameter values (in these experiments and the experiments with varying number of constraints and varying overlap between rules), we generate 50 synthetic policies and average the results. Some experimental results appear in Figure 4.1. For each value of N_{rule}, as the number of conjuncts decreases, $|UP|$ increases (because the numbers of users and resources satisfying each rule increase), the syntactic similarity increases (because as there are fewer conjuncts in each rule in the synthetic policy, it is more likely that the remaining conjuncts are important and will also appear in the mined policy), and the compression factor decreases (because as the policies get more similar, the compression factor must get closer to 1). For example, for $N_{\mathrm{rule}} = 50$, as $N_{\mathrm{cnj}}^{\mathrm{min}}$ decreases from 4 to 0, average $|UP|$ increases from 1975 to 11969, average syntactic similarity increases from 0.62 to 0.75, and average compression factor decreases from 1.75 to 0.84. The figure also shows

the density of the policies, where the density of a policy is defined as $|UP| \div (|U| \times |P|)$, where the set of granted permissions is $P = \bigcup_{\langle u,r,o\rangle \in UP} \{\langle r, o\rangle\}$. The average densities all fall within the range of densities seen in the 9 real-world datasets shown in [MLL+09, Table 1], namely, 0.003 to 0.19. Density is a decreasing function of N_{rule}, because $|UP|$, $|U|$, and $|P|$ each grow roughly linearly as functions of N_{rule}. The standard deviations of some quantities are relatively large in some cases, but, as the relatively small confidence intervals indicate, this is due to the intrinsic variability of the synthetic policies generated by our algorithm, not due to insufficient samples.

Results for Varying Number of Constraints To explore the effect of varying the number of constraints, we generated synthetic policies with N_{rule} ranging from 10 to 50 in steps of 20, with $N_{\text{cns}}^{\text{min}}$ ranging from 2 to 0, and with $N_{\text{cnj}}^{\text{min}} = 0$. For each value of N_{rule}, policies with smaller $N_{\text{cns}}^{\text{min}}$ are obtained by removing constraints from synthetic policies with larger $N_{\text{cns}}^{\text{min}}$. Some experimental results appear in Figure 4.2. For each value of N_{rule}, as the number of constraints decreases, $|UP|$ increases (because the numbers of users and resources satisfying each rule increase), syntactic similarity decreases (because our algorithm gives preference to constraints over conjuncts, so when $N_{\text{cns}}^{\text{min}}$ is small, the mined policy tends to have more constraints and fewer conjuncts than the synthetic policy), and the compression factor decreases (because the additional constraints in the mined policy cause each rule in the mined policy to cover fewer user-permission tuples on average, increasing the number of rules and hence the WSC). For example, for $N_{\text{rule}} = 50$, as $N_{\text{cns}}^{\text{min}}$ decreases from 2 to 0, average $|UP|$ increases from 3560 to 26472, average syntactic similarity decreases slightly from 0.67 to 0.64, and average compression factor decreases from 1.29 to 0.96.

Results for Varying Overlap Between Rules We also explored the effect of varying overlap between rules, to test our conjecture that policies with more overlap between rules are harder to reconstruct through policy mining. The *overlap* between rules ρ_1 and ρ_2 is $[\![\rho_1]\!] \cap [\![\rho_2]\!]$. To increase the average overlap between pairs of rules in a synthetic policy, we extended the policy generation algorithm so that, after generating each rule ρ, with probability P_{over} the algorithm generates another rule ρ' obtained from ρ by randomly removing one conjunct from $\text{uae}(\rho)$ and adding one conjunct (generated in the usual way) to $\text{rae}(\rho)$; typically, ρ and ρ' have a significant amount of overlap. We also add users and resources that together satisfy ρ', so that $[\![\rho']\!] \not\subseteq [\![\rho]\!]$, otherwise ρ' is redundant. This construction is based on a pattern that occurs a few times in our sample policies. We generated synthetic policies with 30 rules, using the extended algorithm described above. For each value of N_{rule}, we generated synthetic policies with P_{over} ranging from 0 to 1 in steps of 0.25, and with $N_{\text{cnj}}^{\text{min}} = 2$ and $N_{\text{cns}}^{\text{min}} = 0$. Some experimental results appear in Figure 4.3. For each value of N_{rule}, as P_{over} increases, the syntactic similarity decreases (because our algorithm effectively removes overlap, i.e., produces policies with relatively little overlap), and the compression factor increases (because removal of more overlap makes the mined policy more concise). For example, for $N_{\text{rule}} = 50$, as P_{over} increases from 0 to 1, the syntactic similarity decreases slightly from 0.74 to 0.71, and the compression factor increases from 1.16 to 1.23.

4.4.3 Generalization

A potential concern with optimization-based policy mining algorithms is that the mined policies might overfit the given data and hence not be robust, i.e., not generalize well, in the sense that the policy requires modifications to accommodate new users. To evaluate how well policies generated by our algorithm generalize, we applied the following methodology, based on [FSBB09]. The inputs to the methodology are an ABAC policy mining algorithm, an ABAC policy π, and a fraction f (informally, f is the fraction of the data used for training); the output is a fraction e called the *generalization error* of the policy mining algorithm on policy π for fraction f. Given a set U' of users and a policy π, the associated resources for U' are the resources r such that π grants some user in U' some permission on r. To compute the generalization error, repeat the following procedure 10 times and average the results: randomly select a subset U' of the user set U of π with $|U'|/|U| = f$, randomly select a subset R' of the associated resources for U' with $|R'|/|R| = f$, generate an ACL policy π_{ACL} containing only the permissions for users in U' for resources in R', apply the policy mining algorithm to π_{ACL} with the attribute data to generate an ABAC policy π_{gen}, compute the generalization error as the fraction of incorrectly assigned permissions for users not in U' and resources not in R', i.e., as $|S \ominus S'|/|S|$, where $S = \{\langle u, r, o\rangle \in [\![\pi]\!] \mid u \in U \setminus U' \wedge r \in R \setminus R'\}$, $S' = \{\langle u, r, o\rangle \in [\![\pi']\!] \mid u \in U \setminus U' \wedge r \in R \setminus R'\}$, and $\ominus$ is symmetric set difference.

We measured generalization error for f from 0.1 to 0.5 in steps of 0.05 for the university (with $N_{\mathrm{dept}} = 40$), health care (with $N_{\mathrm{ward}} = 40$), and project management (with $N_{\mathrm{dept}} = 40$) sample policies. For the university and health care sample policies, the generalization error is zero in all these cases. For the project management sample policy, the generalization error is 0.11 at $f = 0.1$, drops roughly linearly to zero at $f = 0.35$, and remains zero thereafter. There are no other existing ABAC policy mining algorithms, so a direct comparison of the generalization results from our algorithm with generalization results from algorithms based on other approaches, e.g., probabilistic models, is not currently possible. Nevertheless, these results are promising and suggest that policies generated by our algorithm generalize reasonably well.

4.4.4 Noise

Permission Noise To evaluate the effectiveness of our noise detection techniques in the presence of permission noise, we started with an ABAC policy, generated an ACL policy, added noise, and applied our policy mining algorithm to the resulting policy. To add a specified level ν of permission noise, measured as a percentage of $|UP_0|$, we added $\nu|UP_0|/6$ under-assignments and $5\nu|UP_0|/6$ over-assignments to the ACL policy generated from the ABAC policy. This ratio is based on the ratio of Type I and Type II errors in [MLQ+10, Table 1]. The over-assignments are user-permission tuples generated by selecting the user, resource, and operation from categorical distributions with approximately normally distributed probabilities ("approximately" because the normal distribution is truncated on the sides to have the appropriate finite domain); we adopted this approach from [MLQ+10]. The under-assignments are removals of user-permission tuples generated in the same way. For each noise level, we ran our policy mining algorithm with noise detection inside a loop that searched for the best values of α (considering values between 0.01 and 0.09 in steps of .01) and τ

(considering 0.08, values between 0.1 and 0.9 in steps of 0.1, and between 1 and 10 in steps of 1), because we expect τ to depend on the noise level, and we want to simulate an experienced administrator, so that the results reflect the capabilities and limitations of the noise detection technique rather than the administrator. The best values of α and τ are the ones that maximize the Jaccard similarity of the actual (injected) noise and the reported noise. ROC curves that illustrate the trade-off between false positives and false negatives when tuning the values of α and τ appear in Section A.8 in the Supplemental Material.

We started with the university (with $N_{\text{dept}} = 4$), health care (with $N_{\text{ward}} = 6$), and project management (with $N_{\text{dept}} = 6$) sample policies with synthetic attribute data (we also did some experiments with larger policy instances and got similar results), and with synthetic policies with $N_{\text{rule}} = 20$. Figure 4.7 shows the Jaccard similarity of the actual and reported over-assignments and the Jaccard similarity of the actual and reported under-assignments. Note that, for a policy mining algorithm without noise detection (hence the reported noise is the empty set), these Jaccard similarities would be 0. Each data point is an average over 10 policies, and error bars (too small to see in some cases, and omitted when the standard deviation is 0) show 95% confidence intervals using Student's t-distribution. Over-assignment detection is accurate, with average Jaccard similarity always 0.94 or higher (in our experiments). Under-assignment detection is very good for university and project management, with average Jaccard similarity always 0.93 or higher, but less accurate for health care and synthetic policies, with average Jaccard similarity always 0.63 or higher. Intuitively, detecting over-assignments is somewhat easier, because it is unlikely that there are high-quality rules that cover the over-assignments, so we mostly get rules that do not over-assign and hence the over-assignments get classified correctly. However, under-assignments are more likely to affect the generated rules, leading to mis-classification of under-assignments. As a function of noise level in the considered range, the Jaccard similarities are flat in some cases and generally trend slightly downward in other cases. Figure 4.8 shows the semantic similarity of the original and mined policies. Note that, for a policy mining algorithm without noise detection, the semantic similarity would equal $1 - \nu$. With our algorithm, the semantic similarity is always significantly better than this. The average semantic similarity is always 0.98 or higher, even for $\nu = 0.12$. The similarities are generally lower for synthetic policies than sample policies, as expected, because synthetic policies are not reconstructed as well even in the absence of noise.

Permission Noise and Attribute Noise To evaluate the effectiveness of our noise detection techniques in the presence of permission noise and attribute noise, we performed experiments in which, for a given noise level ν, we added $\nu|UP_0|/7$ under-assignments, $5\nu|UP_0|/7$ over-assignments, and $\nu|UP_0|/7$ permission errors due to attribute errors to the ACL policy generated from the ABAC policy (in other words, we add attribute errors until $\nu|UP_0|/7$ user-permission tuples have been added or removed due to attribute errors; this way, attribute errors are measured on the same scale as under-assignments and over-assignments). The attribute errors are divided equally between missing values (i.e., replace a non-bottom value with bottom) and incorrect values (i.e., replace a non-bottom value with another non-bottom value). Our current techniques do not attempt to distinguish permission

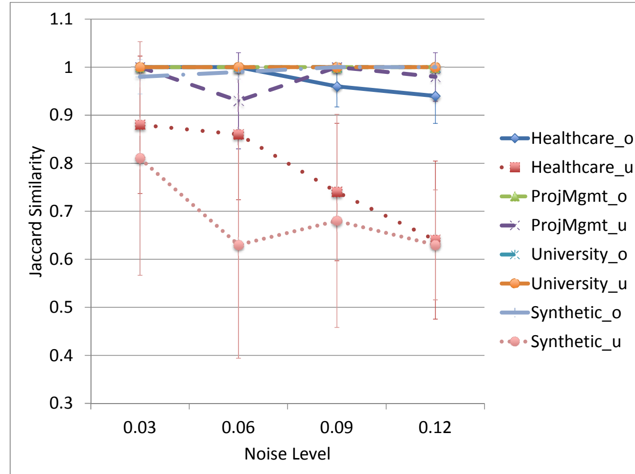

Figure 4.7: Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level. Curve names ending with _o and _u are for over-assignments and under-assignments, respectively. The curves for University_u and Synthetic_o are nearly the same and overlap each other.

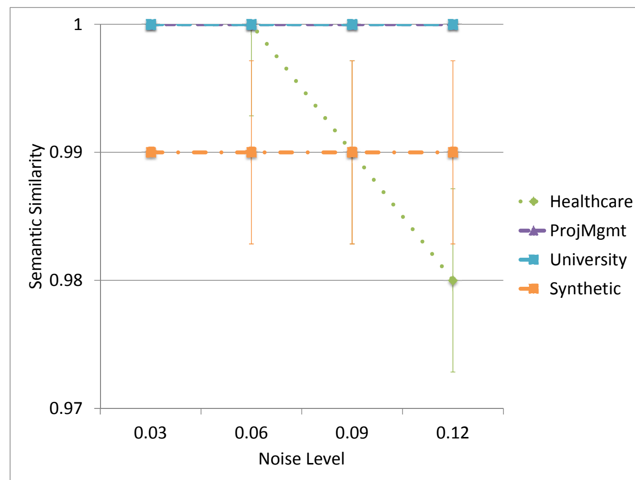

Figure 4.8: Semantic similarity of the original policy and the mined policy, as a function of permission noise level.

noise from attribute noise (this is a topic for future research); policy analysts are responsible for determining whether a reported suspected error is due to an incorrect permission, an incorrect or missing attribute value, or a false alarm. Since our techniques report only suspected under-assignments and suspected over-assignments, when comparing actual noise to reported noise, permission changes due to attribute noise (i.e., changes in the set of user-permission tuples that satisfy the original policy rules) are included in the actual noise. We started with the same policies as above. Graphs of Jaccard similarity of actual and reported noise, and syntactic similarity of original and mined policies, appear in Section A.9 in the Supplemental Material. The results are similar to those without attribute noise, except with slightly lower similarities for the same fraction of permission errors. This shows that our

approach to noise detection remains appropriate in the presence of combined attribute noise and permission noise.

4.4.5 Comparison with Inductive Logic Programming

We implemented a translation from ABAC policy mining to Inductive Logic Programming (ILP) and applied Progol [MB00, MF01], a well-known ILP system developed by Stephen Muggleton, to translations of our sample policies and synthetic policies. Details of the translation appear in Section A.10 in the Supplemental Material. Progol mostly succeeds in reconstructing the policies for university and project management, except it fails to learn rules with conjuncts or operation sets containing multiple constants, instead producing multiple rules. In addition, Progol fails to reconstruct two rules in the health care sample policy. Due to Progol's failure to learn rules with conjuncts or operation sets containing multiple constants, we generated a new set of 20 synthetic policies with at most 1 constant per conjunct and 1 operation per rule. On these policies with $N_{\mathrm{rule}} = 5$, our algorithm achieves a compression factor of 1.92, compared to 1.67 for Progol.

Progol is much slower than our algorithm. For the university (with $N_{\mathrm{dept}} = 10$), health care (with $N_{\mathrm{ward}} = 20$), and project management (with $N_{\mathrm{dept}} = 20$) sample policies, Progol is 302, 375, and 369 times slower than our algorithm, respectively. For synthetic policies with $N_{\mathrm{rule}} = 5$, Progol is 2.74 times slower than our algorithm; for synthetic policies with $N_{\mathrm{rule}} = 10$, we stopped Progol after several hours.

4.5 Related Work

To the best of our knowledge, the algorithm in this paper is the first policy mining algorithm for any ABAC framework. Existing algorithms for access control policy mining produce role-based policies; this includes algorithms that use attribute data, e.g., [MCL+10, CDV12, XS12]. Algorithms for mining meaningful RBAC policies from ACLs and user attribute data [MCL+10, XS12] attempt to produce RBAC policies that are small (i.e., have low WSC) and contain roles that are meaningful in the sense that the role's user membership is close to the meaning of some user attribute expression. User names (i.e., values of uid) are used in role membership definitions and hence are not used in attribute expressions, so some sets of users cannot be characterized exactly by a user attribute expression. The resulting role-based policies are often much larger than attribute-based policies, due to the lack of parameterization; for example, they require separate roles for each department in an organization, in cases where a single rule suffices in an attribute-based policy. Furthermore, algorithms for mining meaningful roles does not consider resource attributes (or permission attributes), constraints, or set relationships.

Xu and Stoller's work on mining parameterized RBAC (PRBAC) policies [XS13b] is more closely related. Their PRBAC framework supports a simple form of ABAC, because users and permissions have attributes that are implicit parameters of roles, the set of users assigned to a role is specified by an expression over user attributes, and the set of permissions granted to a role is specified by an expression over permission attributes. Our work differs from theirs in both the policy framework and the algorithm. Regarding the policy framework, our

ABAC framework supports a richer form of ABAC than their PRBAC framework does. Most importantly, our framework supports multi-valued (also called "set-valued") attributes and allows attributes to be compared using set membership, subset, and equality; their PRBAC framework does not support multi-valued attributes, and it allows attributes to be compared using only equality. Multi-valued attributes are very important in real policies. Due to the lack of multi-valued attributes, the sample policies in [XS13b] contain artificial limitations, e.g., a faculty teaches only one course, and a doctor is a member of only one medical team. Our extensions of their case studies do not have these limitations. In our sample policies, a faculty may teach multiple courses, a doctor may be a member of multiple medical teams, etc. Our algorithm works in a different, and more efficient, way than theirs. Our algorithm directly constructs rules to include in the output. Their algorithm constructs a large set of candidate roles and then determines which roles to include in the output, possibly discarding many candidates (more than 90% for their sample policies).

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [NLC+09]. Specifically, they use supervised machine learning algorithms to learn classifiers that associate permissions with roles, given as input the permissions, the roles, attribute data for the permissions, and (as training data) the role-permission assignment. The resulting classifier—a support vector machine (SVM)—can be used to automate assignment of new permissions to roles. They also consider a similar scenario in which a supervised machine learning algorithm is used to learn classifiers that associate users with roles, given as input the users, the roles, user attribute data, and the user-role assignment. The resulting classifiers are analogous to attribute expressions, but there are many differences between their work and ours. The largest difference is that their approach needs to be given the roles and the role-permission or user-role assignment as training data; in contrast, our algorithm does not require any part of the desired high-level policy to be given as input. Also, their work does not consider anything analogous to constraints, but it could be extended to do so. Exploring ABAC policy mining algorithms based on machine learning algorithms is a direction for future work.

Lim *et al.* investigated the use of evolutionary algorithms to learn and evolve security policies policies [Lim10]. They consider several problems, including difficult problems related to risk-based policies, but not general ABAC policy mining. In the facet of their work most similar to ABAC policy mining, they showed that genetic programming can learn the access condition in the Bell-LaPadula multi-level security model for mandatory access control. The learned predicate was sometimes syntactically more complex than, but logically equivalent to, the desired predicate.

Association rule mining has been studied extensively. Seminal work includes Agrawal *et al.*'s algorithm for mining propositional rules [AS94]. Association rule mining algorithms are not well suited to ABAC policy mining, because they are designed to find rules that are probabilistic in nature [AS94] and are supported by statistically strong evidence. They are not designed to produce a set of rules that are strictly satisfied, that completely cover the input data, and are minimum-sized among such sets of rules. Consequently, unlike our algorithm, they do not give preference to smaller rules or rules with less overlap (to reduce overall policy size).

Bauer *et al.* use association rule mining to detect policy errors [BGR08]. They apply propositional association rule mining to access logs to learn rules expressing that a user

who exercised certain permissions is likely to exercise another permission. A suspected misconfiguration exists if a user who exercised the former permissions does not have the latter permission. Our under-assignment detection follows a similar principle. Bauer *et al.* do not consider attribute data or generate entire policies.

Inductive logic programming (ILP) is a form of machine learning in which concepts are learned from examples and expressed as logic programs. ABAC policies can be represented as logic programs, so ABAC policy mining can be seen as a special case of ILP. However, ILP systems are not ideally suited to ABAC policy mining. ILP is a more difficult problem, which involves learning incompletely specified relations from a limited number of positive and negative examples, exploiting background knowledge, etc. ILP algorithms are correspondingly more complicated and less scalable, and focus more on how much to generalize from the given examples than on optimization of logic program size. For example, Progol (*cf.* Section 4.4.5) uses a compression (rule size) metric to guide construction of each rule but does not attempt to achieve good compression for the learned rules collectively; in particular, it does not perform steps analogous to merging rules, eliminating overlap between rules, and selecting the highest-quality candidate rules for the final solution. As the experiments in Section 4.4.5 demonstrate, Progol is slower and generally produces policies with higher WSC, compared to our algorithm.

N_{rule}	$N_{\text{cnj}}^{\text{min}}$	$\lvert U \rvert$		$\lvert R \rvert$		$\lvert UP \rvert$		$\widehat{[\![\rho]\!]}$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
10	4	206	16	62	5.7	401	90	35	9.2	.069	.01	.63	0.04	0.011	1.79	.21	.060	0.30	0.31	0.09
	2					620	116	136	29	.076	.01	.69	0.03	0.009	1.55	.18	.051	0.47	0.19	0.05
	0					1025	222	298	62	.081	.01	.78	0.05	0.014	1.12	.16	.045	1.02	0.42	0.12
50	4	1008	38	318	15	1975	282	36	5.3	.014	.001	.62	0.02	0.006	1.75	.08	.023	4.86	1.71	0.49
	2					3314	526	144	20	.015	.001	.68	0.02	0.006	1.52	.08	.023	11.78	3.41	0.97
	0					11969	5192	438	136	.025	.007	.75	0.03	0.009	0.84	.18	.051	58.88	18.7	5.31

Table 4.1: Experimental results for synthetic policies with varying $N_{\text{cnj}}^{\text{min}}$. "Synt. Sim." is syntactic similarity. "Compression" is the compression factor. μ is mean, σ is standard deviation, and CI is half-width of 95% confidence interval using Student's t-distribution. An empty cell indicates the same value as the cell above it.

N_{rule}	$N_{\text{cns}}^{\text{min}}$	$\|U\|$		$\|R\|$		$\|UP\|$		$\widehat{[\![\rho]\!]}$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
10	2	172	19	54	6.5	529	162	49	15	.084	.020	.72	.04	.011	1.50	.14	.040	0.28	0.17	0.05
	1					679	174	105	32	.093	.018	.72	.05	.014	1.43	.18	.051	0.36	0.17	0.05
	0					917	325	172	57	.110	.034	.70	.05	.014	1.30	.20	.057	0.49	0.20	0.06
50	2	781	51	276	16	3560	596	61	9.9	.020	.003	.67	.02	.006	1.29	.14	.040	12.72	3.95	1.12
	1					5062	1186	137	28	.024	.004	.66	.02	.006	1.15	.14	.040	16.78	5.09	1.45
	0					8057	2033	241	56	.031	.006	0.64	.02	.006	0.96	0.13	.037	23.38	6.78	1.93

Table 4.2: Experimental results for synthetic policies with varying $N_{\text{cns}}^{\text{min}}$.

N_{rule}	P_{over}	$\lvert U \rvert$		$\lvert R \rvert$		$\lvert UP \rvert$		$\widehat{[\![\rho]\!]}$		Density		Synt. Sim.			Compression			Time		
		μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	CI	μ	σ	CI	μ	σ	CI
50	0	693	37	247	15	5246	1445	92.3	30.0	.029	.006	.74	.02	.006	1.16	.11	.03	12.39	6.34	1.80
	0.5	655	53	216	16	7325	1838	333	69.6	.039	.008	.73	.03	.009	1.18	.21	.06	11.64	4.95	1.41
	1	664	58	216	16	7094	1473	563	109	.038	.006	.71	.03	.009	1.23	.29	.08	11.34	4.08	1.16

Table 4.3: Experimental results for synthetic policies with varying P_{over}.

Chapter 5

Mining Attribute-Based Access Control Policies from Role-Based Policies

In this chapter, we first formally define the problem of mining ABAC policies from RBAC policies and attribute data. An important feature of our problem definition is that it requires that some aspects of the structure of the RBAC policy be preserved in the ABAC policy. We then present an algorithm specifically designed to mine an ABAC policy from an RBAC policy and attribute data. To demonstrate the effectiveness of our algorithm, we also evaluate our algorithm on several manually written case studies. To show the significance of preserving the structure of the RBAC policy, we conduct experiments on variants of our case studies that have the same semantics (i.e., same user-permission relation) but different structure (i.e., different roles). Finally, we discuss related work.

5.1 Problem Definition

An RBAC policy π_{RBAC} is *semantically consistent* with an ABAC policy π if $[\![\pi_{\mathrm{RBAC}}]\!] = [\![\pi]\!]$.

Our goal is to mine an ABAC policy that is semantically consistent with the given RBAC policy and preserves the structure of the RBAC policy. A first thought is to require a 1-to-1 correspondence between roles and rules; in other words, for each role r, the mined policy contains a rule that covers the same user-permission tuples. However, this requirement is too strict, for two reasons. First, some roles cannot be expressed as a single rule, because the set of permissions granted by a rule must be expressible as the Cartesian product of a set of resources and a set of operations, while the set of permissions granted by a role can be arbitrary (although, in practice, it is often expressible as a Cartesian product). Second, it is often desirable to express multiple related roles by a single rule; for example, a set of roles, each granting certain permissions to staff in a particular department, can be expressed more concisely by a single rule that uses a constraint to ensure that each user is granted permissions appropriate to his or her department. Therefore, we relax this requirement in two ways. First, we split the given roles, so that each role's set of assigned permissions is the Cartesian product of a set of resources and a set of operations, and we require a

correspondence between the resulting split roles and the mined rules. Second, we allow multiple roles to correspond to a single rule.

Given a set P of permissions, we want to express P as a sum (union) of Cartesian products. Let $\mathrm{ops}(P)$ be the set of operations that appear in P. Let $\mathrm{resources}(o, P)$ be the set of resources associated with o in P, i.e., $\{s \in \mathit{Res} \mid \langle s, o\rangle \in P\}$. Define two operations to be equivalent if they are associated with the same resources in P, i.e., $o \equiv_P o'$ iff $\mathrm{resources}(o, P) = \mathrm{resources}(o', P)$. Let S be a partition of $\mathrm{ops}(P)$ containing the equivalence classes of O with respect to $\equiv_P$. Define $\mathrm{SOP}(P) = \bigcup_{O\in S}\{\langle \mathrm{resources}(O), O\rangle\}$, where $\mathrm{resources}(O)$ is the set of resources associated with any operation in O (by definition, all operations in O are associated with the same resources). Note that $P = \bigcup_{\langle R,O\rangle\in \mathrm{SOP}(P)} R \times O$.

Given an RBAC policy $\pi_{\mathrm{RBAC}} = \langle U, \mathit{Res}, \mathit{Op}, \mathit{Roles}, \mathit{UA}, \mathit{PA}, \mathit{RH}\rangle$, the *sum-of-products policy* $\mathrm{SOP}(\pi_{\mathrm{RBAC}})$ is $\langle U, \mathit{Res}, \mathit{Op}, \mathit{Roles}', \mathit{UA}', \mathit{PA}', \mathit{RH}'\rangle$, where

$$\begin{aligned}
\mathit{Roles}' &= \bigcup_{r\in \mathit{Roles}} \; \bigcup_{\{R,O\}\in \mathrm{SOP}(\mathrm{assignedP}(r))} \{\langle r, R, O\rangle\} \\
\mathit{UA}' &= \bigcup_{\langle r,R,O\rangle\in \mathit{Roles}'} \mathrm{assignedU}(r) \times \{\langle r, R, O\rangle\} \\
\mathit{PA}' &= \bigcup_{\langle r,R,O\rangle\in \mathit{Roles}'} \{\langle r, R, O\rangle\} \times (R \times O) \\
\mathit{RH}' &= \{\langle\langle r, R, O\rangle, \langle r', R', O'\rangle\rangle \in \mathit{Roles}' \times \mathit{Roles}' \mid \\
&\qquad \langle r, r'\rangle \in \mathit{RH}\}
\end{aligned}$$

Note that we use tuples of the form $\langle r, R, o\rangle$ as role names in the sum-of-products policy. Note that $[\![\pi_{\mathrm{RBAC}}]\!] = [\![\mathrm{SOP}(\pi_{\mathrm{RBAC}})]\!]$. For a role r in a sum-of-products RBAC policy, let $\mathrm{asgndRes}(r) = \bigcup_{\langle r,o\rangle\in \mathrm{assignedP}(r)}\{r\}$ and $\mathrm{asgndOp}(r) = \bigcup_{\langle r,o\rangle\in \mathrm{assignedP}(r)}\{o\}$.

Given an RBAC policy $\pi_{\mathrm{RBAC}} = \langle U, \mathit{Res}, \mathit{Op}, \mathit{Roles}, \mathit{UA}, \mathit{PA}, \mathit{RH}\rangle$ and an ABAC policy $\pi = \langle U, \mathit{Res}, \mathit{Op}, A_{\mathrm{u}}, A_{\mathrm{r}}, d_{\mathrm{u}}, d_{\mathrm{r}}, \mathit{Rules}\rangle$, a *structural correspondence* between π_{RBAC} and π is an onto function κ from the roles in $\mathrm{SOP}(\pi_{\mathrm{RBAC}})$ whose authUP is non-empty to the rules in π such that, for each rule ρ, $[\![\rho]\!] = \bigcup_{r\in\kappa^{-1}(\rho)} \mathrm{authUP}(r)$, where κ^{-1} is the inverse of κ, i.e., $\kappa^{-1}(\rho)$ is the set of roles that map to rule ρ.

An ABAC policy is *structurally consistent* with an RBAC policy if there exists a structural correspondence between them.

Among ABAC policies semantically and structurally consistent with a given RBAC policy π_{RBAC}, which ones are preferable? One criterion is that policies that do not use the attributes uid and rid are preferable, because policies that use uid and rid are partly identity-based, not entirely attribute-based. Thus, an initial idea is to require that each of these attributes be used in the ABAC policy only if necessary, i.e., only if every ABAC policy that is semantically and structurally consistent with π_{RBAC} contains rules that use that attribute.

We refine this initial idea as follows. According to this initial idea, uid is used only when the information available from other attributes is insufficient to "explain" parts of the permission assignment, i.e., insufficient to characterize the sets of users that appear in the RBAC policy. In practice, this is likely to occur fairly often, because the available attribute information is often incomplete. However, rules that use uid to enumerate sets of users by their user identifiers are likely to be lower-level and harder to understand than the corresponding parts of the original RBAC policy. Therefore, we prohibit use of uid in the

ABAC policy, introduce a user attribute that expresses role membership, and allow this new user attribute to be used (instead of uid) when necessary to achieve semantic and structural consistency with the RBAC policy.

A *policy quality metric* is a function from ABAC policies to a totally-ordered set, such as the natural numbers. The ordering is chosen so that small values indicate high quality; this might seem counter-intuitive at first glance but is natural for metrics based on policy size.

The *ABAC-from-RBAC policy mining problem* is: given an RBAC policy $\pi_{\rm RBAC} = \langle U, \mathit{Res}, \mathit{Op}, \mathit{Roles}, \mathit{UA}, \mathit{PA}, \mathit{RH}\rangle$, attribute data $\langle A_{\rm u}, A_{\rm r}, d_{\rm u}, d_{\rm r}\rangle$, and a policy quality metric $Q_{\rm pol}$, find a set *Rules* of rules such that the ABAC policy $\pi = \langle U, \mathit{Res}, \mathit{Op}, A_{\rm u} \cup \{\text{roles}\}, A_{\rm r}, d'_{\rm u}, d_{\rm r}, \mathit{Rules}\rangle$ (1) is semantically and structurally consistent with $\pi_{\rm RBAC}$, (2) does not use uid, (3) uses roles and rid only when necessary, and (4) has the best quality, according to $Q_{\rm pol}$, among policies that satisfy conditions (1) through (3). Here, $d'_{\rm u}$ is $d_{\rm u}$ extended with a user attribute named roles defined by: $d'_{\rm u}(u, \text{roles}) = \{r \in \mathit{Roles}' \mid u \in \text{authU}(r)\}$. For simplicity, we assume $A_{\rm u}$ does not contain an attribute named roles.

For the policy quality metric, we use *weighted structural complexity* (WSC) [MCL+10], a generalization of policy size. The WSC of an ABAC policy is the WSC of the set *Rules* of rules in the policy, defined by

$$\begin{aligned}
\text{WSC}(e) &= \sum_{a \in \text{attr}_1(e)} |e(a)| + \sum_{a \in \text{attr}_{\rm m}(e), s \in e(a)} |s| \\
\text{WSC}(\langle e_u, e_p, O, c\rangle) &= w_1 \text{WSC}(e_u) + w_2 \text{WSC}(e_p) + w_3|O| + w_4|c| \\
\text{WSC}(\mathit{Rules}) &= \sum_{\rho \in \mathit{Rules}} \text{WSC}(\rho)
\end{aligned}$$

where $|s|$ is the cardinality of set s, and the w_i are user-specified weights. In the experiments in Section 5.3, all weights equal 1.

5.2 Policy Mining Algorithm

Let the inputs to the algorithm be denoted as in the problem statement. Let $\pi'_{\rm RBAC} = \langle U, \mathit{Op}, R, \mathit{Roles}', \mathit{UA}', \mathit{PA}', \mathit{RH}'\rangle$ be the sum-of-products policy for $\pi_{\rm RBAC}$. Top-level pseudo-code for our policy mining algorithm appears in Figure 5.1. It calls several functions, described next.

The function computeUAE(s, U) in Figure 5.1 computes a user-attribute expression $e_{\rm u}$ that characterizes the set s of users. Preference is given to attribute expressions that do not use uid, as discussed in Section 3.2. After constructing a candidate expression e, it calls elimRedundantSets(e), which is also defined in Figure 5.1. elimRedundantSets(e) attempts to lower the WSC of e by examining the conjunct for each multi-valued user attribute, and removing each set that is a superset of another set in the same conjunct; this leaves the meaning of the rule unchanged, because $\supseteq$ is used in the condition for multi-valued attributes in the semantics of user attribute expressions. For uniformity with other functions described below, elimRedundantSets(e) returns false. The expression $e_{\rm u}$ returned by computeUAE might not be minimum-sized among expressions that characterize s: it is possible that some attributes mapped to a set of values by $e_{\rm u}$ can instead be mapped to $\top$.

```
// Rules is the set of rules
1: Rules = ∅
// κ is the structural correspondence
2: κ = ∅
3: for r in Roles'
4:     if authUP(r).empty()
5:         continue
6:     end if
   // create a rule corresponding to r
7:     e_u = computeUAE(authU(r))
8:     e_r = computeRAE(asgndRes(r))
9:     O = asgndOp(r)
10:    cc = ⋂_{u∈authU(r),s∈asgndRes(r)}
           candidateConstraint(u, s)
11:    ρ = ⟨e_u, e_r, O, cc⟩
12:    Rules.add(ρ)
13:    κ.add(⟨r, ρ⟩)
14: end for
// Rules is semantically and structurally
// consistent with π_RBAC. Try to improve
// its quality, by repeatedly merging and
// simplifying rules, until this has no
// effect.
15: mergeRules(Rules, κ)
16: while simplifyRules(Rules, κ)
17:    if not mergeRules(Rules, κ)
18:        break
19:    end if
20: end while
21: useRoleAttribute(Rules, κ)
22: return ⟨Rules, κ⟩
```

```
function computeUAE(s)
// Try to characterize s without using
// uid. Use all other attributes which
// have known values for all users in s.
1: e = (λ a ∈ A_u.
       a = uid ∨ (∃u ∈ s. d_u(u, a) = ⊥)
       ? ⊤ : ⋃_{u∈s} d_u(u, a))
2: if ⟦e⟧_U ≠ s
       // uid is needed to characterize s
3:     e(uid) = ⋃_{u∈s} d_u(u, uid)
4: end if
5: elimRedundantSets(e)
6: return e
```

```
function elimRedundantSets(e)
1: for a in attr_m(e)
2:     for s in e(a)
3:         if (∃s' ∈ e(a). s' ⊂ s)
4:             e(a).remove(s)
5:         end if
6:     end for
7: end for
8: return false
```

Figure 5.1: Left: Top-level pseudocode for policy mining algorithm. Right: computeUAE(s) computes a user-attribute expression that characterizes set s of users.

The function computeRAE is defined in the same way as computeUAE, except using resource attributes instead of user attributes, and the call to elimRedundantSets is omitted.

The function candidateConstraint(u, s) returns a set containing all the atomic constraints that hold between user u and resource s.

The function mergeRules($Rules, \kappa$) in Figure 5.2 attempts to reduce the WSC of *Rules*, while preserving semantic and structural consistency, by removing redundant rules and merging pairs of rules. A rule ρ is subsumed by a role ρ' if $[\![\rho]\!] \subseteq [\![\rho']\!]$. A rule in *Rules* is redundant if it subsumed by another rule in *Rules*. Informally, rules ρ_1 and ρ_2 are merged by taking, for each attribute, the union of the conjuncts in ρ_1 and ρ_2 for that attribute. If adding the

resulting rule ρ_{merge} and removing rules subsumed by ρ_{merge} (including ρ_1 and ρ_2) preserves structural consistency, then these changes are made to *Rules*, and the structural correspondence f is updated accordingly. mergeRules(*Rules*, κ) updates *Rules* and κ in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(*Rules*, κ) attempts to simplify all of the rules in *Rules*. It updates its arguments *Rules* and κ in place, replacing rules in *Rules* with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several ways, which are embodied in the following simplification functions that it calls. Generally, each of these simplification functions return a Boolean indicating whether changes were made; this information is used in the top-level pseudocode in Figure 5.1 to determine whether another iteration of merging and simplification is necessary. The function elimRedundantSets is described above. It returns false, even if some redundant sets were eliminated, because elimination of redundant sets does not affect the meaning or mergeability of rules, so it should not trigger another iteration of merging and simplification. The function elimConjuncts(ρ, *Rules*, κ, *UP*) attempts to increase the quality of rule ρ by eliminating some conjuncts. It calls the function elimConjunctsHelper(ρ, A, *Rules*, *UP*), which considers all rules that differ from ρ by mapping a subset A' of the tagged attributes in A to $\top$ instead of to a set of values. For each of the resulting rules ρ', elimConjunctsHelper checks whether ρ' can replace the rules that it subsumes, i.e., whether ρ' has exactly the same meaning as the set of rules it subsumes. Among the resulting rules that satisfy this condition, elimConjunctsHelper returns one with the highest quality. A *tagged attribute* is a pair of the form $\langle$"user"$, a\rangle$ with $a \in A_{\text{u}}$ or $\langle$"res"$, a\rangle$ with $a \in A_{\text{r}}$. The set A_{unrm} in function elimConjuncts is a set of *unremovable* tagged attributes; it is a parameter of the algorithm, specifying attributes that should not be eliminated, because eliminating them increases the risk of generating an overly general policy, i.e., a policy that might grant inappropriate permissions when new users or new resources (hence new permissions) are added to the system. We use a combinatorial algorithm for elimConjuncts that evaluates all combinations of conjuncts that can be eliminated, because elimination of one conjunct might prevent elimination of another conjunct. This algorithm makes elimConjuncts worst-case exponential in the numbers of user attributes and resource attributes that can be eliminated while preserving validity of the rule; in practice the number of such attributes is small, and elimConjuncts is fast. The function elimConstraints(ρ, *Rules*, κ, *UP*) attempts to improve the quality of ρ by removing unnecessary atomic constraints from ρ's constraint. An atomic constraint is *unnecessary* in a rule ρ if removing it from ρ's constraint leaves ρ valid. The loop over i in elimConstraintsHelper considers all possibilities for the first atomic constraint in *cc* that gets removed from the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in *cc* that get removed from the constraint. The function elimElements(ρ, *Rules*, κ) attempts to decrease the WSC of rule ρ by removing elements from sets in conjuncts for multi-valued user attributes, if removal of those elements produces a rule ρ' that can replace the rules it subsumes; note that, because $\subseteq$ is used in the semantics of user attribute expressions, the set of user-permission pairs that satisfy a rule is unchanged or increased (never decreased) by such removals. It would be reasonable to use a combinatorial algorithm for elimElements, in the same style as elimConjuncts and elimConstraints, because elimination of one set element can prevent elimination of another. We decided to use a simple linear algorithm for this function, for simplicity and because it

```
function mergeRules(Rules, κ)
// remove redundant rules
1: for ρ in Rules
2:     if ∃ρ' ∈ Rules \ {ρ}. ⟦ρ⟧ ⊆ ⟦ρ'⟧
3:         Rules.remove(ρ)
4:         for each r in κ^{-1}(ρ)
5:             κ(r) = ρ'
6:         end for
7:     end if
8: end for
// merge rules
9: merged = false
10: workSet = {(ρ_1, ρ_2) | ρ_1 ∈ Rules
                            ∧ ρ_2 ∈ Rules
                            ∧ ρ_1 ≠ ρ_2
                            ∧ con(ρ_1) = con(ρ_2)}
11: while not(workSet.empty())
        // remove an arbitrary element
        // of the workset
12:     (ρ_1, ρ_2) = workSet.remove()
13:     ρ_merge = ⟨uae(ρ_1) ∪ uae(ρ_2),
                  rae(ρ_1) ∪ rae(ρ_2),
                  ops(ρ_1) ∪ ops(ρ_2), con(ρ_1)⟩
14:     subsumed = findSubsumed(ρ, Rules)
15:     if ⟦ρ_merge⟧ = ⋃_{ρ∈subsumed} ⟦ρ⟧
16:         merged = true
17:         replaceRules(subsumed, ρ_merge, Rules, κ)
18:         remove pairs in workSet that contain
19:             an element of subsumed
20:         workSet.addAll({(ρ_merge, ρ) |
                            ρ ∈ Rules
                            ∧ρ ≠ ρ_merge
                         ∧ con(ρ) = con(ρ_merge)})
21:     end if
22: end while
23: return merged
```

```
// find rules in Rules subsumed by ρ
function findSubsumed(ρ, Rules)
1: subsumed = {ρ_1, ρ_2}
2: for ρ in Rules \ {ρ_1, ρ_2}
3:     if ⟦ρ⟧ ⊆ ⟦ρ_merge⟧
4:         subsumed.add(ρ)
5:     end if
6: end for
7: return subsumed
```

```
// replace the rules in S with ρ;
// update Rules and κ accordingly.
function replaceRules(S, ρ, Rules, κ)
1: for ρ in subsumed
2:     Rules.remove(ρ)
3:     for each r in κ^{-1}(ρ)
4:         κ(r) = ρ_merge
5:     end for
6: end for
7: Rules.add(ρ_merge)
```

Figure 5.2: Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$.

is likely to give the same results, because elimElements usually eliminates only 0 or 1 set elements per rule in our experiments. Pseudocode for simplification-related functions appears in Figures 5.3–5.4.

The function useRoleAttribute($Rules, \kappa$) in Figure 5.4 replaces uses of "uid" with uses of the user attribute "roles", whose values are defined as described in the policy mining problem definition.

Iteration Order In the iterations over *Rules* in mergeRules and simplifyRules, the order in which rules are processed is deterministic in our implementation, because *Rules* is implemented as an arraylist, loops iterate over the rules in the order they appear in the arraylist, and newly generated rules are added at the end of the arraylist. In the first loop in mergeRules, if multiple rules ρ' subsume ρ, we choose the first such rule in *Rules* as the witness for the existential quantifier.

5.3 Evaluation

We evaluated our algorithm on manually written case studies. Experiments with a real RBAC policy and real attribute data would be better, but unfortunately, we do not have access to such information. The policies are small but non-trivial and realistic.

Due to space limitations, only brief descriptions of the case studies are included here. Full details, including all input files and output files, are available at `http://www.cs.stonybrook.edu/~stoller/abac-from-rbac/`. The ABAC policies for the case studies are similar to those in [XS13a].

5.3.1 Experiments with Full Attribute Data

These experiments demonstrate that, when all relevant attribute data is available, our algorithm successfully produces an intuitive high-level ABAC policy from an RBAC policy. We manually wrote semantically consistent case study policies in RBAC and ABAC, applied our algorithm to the RBAC policy and accompanying attribute data, and compared the generated ABAC policy with the manually written one.

University Case Study Our university case study is a policy that controls access to applications (for admission), gradebooks, transcripts, and course schedules. There are roles for students in each course, TAs of each course, instructor of each course, chairman of each department, registrar staff, admissions staff, and applicants for admission. The permission assignment allows a student to read his/her transcript, an instructor to assign grades for courses he/she teaches, etc. In the attribute data, user attributes include position (applicant, student, faculty, or staff), department, set of courses taken (for students), set of courses taught or TA-ed, and whether the user is department chair. Resource attributes include resource type (application, gradebook, roster, or transcript), course (for gradebooks and rosters), student (for transcripts and applications), and department. With no guidance (i.e., no attributes are declared unremovable), the generated ABAC policy is almost identical to the manually written ABAC policy, with a 1-to-1 correspondence between rules in the two

```
function simplifyRules(Rules, κ)
1: changed = false
2: for ρ in Rules
3:     changed =
           changed
           ∨ elimRedundantSets(uae(ρ))
           ∨ elimElements(ρ, κ)
           ∨ elimConjuncts(ρ, Rules, κ, UP_0)
4: end for
5: for ρ in Rules
6:     changed =
           changed
           ∨ elimConstraints(ρ, Rules, κ, UP_0)
7: end for
8: return changed

function elimConjuncts(ρ, Rules, κ, UP)
1: A = {"user"} × (.uae(ρ))
       ∪ {"res"} × (.rae(ρ))
2: A = A \ A_unrm
3: ρ_1 = elimConjunctsHelper(ρ, A, Rules,
       UP)
4: if ρ_1 ≠ ρ
5:     replaceRules(findSubsumed(ρ_1, Rules),
                    ρ_1, Rules, κ)
6:     return true
7: else
8:     return false
9: end if

function elimConjunctsHelper(ρ, A, Rules, UP)
1: ρ_best = ρ
// discard tagged attributes ta in A such that
// elimination of the conjunct for ta makes ρ
// invalid.
2: for ta in A
3:     ρ' = elimAttribute(ρ, ta)
4:     if not [[ρ']] ⊆ UP_0
5:         A.remove(ta)
6:     end if
7: end for
// we treat the set A as an array.
8: for i = 1 to A.length
9:     ρ_1 = elimAttribute(ρ, A[i])
10:    ρ_2 = elimConjunctsHelper(ρ_1,
           A[i+1 .. A.length], Rules, UP)
11:    if [[ρ_2]] = ⋃_{ρ'∈findSubsumed(ρ_2,Rules)} [[ρ']]
           ∧ Q_rul(ρ_2, UP) > Q_rul(ρ_best, UP)
12:        ρ_best = ρ_2
13:    end if
14: end for
15: return ρ_best

function elimAttribute(⟨e_u, e_r, O, c⟩, ta)
1: match ta with
2:     ⟨"user", a⟩ → return ⟨e_u[a ↦ ⊤], e_r, O, c⟩
3:     ⟨"res", a⟩ → return ⟨e_u, e_r[a ↦ ⊤], O, c⟩
4: end match
```

Figure 5.3: Functions used to simplify rules.

policies, and all corresponding rules being identical except for one rule, which has small differences. If resource type is specified as an unremovable attribute, the generated policy is identical to the manually written ABAC policy.

Health Care Case Study Our health care case study is a policy that controls access to electronic health records (HRs) and HR items (i.e., entries in health records). There are roles for nurses in each ward (e.g., oncology ward), each medical team, each medical specialty on each medical team (e.g., oncologists on team 1), each patient, and agents for each patient. The permission assignment allows a nurse to add note items in health records for patients in the ward he/she works in, a patient and his/her agents to read note items in the patient's medical record, members of a medical team to read items appropriate to their medical specialty in health records of patients treated by that team, etc. In the attribute

function elimConstraints(ρ, *Rules*, κ, *UP*)
1: ρ_1 = elimConstraintsHelper(ρ, con(ρ), *Rules*, *UP*)
2: **if** $\rho_1 \neq \rho$
3:     replaceRules(findSubsumed(ρ_1, *Rules*), ρ_1, *Rules*, κ)
4:     **return** true
5: **else**
6:     **return** false
7: **end if**

function elimConstraintsHelper(ρ, *cc*, *Rules*, *UP*)
1: $\rho_{\text{best}} = \rho$
// discard formulas that, when removed
// from ρ, produce invalid rules
2: **for** f **in** *cc*
3:     $\rho' = \langle \text{uae}(\rho), \text{rae}(\rho), \text{ops}(\rho), \text{con}(\rho) \setminus \{f\} \rangle$
4:     **if** $[\![\rho']\!] \not\subseteq UP_0$
5:         remove f from *cc*
6:     **end if**
7: **end for**
// we treat the set *cc* as an array.
8: **for** i=1 to *cc*.length
9:     $\rho_1 = \langle \text{uae}(\rho), \text{rae}(\rho), \text{ops}(\rho), \text{con}(\rho) \setminus \{cc[i]\} \rangle$
10:     ρ_2 = elimConstraintsHelper(ρ_1, $cc[i{+}1 \,..\, cc.\text{length}]$, *Rules*, *UP*)
11:     **if** $[\![\rho_2]\!] = \bigcup_{\rho' \in \text{findSubsumed}(\rho_2, Rules)} [\![\rho']\!] \wedge Q_{\text{rul}}(\rho_2, UP) > Q_{\text{rul}}(\rho_{\text{best}}, UP)$
12:         $\rho_{\text{best}} = \rho_2$
13:     **end if**
14: **end for**
15: **return** ρ_{best}

function elimElements(ρ, *Rules*, κ)
1: *changed* = false
2: $e_{\text{u}} = \text{uae}(\rho)$
3: **for** a **in** $A_{\text{u,m}}$
4:     **for** s **in** $e_{\text{u}}(a)$
5:         **for** v **in** s
            // try to remove v from s
6:             ρ_1 = copy of ρ with v removed from s
7:             *subsumed* = findSubsumed(ρ_1, *Rules*)
8:             **if** $[\![\rho_1]\!] = \bigcup_{\rho' \in subsumed} [\![\rho']\!]$
9:                 replaceRules(*subsumed*, ρ_1, *Rules*, κ)
10:                 *changed* = true
11:             **end if**
12:         **end for**
13:     **end for**
14: **end for**
15: **return** *changed*

function useRoleAttribute(*Rules*, κ)
1: **for** ρ **in** *Rules*
2:     **if** $\text{uae}(\rho)(\text{uid}) \neq \top$
3:         $s = \{\{r\} \mid r \in \kappa^{-1}(\rho)\}$
4:         $\rho = \langle \text{uae}(\rho)[\text{uid} \to \top, \text{roles} \to s], \text{rae}(\rho), \text{con}(\rho) \rangle$
5:     **end if**
6: **end for**

Figure 5.4: Functions used to simplify rules (continued) and function useRoleAttribute.

data, user attributes include position (doctor or nurse; for other users, this attribute equals $\perp$), the user's medical specialties, teams the user is a member of, ward, and patients for which the user is an agent. Resource attributes include resource type (HR or HR item), the patient that the HR or HR item is for, medical teams treating that patient, ward housing that patient, author, and topics. With no guidance (i.e., no attributes are declared unremovable), the generated ABAC policy is very similar to the manually written ABAC policy, with a 1-to-1 correspondence between rules in the two policies, and with small differences between

some corresponding rules. If resource type is specified as an unremovable attribute, the only remaining difference is that one rule in generated policy has an additional conjunct that reduces overlap with another rule.

Project Management Case Study Our project management case study is a policy that controls access to budgets, schedules, and tasks associated with projects. There are roles for the manager of each department; for the accountants, auditors, planners, leaders, designers, and coders working on each project; and for the designers and coders assigned to each task. The roles also distinguish employees from non-employees (contractors). Role hierarchy is used to combine the roles for users of each specialty working on a project into a role for all users working on the project. The permission assignment allows a user working on a project to read the project schedule, a user working on a task to update the status of the task, a non-employee working on a project to read information about non-proprietary tasks in that project that match his/her technical expertise, etc. In the attribute data, user attributes include the set of projects the user is working on, the projects led by the user, the user's administrative roles (e.g., accountant, auditor), the user's areas of technical expertise, tasks assigned to the user, department, and whether the user is an employee. Resource attributes include resource type (task, schedule, or budget), project, department, areas of technical expertise required to work on the task, and whether the task involves proprietary information. With no guidance (i.e., no attributes are declared unremovable), the generated ABAC policy is very similar to the manually written ABAC policy, with a 1-to-1 correspondence between rules in the two policies, and with small differences between some corresponding rules. If resource type is specified as an unremovable attribute, the only remaining difference is that one rule in generated policy has an additional conjunct that reduces overlap with another rule.

5.3.2 Experiments with Incomplete Attribute Data

These experiments demonstrate that, when some relevant attribute information is unavailable, our algorithm successfully produces an intuitive high-level ABAC policy that uses the available attribute data and uses role membership information as a substitute for missing attribute data.

For the health care case study, we deleted the user attribute data specifying which users are agents for which patients; this data seems less essential to the hospital's IT system, and hence more likely to be unavailable, than employee-related user attribute data. With this input, the generated ABAC policy is mostly identical to the ABAC policy generated with full attribute data (as described above): rules unrelated to agents are unaffected, while rules granting permissions to agents are replaced with similar rules that use agent roles instead of the "agent for" attribute. The number of agent-related rules increases, because a separate rule is needed for each patient's agents.

For the university case study, we deleted the user attribute data specifying whether a user is a department chair. As expected, only the rule granting permissions to department chairs is affected, and the only change in that rule is replacement of the conjunct "isChair=true" in the user attribute expression with the conjunct "role supseteqln {{eeChair}, {csChair}}".

π_{R1} :
UA(csStudent, {...})
UA(eeStudent, {...})
RH(student, csStudent)
RH(student, eeStudent)
PA(csStudent, {<csServer, runApp>})
PA(eeStudent, {<eeServer, runApp>})
PA(student, {<univServer, runApp>})

π_{R2} :
UA(csStudent, {...})
UA(eeStudent, {...})
PA(csStudent, { <univServer, runApp>,
<csServer, runApp>})
PA(eeStudent, { <univServer, runApp>,
<eeServer, runApp>})

π_{A1} :
rule(dept=cs; type=csServer; {runApp};)
rule(dept=ee; type=eeServer; {runApp};)
rule(dept in {cs, ee}; type=univServer;
{runApp};)

π_{A2} :
rule(dept in {cs}; type in {univServer,
csServer}; {runApp};)
rule(dept in {ee}; type in {univServer,
eeServer}; {runApp};)

Figure 5.5: Department/university server example.

5.3.3 Experiments with Varying Policy Structure

These experiments demonstrate how the structure of the RBAC policy propagates into the structure of the generated ABAC policy. As a small initial example, consider the RBAC policies π_{R1} and π_{R2} in Figure 5.5, which control students' permission to run some application on departmental and university servers (any other appropriate operation, e.g., upload a file, could be used instead). These policies have the same user-permission relation but different structure. π_{R2} has lower WSC than π_{R1}, but π_{R1} might be preferable for other reasons, for example, if rules that grant permissions on university servers are administered by the IT Department, and rules that grant permissions on a departmental server is administered by the owning department. Assuming suitable attribute data (a user attribute "dept" indicating the user's department, etc.), our algorithm applied to π_{R1} produces the ABAC policy π_{A1}, which has the same structure as π_{R1} and hence can be administered in the same way. In contrast, our algorithm applied to π_{R2} produces the ABAC policy π_{A2}, which has lower WSC than π_{A1} but cannot be administered in the same way as π_{A1}.

For a second example, consider the fragment of the university case study that grants permissions on gradebooks to the staff for each course, i.e., the instructor and TAs. Figures 5.6 and 5.7 show two ways of expressing these permissions in RBAC (PA statements for only one representative course are shown) and the corresponding ABAC rules generated by our algorithm. Informally, in Figure 5.6, the roles (and hence the rules) are organized by operation, while the roles in Figure 5.7 are organized by the user's position (instructor or TA). The algorithm in [XS13a], applied to the ACL expansion of either RBAC policy, produces the rules in Figure 5.6, which have lower WSC.

5.4 Related Work

To the best of our knowledge, this paper presents the first algorithm specifically designed to mine ABAC policies from RBAC policies and attribute data, and the only prior work

PA(cs101Staff, { <cs101gradebook, addScore>, <cs101gradebook, readScore>})
PA(cs101Instructor, { <cs101gradebook, changeScore>,
<cs101gradebook, assignGrade>})

// a user (instructor or TA) can add scores and read scores in gradebooks
// for taught courses
rule(; type=gradebook; {addScore, readScore}; crsTaught $\ni$ crs)
// the instructor for a course can change scores and assign grades in the
// course's gradebook.
rule(position=faculty; type=gradebook; {changeScore, assignGrade};
crsTaught $\ni$ crs)

Figure 5.6: Gradebook permissions in university case study, version 1.

PA(cs101TA, { <cs101gradebook, addScore>, <cs101gradebook, readScore>})
PA(cs101Instructor, { <cs101gradebook, addScore>, <cs101gradebook, readScore>,
<cs101gradebook, changeScore>, <cs101gradebook, assignGrade>})

// a TA can add scores and read scores in gradebooks for taught courses
rule(position=student; type=gradebook; {addScore, readScore}; crsTaught $\ni$ crs)
// the instructor for a course can change scores and assign grades in the
// course's gradebook.
rule(position=faculty; type=gradebook; {addScore, readScore, changeScore
, assignGrade}; crsTaught $\ni$ crs)

Figure 5.7: Gradebook permissions in university case study, version 2.

on mining ABAC policies is the algorithm of Xu and Stoller [XS13a] that mines ABAC policies from ACLs and attribute data. There are significant differences in the workings of the algorithm, including (1) the algorithm in this paper constructs rules corresponding to roles, while the algorithm in [XS13a] constructs rules corresponding to user-permission tuples and then attempts to generalize the resulting rules, and (2) all of the rules constructed by the algorithm in this paper either get merged with other rules or included in the output, while the algorithm in [XS13a] contains with a selection phase that may discard many of the constructed rules. Despite these differences, the algorithm in this paper builds on the work in [XS13a]; specifically, the functions computeUAE, computeRAE, and elimRedundantSets are the same as in [XS13a], and the functions mergeRules, simplifyRules and the "elim" functions called by simplifyRules are similar to the corresponding functions in [XS13a] but with important changes in (1) the condition used to decide when to perform a merge or elimination operation and (2) the code used to update the ABAC policy after a merge or elimination operation. The algorithm in [XS13a] can be used to mine ABAC policies from RBAC policies (and attribute data), by expanding RBAC policies into ACLs. However, that approach has significant disadvantages compared to the algorithm presented in this paper, mainly (1) the generated ABAC policy is less likely to have the desired structure, because the structure of the RBAC policy is not used to guide the structure of the ABAC policy, and (2)

role membership information is not used to substitute for unavailable attribute information, leading to lower-level policies that use user identity instead of role membership information where the available attribute information is insufficient.

The next most closely related work is Xu and Stoller's algorithm for mining parameterized RBAC (PRBAC) policies from ACLs and attribute data [XS13b]. In their PRBAC framework, users and permissions have attributes that are implicit parameters of roles, the set of users assigned to a role is specified by an expression over user attributes, and the set of permissions granted to a role is specified by an expression over permission attributes. Thus, their PRBAC framework supports a simple form of ABAC, but quite limited compared to our ABAC framework. Most importantly, our framework supports multi-valued (also called "set-valued") attributes and allows attributes to be compared using set membership, subset, and equality; their PRBAC framework does not support multi-valued attributes, and it allows attributes to be compared using only equality. The differences in input policy language (ACL vs. RBAC) and output policy language (PRBAC vs. ABAC) naturally lead to significant differences between the algorithms.

Less closely related work includes policy mining algorithms that take attribute data into account when mining RBAC policies (without parameters) from ACLs, e.g., [MCL+10, CDV12, XS12].

Chapter 6

Mining Attribute-Based Access Control Policies from Logs

In previous chapters, we assume that the entire user-permission relation is available in some form (such as ACLs or RBAC). However, an ACL policy or RBAC policy might not be available, e.g., if the current access control policy is encoded in a program or is not enforced by a computerized access control mechanism. An alternative source of information about the current access control policy is operation logs, or "logs" for short. Many software systems produce logs, e.g., for auditing, accounting, and accountability purposes. Molloy, Park, and Chari proposed the idea of mining policies from logs and developed algorithms for mining RBAC policies from logs [MPC12b]. This paper presents an algorithm for mining ABAC policies from logs and attribute data.

6.1 Problem Definition

An *operation log entry* e is a tuple $\langle u, r, o, t \rangle$ where $u \in U$ is a user, $r \in R$ is a resource, $o \in Op$ is an operation, and t is a timestamp. An *operation log* is a sequence of operation log entries. The user-permission relation induced by an operation log L is $UP(L) = \{\langle u, r, o \rangle \mid \exists t.\ \langle u, r, o, t \rangle \in L\}$.

The input to the *ABAC-from-logs policy mining problem* is a tuple $I = \langle U, R, Op, A_{\rm u}, A_{\rm r}, d_{\rm u}, d_{\rm r}, L \rangle$, where U is a set of users, R is a set of resources, Op is a set of operations, $A_{\rm u}$ is a set of user attributes, $A_{\rm r}$ is a set of resource attributes, $d_{\rm u}$ is user attribute data, $d_{\rm r}$ is resource attribute data, and L is an operation log, such that the users, resources, and operations that appear in L are subsets of U, R, and Op, respectively. The goal of the problem is to find a set of rules *Rules* such that the ABAC policy $\pi = \langle U, R, Op, A_{\rm u}, A_{\rm r}, d_{\rm u}, d_{\rm r}, Rules \rangle$ maximizes a suitable policy quality metric.

The policy quality metric should reflect the size and meaning of the policy. Size is measured by *weighted structural complexity* (WSC) [MCL+10], and smaller policies are considered to have higher quality. This is consistent with usability studies of access control rules, which conclude that more concise policies are more manageable. Informally, the WSC of an ABAC policy is a weighted sum of the number of elements in the policy. Specifically, the WSC of an

attribute expression is the number of atomic values that appear in it, the WSC of an operation set is the number of operations in it, the WSC of a constraint is the number of atomic constraints in it, and the WSC of a rule is a weighted sum of the WSCs of its components, namely, $\mathrm{WSC}(\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle) = w_1\mathrm{WSC}(e_{\mathrm{u}}) + w_2\mathrm{WSC}(e_{\mathrm{r}}) + w_3\mathrm{WSC}(O) + w_4\mathrm{WSC}(c)$, where the w_i are user-specified weights. The WSC of a set of rules is the sum of the WSCs of its members.

The meaning $[\![\pi]\!]$ of the ABAC policy is taken into account by considering the differences from $UP(L)$, which consist of over-assignments and under-assignments. The over-assignments are $[\![\pi]\!] \setminus UP(L)$. The under-assignments are $UP(L) \setminus [\![\pi]\!]$. Since logs provide only a lower-bound on the actual user-permission relation (a.k.a entitlements), it is necessary to allow some over-assignments, but not too many. Allowing under-assignments is beneficial if the logs might contain noise, in the form of log entries representing uses of permissions that should not be granted, because it reduces the amount of such noise that gets propagated into the mined policy; consideration of noise is left for future work. We define a policy quality metric that is a weighted sum of these aspects:

$$Q_{\mathrm{pol}}(\pi, L) = \mathrm{WSC}(\pi) + w_o \, |[\![\pi]\!] \setminus UP(L)| \, / \, |U| \tag{6.1}$$

where the *policy over-assignment weight* w_o is a user-specified weight for over-assignments, and for a set S of user-permission tuples, the frequency-weighted size of S with respect to log L is $|S|_L = \sum_{\langle u,r,o\rangle \in S} \mathrm{freq}(\langle u, r, o\rangle, L)$, where the relative frequency of a user-permission tuple in a log is given by the *frequency function* $\mathrm{freq}(\langle u, r, o\rangle, L) = |\{e \in L \mid \mathrm{userPerm}(e) = \langle u, r, o\rangle\}| \, / \, |L|$, where the user-permission part of a log entry is given by $\mathrm{userPerm}(\langle u, r, o, t\rangle) = \langle u, r, o\rangle$.

For simplicity, our presentation of the problem and algorithm assume that attribute data does not change during the time covered by the log. Accommodating changes to attribute data is not difficult. It mainly requires re-defining the notions of policy quality and rule quality (introduced in Section 6.2) to be based on the set of log entries covered by a rule, denoted $[\![\rho]\!]_{\mathrm{LE}}$, rather than $[\![\rho]\!]$. The definition of $[\![\rho]\!]_{\mathrm{LE}}$ is similar to the definition of $[\![\rho]\!]$, except that, when determining whether a log entry is in $[\![\rho]\!]_{\mathrm{LE}}$, the attribute data in effect at the time of the log entry is used.

6.2 Algorithm

Our algorithm is based on the algorithm for mining ABAC policies from ACLs and attribute data in [XS13a]. Our algorithm does not take the order of log entries into account, so the log can be summarized by the user-permission relation UP_0 induced by the log and the frequency function freq, described in the penultimate paragraph of Section 3.2.

Top-level pseudocode appears in Figure 6.1. We refer to tuples selected in the first statement of the first while loop as *seeds*. The top-level pseudocode is explained by embedded comments. It calls several functions, described next. Function names hyperlink to pseudocode for the function, if it is included in the paper, otherwise to the description of the function.

The function $\mathrm{addCandidateRule}(s_{\mathrm{u}}, s_r, s_{\mathrm{o}}, cc, \mathrm{uncovUP}, \mathit{Rules})$ in Figure 6.2 first calls

computeUAE to compute a user-attribute expression e_{u} that characterizes s_{u}, and computeRAE to compute a resource-attribute expression e_{r} that characterizes s_r. It then calls generalizeRule(ρ, cc, uncovUP, *Rules*) to generalize rule $\rho = \langle e_{\mathrm{u}}, e_{\mathrm{r}}, s_{\mathrm{o}}, \emptyset\rangle$ to ρ' and adds ρ' to candidate rule set *Rules*. The details of the functions called by addCandidateRule are described next.

The function computeUAE(s, U) computes a user-attribute expression e_{u} that characterizes the set s of users. Preference is given to attribute expressions that do not use uid, since attribute-based policies are generally preferable to identity-based policies, even when they have higher WSC, because attribute-based generalize better. Similarly, computeRAE(s, R) computes a resource-attribute expression that characterizes the set s of resources. Pseudocode for computeUAE and computeRAE are omitted. The function candidateConstraint(r, u) returns a set containing all of the atomic constraints that hold between resource r and user u. Pseudocode for candidateConstraint is straightforward and omitted.

The function generalizeRule(ρ, cc, uncovUP, *Rules*) in Figure 6.3 attempts to generalize rule ρ by adding some of the atomic constraints in cc to ρ and eliminating the conjuncts of the user attribute expression and/or the resource attribute expression corresponding to the attributes used in those constraints, i.e., mapping those attributes to $\top$. We call a rule obtained in this way a *generalization* of ρ. Such a rule is more general than ρ in the sense that it refers to relationships instead of specific values. Also, the user-permission relation induced by a generalization of ρ is a superset of the user-permission relation induced by ρ. generalizeRule(ρ, cc, uncovUP, *Rules*) returns the generalization ρ' of ρ with the best quality according to a given rule quality metric. Note that ρ' may cover tuples that are already covered (i.e., are in UP); in other words, our algorithm can generate policies containing rules whose meanings overlap.

A *rule quality metric* is a function $Q_{\mathrm{rul}}(\rho, UP)$ that maps a rule ρ to a totally-ordered set, with the ordering chosen so that larger values indicate high quality. The second argument UP is a set of user-permission tuples. Our rule quality metric assigns higher quality to rules that cover more currently uncovered user-permission tuples and have smaller size, with an additional term that imposes a penalty for over-assignments, measured as a fraction of the number of user-permission tuples covered by the rule, and with a weight specified by a parameter w'_o, called the *rule over-assignment weight.*

$$Q_{\mathrm{rul}}(\rho, UP) = \frac{|[\![\rho]\!] \cap UP|}{|\rho|} \times (1 - \frac{w'_o \times |[\![\rho]\!] \setminus UP(L)|}{|[\![\rho]\!]|}).$$

In generalizeRule, uncovUP is the second argument to Q_{rul}, so $[\![\rho]\!] \cap UP$ is the set of user-permission tuples in UP_0 that are covered by ρ and not covered by rules already in the policy. The loop over i near the end of the pseudocode for generalizeRule considers all possibilities for the first atomic constraint in cc that gets added to the constraint of ρ. The function calls itself recursively to determine the subsequent atomic constraints in c that get added to the constraint.

We also developed a frequency-sensitive variant of this rule quality metric. Let $Q^{\mathrm{freq}}_{\mathrm{rul}}$ denote the frequency-weighted variant of Q_{rul}, obtained by weighting each user-permission tuple by its relative frequency (i.e., fraction of occurrences) in the log, similar to the definition of λ-distance in [MPC12b]. Specifically, the definition of $Q^{\mathrm{freq}}_{\mathrm{rul}}$ is obtained from the definition

of Q_{rul} by replacing $|[\![\rho]\!] \cap UP|$ with $|[\![\rho]\!] \cap UP|_L$ (recall that $|\cdot|_L$ is defined in Section 3.2).

We also developed a rule quality metric $Q_{\text{rul}}^{\text{ILP}}$ based closely on the theory quality metric for inductive logic programming described in [Mug95]. Details of the definition appear in Appendix B.1.

The function mergeRules(*Rules*) in Figure 6.3 attempts to improve the quality of *Rules* by removing redundant rules and merging pairs of rules. A rule ρ in *Rules* is *redundant* if *Rules* contains another rule ρ' such that every user-permission tuple in UP_0 that is in $[\![\rho]\!]$ is also in $[\![\rho']\!]$. Informally, rules ρ_1 and ρ_2 are merged by taking, for each attribute, the union of the conjuncts in ρ_1 and ρ_2 for that attribute. If adding the resulting rule ρ_{merge} to the policy and removing rules (including ρ_1 and ρ_2) that become redundant improves policy quality and ρ_{merge} does not have over-assignments, then ρ_{merge} is added to *Rules*, and the redundant rules are removed from *Rules*. As optimizations (in the implementation, not reflected in the pseudocode), meanings of rules are cached, and policy quality is computed incrementally. mergeRules(*Rules*) updates its argument *Rules* in place, and it returns a Boolean indicating whether any rules were merged.

The function simplifyRules(*Rules*) attempts to simplify all of the rules in *Rules*. It updates its argument *Rules* in place, replacing rules in *Rules* with simplified versions when simplification succeeds. It returns a Boolean indicating whether any rules were simplified. It attempts to simplify each rule in several ways, including elimination of subsumed sets in conjuncts for multi-valued attributes, elimination of conjuncts, elimination of constraints, elimination of elements of sets in conjuncts for multi-valued user attributes, and elimination of overlap between rules. The detailed definition is similar to the one in [XS13a] and is omitted to save space.

6.2.1 Example

We illustrate the algorithm on a small fragment of our university case study (*cf.* Section 6.4.1). The fragment contains a single rule

$$\rho_0 = \langle \text{true}, type \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{crsTaught} \ni \text{crs}\rangle$$

and all of the attribute data from the full case study, except attribute data for gradebooks for courses other than cs601. We consider an operation log L containing three entries:

$$\{\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore}, \text{t}_1\rangle, \langle \text{csFac2}, \text{cs601gradebook}, \text{readScore}, \text{t}_2\rangle, \langle \text{csStu3}, \text{cs601gradebook}, \text{addScore}, \text{t}_3\rangle\}$$

User csFac2 is a faculty in the computer science department who is teaching cs601; attributes are position = faculty, dept = cs, and crsTaught = {cs601}. csStu3 is a CS student who is a TA of cs601; attributes are position = student, dept = cs, and crsTaught = {cs601}. cs601gradebook is a resource with attributes type = gradebook, dept = cs, and crs = cs601.

Our algorithm selects user-permission tuple $\langle \text{csFac2}, \text{cs601gradebook}, \text{addScore}\rangle$ as the first seed, and calls function candidateConstraint to compute the set of atomic constraints that hold between csFac2 and cs601gradebook; the result is $cc = \{\text{dept} = \text{dept}, \text{crsTaught} \ni \text{crs}\}$. addCandidateRule is called twice to compute candidate rules. The first call to addCandidateRule calls computeUAE to compute a UAE e_{u} that characterizes the set s_{u}

```
// Rules is the set of candidate rules
Rules = ∅
// uncovUP contains user-permission tuples
// in UP_0 that are not covered by Rules
uncovUP = UP_0.copy()
while ¬uncovUP.empty()
  // Select an uncovered tuple as a "seed".
  ⟨u, r, o⟩ = some tuple in uncovUP
  cc = candidateConstraint(r, u)
  // s_u contains users with permission ⟨r, o⟩
  // and that have the same candidate
  // constraint for r as u
  s_u = {u' ∈ U | ⟨u', r, o⟩ ∈ UP_0
           ∧ candidateConstraint(r, u') = cc}
  addCandidateRule(s_u, {r}, {o}, cc, uncovUP, Rules)
  // s_o is set of operations that u can apply to r
  s_o = {o' ∈ Op | ⟨u, r, o'⟩ ∈ UP_0}
  addCandidateRule({u}, {r}, s_o, cc, uncovUP, Rules)
end while
// Repeatedly merge and simplify
// rules, until this has no effect
mergeRules(Rules)
while simplifyRules(Rules)
        && mergeRules(Rules)
  skip
end while
// Select high quality rules into Rules'.
Rules' = ∅
Repeatedly move highest-quality rule
from Rules to Rules' until
Σ_{ρ∈Rules'} [[ρ]] ⊇ UP_0, using
UP_0 \ [[Rules']] as second argument to
Q_rul, and discarding a rule if it does
not cover any tuples in UP_0 currently
uncovered by Rules'.
return Rules'
```

Figure 6.1: Policy mining algorithm. The pseudocode starts in column 1 and continues in column 2.

```
function addCandidateRule(s_u, s_r, s_o, cc, uncovUP, Rules)
// Construct a rule ρ that covers user-perm. tuples {⟨u, r, o⟩ | u ∈ s_u ∧ r ∈ s_r ∧ o ∈ s_o}.
e_u = computeUAE(s_u, U);  e_r = computeRAE(s_r, R);  ρ = ⟨e_u, e_r, s_o, ∅⟩
ρ' = generalizeRule(ρ, cc, uncovUP, Rules);  Rules.add(ρ');  uncovUP.removeAll([[ρ']])
```

Figure 6.2: Compute a candidate rule ρ' and add ρ' to candidate rule set *Rules*

containing users with permission $\langle \text{addScore}, \text{cs601gradebook} \rangle$ and with the same candidate constraint as csFac2 for cs601gradebook; the result is $e_\text{u} = (\text{position} \in \{\text{faculty}, \text{student}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{crsTaught} \supseteq \{\{\text{cs601}\}\})$. addCandidateRule also calls computeRAE to compute a resource-attribute expression that characterizes $\{\text{cs601gradebook}\}$; the result is $e_\text{r} = (\text{crs} \in \{\text{cs601}\} \wedge \text{dept} \in \{\text{cs}\} \wedge \text{type} \in \{\text{gradebook}\})$. The set of operations considered in this call to addCandidateRule is simply $s_\text{o} = \{\text{addScore}\}$. addCandidateRule then calls generalizeRule, which generates a candidate rule ρ_1 which initially has e_u, e_r and s_o in the first three components, and then atomic constraints in cc are added to ρ_1, and conjuncts in e_u and e_r for attributes used in cc are eliminated; the result is $\rho_1 = \langle \text{position} \in \{\text{faculty}, \text{student}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$, which also covers the third log entry. Similarly, the second call to addCandidateRule generates a candidate rule $\rho_2 = \langle \text{position} \in \{\text{faculty}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore}, \text{readScore}\}, \text{dept} = \text{dept} \wedge \text{crsTaught} \ni \text{crs} \rangle$, which also covers the second log entry.

All of $UP(L)$ is covered, so our algorithm calls mergeRules, which attempts to merge ρ_1 and ρ_2 into rule $\rho_3 = \langle \text{position} \in \{\text{faculty}, \text{student}\}, \text{type} \in \{\text{gradebook}\}, \{\text{addScore},$

```
function generalizeRule(ρ, cc, uncovUP,
                               Rules)
// ρ_best is best generalization of ρ
ρ_best = ρ
// gen[i][j] is a generalization of ρ using
// cc′[i]
gen = new Rule[cc.length][3]
for i = 1 to cc.length
  f = cc[i]
  // generalize by adding f and eliminating
  // conjuncts for both attributes used in f.
  gen[i][1] = ⟨uae(ρ)[uAttr(f) ↦ ⊤],
               rae(ρ)[rAttr(f) ↦ ⊤],
               ops(ρ), con(ρ) ∪ {f}⟩
  // generalize by adding f and eliminating
  // conjunct for user attribute used in f
  gen[i][2] = ⟨uae(ρ)[uAttr(f) ↦ ⊤], rae(ρ),
               ops(ρ), con(ρ) ∪ {f}⟩
  // generalize by adding f and eliminating
  // conjunct for resource attrib. used in f.
  gen[i][3] = ⟨uae(ρ), rae(ρ)[rAttr(f) ↦ ⊤],
               ops(ρ), con(ρ) ∪ {f}⟩
end for
for i = 1 to cc.length and j = 1 to 3
  // try to further generalize gen[i]
  ρ″ = generalizeRule(gen[i][j], cc[i+1 ..],
                       uncovUP, Rules)
  if Q_rul(ρ″, uncovUP) > Q_rul(ρ_best,
                                   uncovUP)
    ρ_best = ρ″
  end if
end for
return ρ_best
```

```
function mergeRules(Rules)
// Remove redundant rules
redun = {ρ ∈ Rules | ∃ ρ′ ∈ Rules \ {ρ}.
              [[ρ]] ∩ UP_0 ⊆ [[ρ′]] ∩ UP_0}
Rules.removeAll(redun)
// Merge rules
workSet = {(ρ_1, ρ_2) | ρ_1 ∈ Rules ∧ ρ_2 ∈ Rules
              ∧ ρ_1 ≠ ρ_2 ∧ con(ρ_1) = con(ρ_2)}
while not(workSet.empty())
  (ρ_1, ρ_2) = workSet.remove()
  ρ_merge = ⟨uae(ρ_1) ∪ uae(ρ_2),
             rae(ρ_1) ∪ rae(ρ_2),
             ops(ρ_1) ∪ ops(ρ_2), con(ρ_1)⟩
  // Find rules that become redundant
  // if merged rule ρ_merge is added
  redun = {ρ ∈ Rules | [[ρ]] ⊆ [[ρ_merge]]}
  // Add the merged rule and remove redun-
  // dant rules if this improves policy quality
  // and the merged rule does not have
  // over-assignments.
  if (Q_pol(Rules ∪ {ρ_merge} \ redun) < Q_pol(Rules)
      ∧ [[ρ_merge]] ⊆ UP_0)
    Rules.removeAll(redun)
    workSet.removeAll({(ρ_1, ρ_2) ∈ workSet |
        ρ_1 ∈ redun ∨ ρ_2 ∈ redun})
    workSet.addAll({(ρ_merge, ρ) | ρ ∈ Rules
                 ∧ con(ρ) = con(ρ_merge)})
    Rules.add(ρ_merge)
  end if
end while
return true if any rules were merged
```

Figure 6.3: Left: Generalize rule ρ by adding some formulas from cc to its constraint and eliminating conjuncts for attributes used in those formulas. $f[x \mapsto y]$ denotes a copy of function f modified so that $f(x) = y$. $a[i..]$ denotes the suffix of array a starting at index i. Right: Merge pairs of rules in *Rules*, when possible, to reduce the WSC of *Rules*. (a, b) denotes an unordered pair with components a and b. The union $e = e_1 \cup e_2$ of attribute expressions e_1 and e_2 over the same set A of attributes is defined by: for all attributes a in A, if $e_1(a) = \top$ or $e_2(a) = \top$ then $e(a) = \top$ otherwise $e(a) = e_1(a) \cup e_2(a)$.

readScore}, dept = dept $\wedge$ crsTaught $\ni$ crs$\rangle$. ρ_3 is discarded because it introduces an over-assignment while ρ_1 and ρ_2 do not. Next, simplifyRules is called, which first simplifies ρ_1 and ρ_2 to ρ'_1 and ρ'_2, respectively, and then eliminates ρ'_1 because it covers a subset of the tuples covered by ρ'_2. The final result is ρ'_2, which is identical to the rule ρ_0 in the original policy.

6.3 Algorithm Based on Generative Model

Another approach to this problem is to apply a machine learning algorithm that uses a statistical approach, based upon a generative model, to find the policy that is most likely to generate the behavior (usage of permissions) observed in the logs. This approach is inspired by Molloy *et al.*'s work on mining RBAC policies and simple ABAC policies from logs [MPC12a], using Rosen-Zvi et al.'s algorithm for learning author-topic models [RZCG+10].

The author-topic model (ATM) is a probabilistic generative model for collections of discrete data, such as documents [RZCG+10]. ATM assumes the following process to generate a document d: for each word w in d, an author a is randomly chosen, and then a topic t is chosen from a multinomial distribution over topics specific to the author a, and then the word w is chosen from a multinomial distribution over words specific to the selected topic t. The inputs to the ATM algorithm are a set of authors, a set of documents, a function giving the set of authors of each document, and the number of topics. The ATM algorithm finds a probability distribution relating authors to topics (i.e., how likely each author is to write about each topic) and a probability distribution relating topics to words (i.e., how likely each word is to be used in text on each topic), such that the process described above, with these probability distributions, is likely to generate the given documents.

Molloy *et al.* use ATM to mine meaningful roles from logs and user attribute data. They employ the following correspondence [MPC12a]: authors correspond to a restricted form of user attribute expressions (specifically, user attribute expressions with at most three conjuncts of the form "*attribute=value*"), topics correspond to roles, words correspond to permissions, and documents correspond to users (i.e., for each user u, there is a document containing the permissions from the log entries for user u). For the document corresponding to user u, the set of authors is the set of user attribute expressions (UAEs) satisfied by u. With this correspondence, the ATM finds a probability distribution ϕ_1 relating UAEs to roles (i.e., $\phi_1(e, r)$ is the probability that users satisfying UAE e are members of role r) and a probability distribution ϕ_2 relating roles to permissions (i.e., $\phi_2(r, p)$ is the probability that role r has permission p). For each user u, the probability distributions $\phi_1(e, r)$ for each UAE e satisfied by u are averaged. Together, the results form a probability distribution $\hat{\phi}_1$ relating users to roles (i.e., $\hat{\phi}_1(u, r)$ is the probability that user u is a member of role r). Finally, the probability distributions $\hat{\phi}_1$ and ϕ_2 are discretized to obtain a user-role assignment and a role-permission assignment.

To adapt this approach to mine ABAC policies from logs and attribute data, a new correspondence between the components of the author-topic model and components of the access control model is needed, to accommodate the following differences between our problem and theirs: the goal is to generate an ABAC policy, not a meaningful RBAC policy; resource attribute data, as well as user attribute data, is available; attributes can be multi-valued, and

set relations can be used in attribute expressions; and constraints relating user attributes and resource attributes are allowed.

We propose the following correspondence. The basic idea is to modify the correspondence in [MPC12a] so that resources are treated the same way as users, because in our framework, users and resources both have attributes. Another reason that users and resources need to be treated together is the presence of constraints: users and resources must be associated with rules in a coordinated way, not independently. Thus, resources are tupled with users, instead of being tupled with operations to form permissions. Authors correspond to tuples $\langle \mathrm{uae}, \mathrm{rae}, \mathrm{con} \rangle$, where uae is a user attribute expression with at most b_u conjuncts, rae is a resource attribute expression with at most b_r conjuncts, and con is a constraint with at most b_c atomic constraints. Furthermore, (1) each conjunct in uae or rae may contain only one value, specifically, one atomic value or one set of size at most b_s, depending on whether the attribute is single-valued or multi-valued, and (2) each user attribute and each resource attribute appear in at most one atomic constraints in con. Disjunction (i.e., conjuncts with multiple values) is introduced later, by "merging" sets of rules, as described below. In experiments, we take $b_u = 2$, $b_r = 2$, $b_c = 2$, and $b_s = 1$; these are the smallest values sufficient to express our case study policies. Topics correspond to rules. Words correspond to operations. Documents correspond to user-resource pairs; the document corresponding to a user-resource pair $\langle u, r \rangle$ is the sequence of operations performed (according to the log) by user u on resource r. For the document corresponding to $\langle u, r \rangle$, the set of authors is the set of $\langle \mathrm{uae}, \mathrm{rae}, \mathrm{con} \rangle$ tuples such that $u \models \mathrm{uae} \wedge r \models \mathrm{rae} \wedge u, r \models \mathrm{con}$.

The results of the ATM learning algorithm are a set of k topics, and two families of probability distributions θ and ϕ. For each author a, θ_a is a probability distribution over topics, such that $\theta_{a,t}$ is the probability that author a will select to write about topic t. For each topic t, ϕ_t is a probability distribution over words, such that $\phi_{t,w}$ is the probability that an author writing on topic t will use word w. A discretization algorithm, taken from Algorithm 1 in [MPC12a], is used to obtain an ABAC policy from the topics and probability distributions. The details of discretization algorithm, written in our notation, appear in Figure 6.4. The inputs are: the input I to the policy mining problem defined in Section 3.2, the families of probability distributions θ and ϕ described above, the maximum number of iterations *maxIter*, the author set A, the word set W, the number of topics k. The integer array AT used in the algorithm defines the discretized author-topic assignment; specifically, author a is associated with the $AT[a]$ topics that have the largest probability according to θ_a. Similarly, the array TW defines the discretized topic-word assignment; specifically, topic t is is associated with the $TW[t]$ words that have the largest probability according to ϕ_t. A set of rules is constructed from AT and TW by the constructABACRules function defined in Figure 6.5. For each topic t, and for each author $\langle \mathrm{uae}, \mathrm{rae}, \mathrm{con} \rangle$ that is associated with t by AT, the rule $\langle \mathrm{uae}, \mathrm{rae}, O, \mathrm{con} \rangle$ is included in the rule set, where O is the set of words that are associated with t by TW.

To reduce the policy's WSC and improve its quality, constructABACRules invokes a variant of the mergeRules function defined in Figure 5.2. This variant, called mergeRulesGen, is obtained from mergeRules by replacing the validity test $[\![\rho_{\mathrm{merge}}]\!] \subseteq UP_0$ (which is inappropriate here because over-assignments are allowed) with a test that merging does not change the meaning of the policy, namely, $[\![\rho_{\mathrm{merge}}]\!] \subseteq [\![Rules]\!]$.

The discretization algorithm aims to find values for AT and TW that maximize a policy

quality metric. We use the policy quality metric defined in equation 6.1 in Section 3.2. The algorithm is based on annealing. The algorithm begins with a random initialization of AT and TW, and iteratively updates them until the updates converge or the maximum number of iteration is reached. In each iteration, the algorithm first tries to update each $AT[a]$ with a new value that differs from the current value by at most ϵ, where ϵ is a parameter of the annealing process. If the updated value results in an ABAC policy with a higher quality, then the new value is accepted (by storing it in $AT_{tmp}[a]$ and later assigning it to $AT[a]$). Otherwise, the new value is accepted with a probability drawn from the probability distribution Pr[exp((Q(π, L) - Q(π', L)))/T)/Z], where Z is a normalization factor, and T is the temperature. The initial temperature T_0 is relatively high, and the temperature is decreased with rate γ in each iteration, where T_0 and γ are parameters to the annealing process. TW is updated similarly. All accepted updates are applied to AT and TW simultaneously at the end of each iteration, by the assignment $AT = AT_{tmp}$ and a similar assignment for TW. This algorithm is greedy in the sense that it considers only associations in which some number of the highest-ranked topics or words are associated with an author or topic, respectively; consequently, it might be unable to produce an optimal discretization in some cases.

Recall that the ATM algorithm takes the number k of topics (i.e., the desired number of rules in the policy) as an input. In general, k is not known in advance. A simple approach to determine a reasonable value of k is to start with a low estimate and iteratively increase it, re-running the entire algorithm each time, until the improvement in policy quality falls below a threshold. Developing an incremental version of the ATM algorithm would be difficult, so the policy mining algorithm would executed from scratch each time, making this approach inefficient. It might be possible to develop a more efficient search strategy for finding the smallest value of k that, when incremented, produces a below-threshold improvement in policy quality.

6.4 Evaluation Methodology

We evaluate our policy mining algorithms on synthetic operation logs generated from ABAC policies (some handwritten and some synthetic) and probability distributions characterizing the frequency of actions. This allows us to evaluate the effectiveness of our algorithm by comparing the mined policies with the original ABAC policies. We are eager to also evaluate our algorithm on actual operation logs and actual attribute data, when we are able to obtain them.

6.4.1 ABAC Policies

Case Studies We developed four case studies for use in evaluation of our algorithm, described briefly here. Details of the case studies, including all policy rules, various size metrics (number of users, number of resources, etc.), and some illustrative attribute data, appear in [XS13a].

Our *university case study* is a policy that controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. Our *health care case study* is a

```
    function Discretization(I, θ, φ, maxIter, A, W, k)
1:  AT = new Vector(), TW = new Vector()
2:  for a = 1 to |A|
3:     AT[a] = randomInt(0, k)
4:  end for
5:  for t = 1 to k
6:     TW[t] = randomInt(0, |W|)
7:  end for
8:  Rules = constructABACRules(θ, φ, AT, TW, A)
9:  π = I[8 → Rules]
10: T = T_0
11: i = 0
12: while i ≤ maxIter
13:    AT_tmp = AT.copy()
14:    for a = 1 to |A|
15:       AT′ = AT.copy()
16:       AT′[a] = randomInt(max(0, AT[a] − ε), min(k, AT[a] + ε))
17:       Rules′ = constructABACRules(θ, φ, AT′, TW, A)
18:       π′ = I[8 → Rules′]
19:       if Q(π′, π_in.L) < Q(π, π_in.L)
20:          AT_tmp[a] = AT′[a]
21:       else
22:          if random(0, 1) ≤ Pr[exp((Q(π′, L) − Q(π, L)))/T)/Z]
23:             AT_tmp[a] = AT′[a]
24:          end if
25:       end if
26:    end for
27:    /* Perform a similar updating process for TW */
28:    Rules = constructABACRules(θ, φ, AT_tmp, TW_tmp, A)
29:    π = I[8 → Rules]
30:    if AT = AT_tmp and TW = TW_tmp
31:       break
32:    end if
33:    AT = AT_tmp; TW = TW_tmp
34:    T = γT
35:    i = i + 1
36: end while
37: return π
```

Figure 6.4: Discretization algorithm. random(x, y) returns a random number in the range $[x, y)$. randomInt(i, j) returns a random integer in the range $[i, j]$. $t[i \to v]$ denotes the tuple obtained from tuple t by changing the value of the i'th component to v.

```
    function constructABACRules(θ, φ, AT, TW, A)
1:  Rules = new Set()
2:  for a in A
3:      T = the set containing the topics corresponding to
4:              the AT[a] largest values of θ_a
5:      for t in T
6:          O = the set containing the words corresponding to
7:                  the TW[t] largest values of φ_t
8:          ⟨uae, rae, con⟩ = a
9:          ρ = ⟨uae, rae, O, con⟩
10:         Rules.add(ρ)
11:     end for
12: end for
13: mergeRulesGen(Rules)
14: return Rules
```

Figure 6.5: Construct an ABAC policy based on an author-topic assignment and a topic-word assignment.

policy that controls access by nurses, doctors, patients, and agents (e.g., a patient's spouse) to electronic health records (HRs) and HR items (i.e., entries in health records). Our *project management case study* is a policy that controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. Our *online video case study* is a policy that controls access to videos by users of an online video service.

The number of rules in the case studies is relatively small (10 ± 1 for the first three case studies, and 6 for online video), but they express non-trivial policies and exercise all the features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The manually written attribute dataset for each case study contains a small number of instances of each type of user and resource.

For the first three case studies, we generated a series of synthetic attribute datasets, parameterized by a number N, which is the number of departments for the university and project management case studies, and the number of wards for the health care case study. The generated attribute data for users and resources associated with each department or ward are similar to but more numerous than the attribute data in the manually written datasets. We did not bother creating synthetic data for the online video case study, because the rules are simpler.

Synthetic Policies We generated synthetic policies using the algorithm proposed by Xu and Stoller [XS13a]. Briefly, the policy synthesis algorithm first generates the rules and then uses the rules to guide generation of the attribute data; this allows control of the number of granted permissions. The algorithm takes N_{rule}, the desired number of rules, as an input. The numbers of users and resources are proportional to the number of rules. Generation of rules and attribute data is based on several probability distributions, which are based loosely

on the case studies or assumed to have a simple functional form (e.g., uniform distribution).

6.4.2 Log Generation

The inputs to the algorithm are an ABAC policy π, the desired completeness of the log, and several probability distributions. The *completeness* of a log, relative to an ABAC policy, is the fraction of user-permission tuples in the meaning of the policy that appear in at least one entry in the log. A straightforward log generation algorithm would generate each log entry by first selecting an ABAC rule, according to a probability distribution on rules, and then selecting a user-permission tuple that satisfies the rule, according to probability distributions on users, resources, and operations. This process would be repeated until the specified completeness is reached. This algorithm is inefficient when high completeness is desired. Therefore, we adopt a different approach that takes advantage of the fact that our policy mining algorithm is insensitive to the order of log entries and depends only on the frequency of each user-permission tuple in the log. In particular, instead of generating logs (which would contain many entries for popular user-permission tuples), our algorithm directly generates a *log summary*, which is a set of user-permission tuples with associated frequencies (equivalently, a set of user-permission tuples and a frequency function).

Probability Distributions An important characteristic of the probability distributions used in synthetic log and log summary generation is the ratio between the most frequent (i.e., most likely) and least frequent items of each type (rule, user, etc.). For case studies with manually written attribute data, we manually created probability distributions in which this ratio ranges from about 3 to 6. For case studies with synthetic data and synthetic policies, we generated probability distributions in which this ratio is 25 for rules, 25 for resources, 3 for users, and 3 for operations (the ratio for operations has little impact, because it is relevant only when multiple operations appear in the same rule, which is uncommon).

6.4.3 Metrics

For each case study and each associated attribute dataset (manually written or synthetic), we generate a synthetic operation log using the algorithm in Section 6.4.2 and then run our ABAC policy mining algorithms. We evaluate the effectiveness of each algorithm by comparing the generated ABAC policy to the original ABAC policy, using the metrics described below.

Syntactic Similarity Jaccard similarity of sets is $J(S_1, S_2) = |S_1 \cap S_2| \,/\, |S_1 \cup S_2|$. Syntactic similarity of UAEs is defined by $s_{\mathrm{uae}}(e, e') = |A_{\mathrm{u}}|^{-1} \sum_{a \in A_{\mathrm{u}}} J(e(a), e'(a))$. Syntactic similarity of RAEs is defined by $s_{\mathrm{rae}}(e, e') = |A_{\mathrm{r}}|^{-1} \sum_{a \in A_{\mathrm{r}}} J(e(a), e'(a))$. The syntactic similarity of rules $\langle e_{\mathrm{u}}, e_{\mathrm{r}}, O, c\rangle$ and $\langle e'_{\mathrm{u}}, e'_{\mathrm{r}}, O', c'\rangle$ is the average of the similarities of their components, specifically, the average of $s_{\mathrm{uae}}(e_{\mathrm{u}}, e'_{\mathrm{u}})$, $s_{\mathrm{rae}}(e_{\mathrm{r}}, e'_{\mathrm{r}})$, $J(O, O')$, and $J(c, c')$. The *syntactic similarity* of rule sets *Rules* and *Rules*$'$ is the average, over rules ρ in *Rules*, of the syntactic similarity between ρ and the most similar rule in *Rules*$'$. The *syntactic similarity* of policies π and π' is the maximum of the syntactic similarities of the sets of rules in the policies,

considered in both orders (this makes the relation symmetric). Syntactic similarity ranges from 0 (completely different) to 1 (identical).

Semantic Similarity Semantic similarity measures the similarity of the entitlements granted by two policies. The *semantic similarity* of policies π and π' is defined by $J(\llbracket\pi\rrbracket, \llbracket\pi'\rrbracket)$. Semantic similarity ranges from 0 (completely different) to 1 (identical).

Fractions of Under-Assignments and Over-Assignments To characterize the semantic differences between an original ABAC policy π_0 and a mined policy π in a way that distinguishes under-assignments and over-assignments, we compute the fraction of over-assignments and the fraction of under-assignments, defined by $|\llbracket\pi\rrbracket \setminus \llbracket\pi_0\rrbracket| \,/\, |\llbracket\pi\rrbracket|$ and $|\llbracket\pi_0\rrbracket \setminus \llbracket\pi\rrbracket| \,/\, |\llbracket\pi\rrbracket|$, respectively.

6.5 Experimental Results

This section presents experimental results using an implementation of our algorithm in Java. The implementation, case studies, and synthetic policies used in the experiments are available at `http://www.cs.stonybrook.edu/~stoller/`.

Over-Assignment Weight The optimal choice for the over-assignment weights w_o and w'_o in the policy quality and rule quality metrics, respectively, depends on the log completeness. When log completeness is higher, fewer over-assignments are desired, and larger over-assignments weights give better results. In experiments, we take $w_o = 50c - 15$ and $w'_o = w_o/10$, where c is log completeness. In a production setting, the exact log completeness would be unknown, but a rough estimate suffices, because our algorithm's results are robust to error in this estimate. For example, for case studies with manually written attribute data, when the actual log completeness is 80%, and the estimated completeness used to compute w_o varies from 70% to 90%, the semantic similarity of the original and mined policies varies by 0.04, 0.02, and 0 for university, healthcare, and project management, respectively.

Experimental Results Figure 6.6 shows results from our algorithm. In each graph, curves are shown for the university, healthcare, and project management case studies with synthetic attribute data with N equal to 6, 10, and 10, respectively (average over results for 10 synthetic datasets, with 1 synthetic log per synthetic dataset), the online video case study with manually written attribute data (average over results for 10 synthetic logs), and synthetic policies with $N_{\text{rule}} = 20$ (average over results for 10 synthetic policies, with 1 synthetic log per policy). Error bars show standard deviation. Running time is at most 12 sec for each problem instance in our experiments.

For log completeness 100%, all four case study policies are reconstructed exactly, and the semantics of synthetic policies is reconstructed almost exactly: the semantic similarity is 0.99. This is a non-trivial result, especially for the case studies: an algorithm could easily generate a policy with over-assignments or more complex rules. As expected, the results get worse as log completeness decreases. When evaluating the results, it is important to consider

what levels of log completeness are likely to be encountered in practice. One datapoint comes from Molloy *et al.*'s work on role mining from real logs [MPC12b]. For the experiments in [MPC12b, Tables 4 and 6], the actual policy is not known, but their algorithm produces policies with 0.52% or fewer over-assignments relative to $UP(L)$, and they interpret this as a good result, suggesting that they consider the log completeness to be near 99%. Based on this, we consider our experiments with log completeness below 90% to be severe stress tests, and results for log completeness 90% and higher to be more representative of typical results in practice.

Syntactic similarity for all four case studies is above 0.91 for log completeness 60% or higher, and is above 0.94 for log completeness 70% or higher. Syntactic similarity is lower for synthetic policies, but this is actually a good result. The synthetic policies tend to be unnecessarily complicated, and the mined policies are better in the sense that they have lower WSC. For example, for 100% log completeness, the mined policies have 0.99 semantic similarity to the synthetic policies (i.e., the meaning is almost the same), but the mined policies are simpler, with WSC 17% less than the original synthetic policies.

Semantic similarity is above 0.85 for log completeness 60% or higher, and above 0.94 for log completeness 80% or higher. These results are quite good, in the sense that our algorithm compensates for most of the log incompleteness. For example, at log completeness 0.6, for policies generated by a policy mining algorithm that produces policies granting exactly the entitlements reflected in the log, the semantic similarity would be 0.6. With our algorithm, the semantic similarity, averaged over the 5 examples, is 0.95. Thus, in this case, our algorithm compensates for $35/40 = 87.5\%$ of the incompleteness.

The fractions of over-assignments are below 0.03 for log completeness 60% or higher. The fractions of under-assignments are below 0.05 for log completeness 60% or higher for the case studies and are below 0.05 for log completeness 80% or higher for synthetic policies. The graphs also show that the semantic differences are due more to under-assignments than over-assignments; this is desirable from a security perspective.

Comparison of Rule Quality Metrics The above experiments use the first rule quality metric, Q_{rul}, in Section 6.2. We also performed experiments using $Q_{\mathrm{rul}}^{\mathrm{freq}}$ and $Q_{\mathrm{rul}}^{\mathrm{ILP}}$ on case studies with manually written attribute data and synthetic policies. Q_{rul} is moderately better overall than $Q_{\mathrm{rul}}^{\mathrm{freq}}$ and significantly better overall than $Q_{\mathrm{rul}}^{\mathrm{ILP}}$.

Comparison with Inductive Logic Programming To translate ABAC policy mining from logs to Progol [MF01], we used the translation of ABAC policy mining from ACLs to Progol in [XS13a, Sections 5.5, 16], except negative examples corresponding to absent user-permission tuples are omitted from the generated program, and the statement `set(posonly)?` is included, telling Progol to use its algorithm for learning from positive examples. For the four case studies with manually written attribute data (in contrast, Figure 6.6 uses synthetic attribute for three of the case studies), for log completeness 100%, semantic similarity of the original and Progol-mined policies ranges from 0.37 for project management and healthcare to 0.93 for online video, while our algorithm exactly reconstructs all four policies.

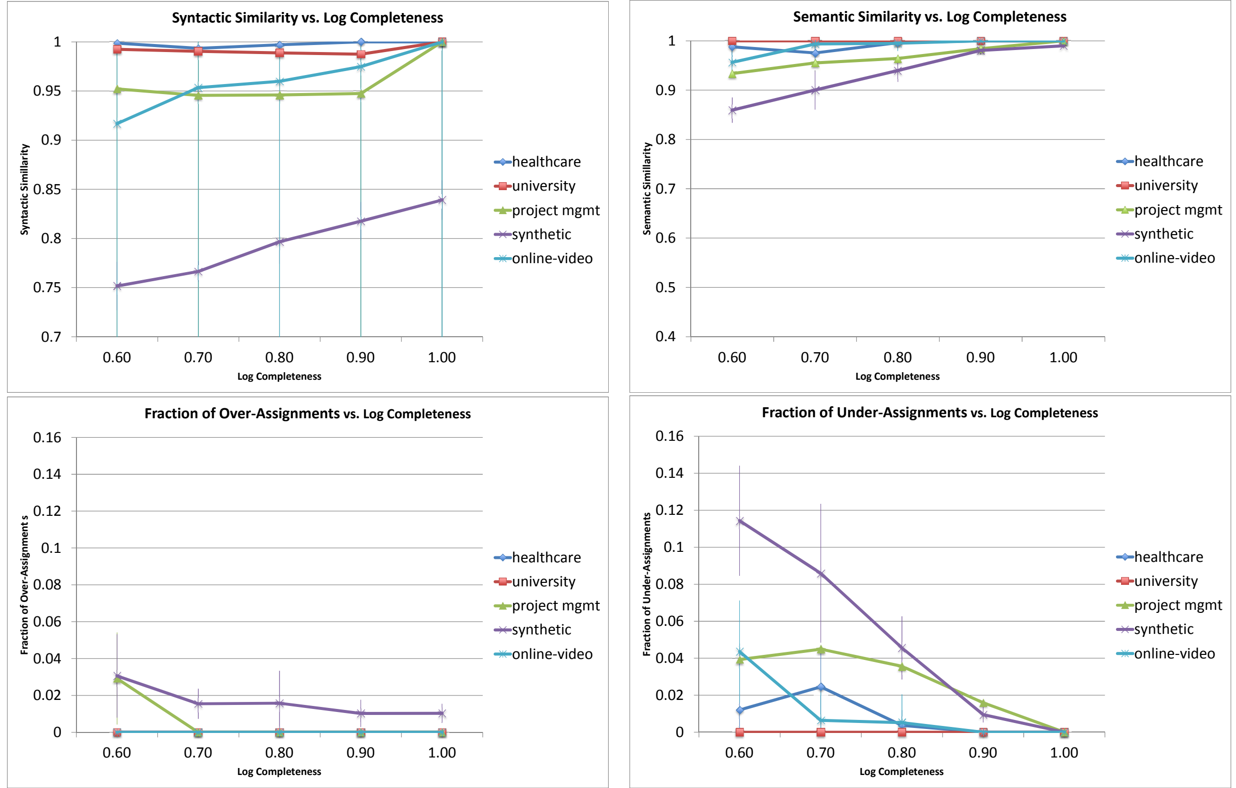

Figure 6.6: Top: Syntactic similarity and semantic similarity of original and mined ABAC policies, as a function of log completeness. Bottom: Fractions of over-assignments and under-assignments in mined ABAC policy, as a function of log completeness.

6.6 Related Work

We are not aware of prior work on ABAC mining from logs. We discuss prior work on related problems.

Our policy mining algorithm is based on our algorithm for ABAC policy mining from ACLs [XS13a]. The main differences are described in Section 4.

Association rule mining is another possible basis for ABAC policy mining. However, association rule mining algorithms are not well suited to ABAC policy mining, because they are designed to find rules that are probabilistic in nature and are supported by statistically strong evidence. They are not designed to produce a set of rules that completely cover the input data and are minimum-sized among such sets of rules. Consequently, unlike our algorithm, they do not give preference to smaller rules or rules with less overlap.

Ni *et al.* investigated the use of machine learning algorithms for security policy mining [NLC+09]. In the most closely related part of their work, a supervised machine learning algorithm is used to learn classifiers (analogous to attribute expressions) that associate users with roles, given as input the users, the roles, user attribute data, and the user-role assignment. Perhaps the largest difference between their work and ABAC policy mining is that their approach needs to be given the roles and the role-permission or user-role assignment as training data; in contrast, ABAC policy mining algorithms do not require any part of the

desired high-level policy to be given as input. Also, their work does not consider anything analogous to constraints.

Gal-Oz *et al.* [GOGY+11] mine roles from logs that record sets of permissions exercised together in one high-level operation. Their algorithm introduces roles whose sets of assigned permissions are the sets of permissions in the log. Their algorithm introduces over-assignments by removing roles with few users or whose permission set occurs few times in the log and re-assigning their members to roles with more permissions. Their algorithm does not use attribute data.

Molloy *et al.* apply a machine learning algorithm that uses a statistical approach, based upon a generative model, to find the RBAC policy that is most likely to generate the behavior (usage of permissions) observed in the logs [MPC12b]. They give an algorithm, based on Rosen-Zvi et al.'s algorithm for learning Author-Topic Models (ATMs), to mine meaningful roles from logs and attribute data, i.e., roles such that the user-role assignment is statistically correlated with user attributes. Their approach can be adapted to ABAC policy mining from logs, but its scalability in this context is questionable, because the adapted algorithm would enumerate and then rank all tuples containing a UAE, RAE and constraint (i.e., all tuples with the components of a candidate rule other than the operation set), and the number of such tuples is very large. In contrast, our algorithm never enumerates such candidates.

Zhang *et al.* use machine learning algorithms to improve the quality of a given role hierarchy based on users' access patterns as reflected in operation logs [ZGL+11, ZCG+13]. These papers do not consider improvement or mining of ABAC policies.

Chapter 7

Future Work

Access control is a cornerstone of computer security. Access control policies are critical to the security of many IT systems. Higher-level access control policy frameworks, such as RBAC and ABAC, promise long-term cost savings through reduced management effort, compared to lower-level policy frameworks such as ACLs, but manual development of an initial policy can be difficult [BM13] and expensive [HFK$^+$13]. Policy mining algorithms promise to drastically reduce the cost of migrating to a higher-level policy framework, by partially automating that process.

In this dissertation, we first developed RBAC policy mining algorithms that can easily be used to optimize a variety of policy quality metrics, including metrics based on policy size, metrics based on interpretability of the roles with respect to user attribute data, and compound metrics that consider size and interpretability. We then defined an expressive PRBAC framework, which supports a simple form of ABAC, and developed two algorithms for mining PRBAC policies from ACLs, user attribute data, and permission attribute data. Finally, we defined an expressive ABAC framework and developed three algorithms for mining ABAC policies, which differ in the source of user-permission data: ACLs, RBAC policies, and operation logs, respectively. To the best of our knowledge, these are the first policy mining algorithms for any PRBAC or ABAC frameworks.

The rest of this section describes some directions for future work on policy mining.

Mining RBAC Policies One direction for future work is to consider other metrics for policy interpretability, e.g., metrics that consider heterogeneity of users in different roles as well homogeneity of users in the same role [SR05]. Another direction is to improve scalability. One possible approach is divide-and-conquer, by decomposing a role mining problem into smaller sub-problems, solving them separately, and then combining the results. The main challenge is to develop a way to decompose the problem that minimizes the resulting loss in the quality of the final policy. This approach has been explored by Verde *et al.* [VVAC12]. While their method already gives good results, improvements might be possible by extending their method to take attribute information into account in all steps; their current method uses attribute information only in the first step.

Mining ABAC Policies from ACLs While we have developed the first ABAC policy mining algorithm, we regard this as opening, not closing, the doors for further work in this

area. There are many directions for future work.

One direction is to extend our policy language and policy mining algorithm to support additional forms of attribute expressions and constraints, including more flexible use of set relations, and additional data types and primitive relations on those data types, e.g., trees and the sub-tree relation, and sequences and the prefix relation.

Another direction is to extend the language and algorithm to support predicates on multiple attributes. Currently, each conjunct of an attribute expression involves at most one user attribute or resource attribute. Extending the language and algorithm to support predicates on multiple attributes would allow some policies to be expressed more concisely and will bring the ABAC policy language closer to Datalog. It might be possible to adapt techniques from inductive logic programming for this purpose.

Another useful extension to the language and algorithm would be support for negation in attribute expressions and constraints, and support for deny rules. Deny rules are used to override authorizations granted by permission rules. These features are similar, because they are both used to make policies more concise in cases where the complement of a set can be expressed more concisely than the set itself. They are supported in some widely deployed ABAC frameworks, notably XACML [XAC], so support for them is of practical significance.

Another direction for future work is to consider policy evolution. Given an ABAC policy and desired changes to the entitlements, the problem is to find a new ABAC policy with good quality according to a specified policy quality metric (e.g., based on policy size) and with minimum distance from (i.e., minimum differences from) the given policy according to a specified distance metric. Policy evolution has been considered in the context of RBAC [VAGA08, Lim10] but not ABAC. Our ABAC policy mining algorithm could be adapted to this problem by modifying the policy quality and rule quality metrics appropriately.

Context-dependent policies, such as temporal policies, spatial (a.k.a. location-aware) policies, and purpose-based policies, are increasingly important, in part due to the growing use of mobile computers (tablets, etc.). This suggests developing techniques for mining context-dependent ABAC policies from ACLs or logs. Mining of temporal roles from a user-permission matrix with temporal constraints is studied in [MSAV13]. It might be possible to adapt their approach to mining ABAC policies from ACLs with temporal constraints. Mining context-dependent ABAC policies from logs is more difficult, because the context constraints are not given explicitly; instead, they must be inferred from usage patterns in the logs. When mining spatial policies, an additional challenge is that locations may be hierarchically structured (e.g., rooms within suites within buildings within campuses).

Mining ABAC Policies from RBAC Policies One direction for future work is to modify the problem definition and algorithm to allow a loosening of the correspondence between roles and rules; this might be a desirable trade-off when it allows a significant improvement in the WSC of the policy. Other directions for future work include extending the policy language and policy mining algorithm to support the ABAC policy language features mentioned as extensions for mining ABAC from ACLs.

ABAC Policy Mining from Logs One direction for future work is to consider changes to attribute data. For example, the log might contain entries representing updates to attribute

data. This makes the analysis sensitive to the order of log entries; our current algorithm is insensitive to the order of log entries. Another direction is to explore policy quality metrics that explicitly assign higher quality to policies with rules that apply to groups of users with similar usage frequencies for the permissions granted by the rule. Our current algorithm does not do this explicitly, although it does this implicitly to some extent when using a rule quality metric that assigns higher quality to rules that grant more frequently used permissions; also, it may do this implicitly to some extent if users with similar usage frequencies for permissions have similar attributes.

Appendices

Appendix A

Supplemental Material on ABAC Mining from ACLs

A.1 Proof of NP-Hardness

This section shows that the ABAC policy mining problem is NP-hard, by reducing the Edge Role Mining Problem (Edge RMP) [LVA08] to it.

An *RBAC policy* is a tuple $\pi_{\rm RBAC} = \langle U, P, R, UA, PA \rangle$, where R is a set of roles, $UA \subseteq U \times R$ is the user-role assignment, and $PA \subseteq R \times P$ is role-permission assignment. The *number of edges* in an RBAC policy $\pi_{\rm RBAC}$ with this form is $|UA|+|PA|$. The user-permission assignment induced by an RBAC policy with the above form is $[\![\pi_{\rm RBAC}]\!] = UA \circ PA$, where $\circ$ denotes relational composition.

The *Edge Role Mining Problem (Edge RMP)* is [LVA08]: Given an ACL policy $\langle U, P, UP \rangle$, where U is a set of users, P is a set of permissions, and $UP \subseteq U \times P$ is a user-permission relation, find an RBAC policy $\pi_{\rm RBAC} = \langle U, P, R, UA, PA \rangle$ such that $[\![\pi_{\rm RBAC}]\!] = UP$ and $\pi_{\rm RBAC}$ has minimum number of edges among RBAC policies satisfying this condition. NP-hardness of Edge RMP follows from Theorem 1 in [MCL+10], since Edge RMP corresponds to the Weighted Structural Complexity Optimization (WSCO) Problem with $w_r = 0$, $w_u = 1$, $w_p = 1$, $w_h = \infty$, and $w_d = \infty$.

Given an Edge RMP problem instance $\langle U, P, UP \rangle$, consider the ABAC policy mining problem instance with ACL policy $\pi_0 = \langle U \cup \{u_0\}, P \cup \{r_0\}, \{op_0\}, UP_0 \rangle$, where u_0 is a new user and r_0 is a new resource, $UP_0 = \{\langle u, r, op_0 \rangle \mid \langle u, r \rangle \in UP\}$, user attributes $A_{\rm u} = \{{\rm uid}\}$, resource attributes $A_{\rm r} = \{{\rm rid}\}$, user attribute data $d_{\rm u}$ defined by $d_{\rm u}(u, {\rm uid}) = u$, resource attribute data $d_{\rm r}$ defined by $d_{\rm r}(r, {\rm rid}) = r$, and policy quality metric $Q_{\rm pol}$ defined by WSC with $w_1 = 1$, $w_2 = 1$, $w_3 = 0$, and $w_4 = 1$. Without loss of generality, we assume $U \cap P = \emptyset$; this should always hold, because in RBAC, users are identified by names that are atomic values, and permissions are resource-operation pairs; if for some reason this assumption doesn't hold, we can safely rename users or permissions to satisfy this assumption, because RBAC semantics is insensitive to equalities between users and permissions.

A solution to the given Edge-RMP problem instance can be constructed trivially from a solution $\pi_{\rm ABAC}$ to the above ABAC policy mining instance by interpreting each rule as a role. Note that rules in $\pi_{\rm ABAC}$ do not contain any constraints, because uid and rid are

the only attributes, and $U \cap P = \emptyset$ ensures that constraints relating uid and rid are useless (consequently, any non-zero value for w_4 suffices). The presence of the "dummy" user u_0 and "dummy" resource r_0 ensure that the UAE and RAE in every rule in π_{ABAC} contains a conjunct for uid or rid, respectively, because no correct rule can apply to all users or all resources. These observations, and the above choice of weights, implies that the WSC of a rule ρ in π_{RBAC} equals the number of users that satisfy ρ plus the number of resources (i.e., permissions) that satisfy ρ. Thus, WSC(π_{RBAC}) equals the number of edges in the corresponding RBAC policy, and an ABAC policy with minimum WSC corresponds to an RBAC policy with minimum number of edges.

A.2 Asymptotic Running Time

This section analyzes the asymptotic running time of our algorithm. We first analyze the main loop in Figure 5.1, i.e., the while loop in lines 3–11. First consider the cost of one iteration. The running time of candidateConstraint in line 5 is $O(|A_{\text{u}}| \times |A_{\text{r}}|)$. The running time of line 6 is $O(|U_{r,o}| \times |A_{\text{u}}| \times |A_{\text{r}}|)$, where $U_{r,o} = \{u' \in U \mid \langle u', r, o\rangle \in \text{uncovUP}\}$; this running time is achieved by incrementally maintaining an auxiliary map that maps each pair $\langle r, o\rangle$ in $R \times Op$ to $U_{r,o}$. The running time of function generalizeRule in line 4 in Figure 6.2 is $O(|2^{|cc|}|)$. Other steps in the main loop are either constant time or linear, i.e., $O(|A_{\text{u}}|+|A_{\text{r}}|+|UP_0|)$. Now consider the number of iterations of the main loop. The number of iterations is $|Rules_1|$, where $Rules_1$ is the set of rules generated by the main loop. In the worst case, the rule generated in each iteration covers one user-permission tuple, and $|Rules_1|$ is as large as $|UP_0|$. Typically, rules generalize to cover many user-permission tuples, and $|Rules_1|$ is much smaller than $|UP_0|$.

The running time of function mergeRules is $O(|Rules_1|^3)$. The running time of function simplifyRules is based on the running times of the five "elim" functions that it calls. Let $lc_{\text{u,m}}$ (mnemonic for "largest conjunct") denote the maximum number of sets in a conjunct for a multi-valued user attribute in the rules in $Rules_1$, i.e., $\forall a \in A_{\text{u,m}}.\, \forall \rho \in Rules_1.\, |\text{uae}(\rho)(a)| \leq lc_{\text{u,m}}$. The value of $lc_{\text{u,m}}$ is at most $|\text{Val}_{\text{m}}|$ but typically small (one or a few). The running time of function elimRedundantSets is $O(|A_{\text{u}}| \times lc^2_{\text{u,m}} \times |\text{Val}_{\text{s}}|)$. Checking validity of a rule ρ takes time linear in $|[\![\rho]\!]|$. Let lm (mnemonic for "largest meaning") denote the maximum value of $|[\![\rho]\!]|$ among all rules ρ passed as the first argument in a call to elimConstraints, elimConjuncts, or elimElements. The value of lm is at most $|UP_0|$ but typically much smaller. The running time of function elimConstraints is $O((2^{|cc|}) \times lm)$. The running time of function elimConjuncts is $O((2^{|A_{\text{u}}|} + 2^{|A_{\text{r}}|}) \times lm)$. The exponential factors in the running time of elimConstraints and elimConjuncts are small in practice, as discussed above; note that the factor of lm represents the cost of checking validity of a rule. The running time of elimElements is $O(|A_{\text{u}}| \times lm)$. Let le (mnemonic for "largest expressions") denote the maximum of $\text{WSC}(\text{uae}(\rho)) + \text{WSC}(\text{rae}(\rho))$ among rules ρ contained in any set $Rules$ passed as the first argument in a call to simplifyRules. The running time of elimOverlapVal is $O(|Rules_1| \times (|A_{\text{u}}|+|A_{\text{r}}|) \times le)$. The running time of elimOverlapOp is $O(|Rules_1| \times |Op| \times le)$. The factor le in the running times of elimOverlapVal and elimOverlapOp represents the cost of subset checking. The number of iterations of the while loop in line 13–15 is $|Rules_1|$ in the worst case. The overall running time of the algorithm is worst-case cubic in $|UP_0|$.

A.3 Processing Order

This section describes the order in which tuples and rules are processed by our algorithm.

When selecting an element of uncovUP in line 4 of the top-level pseudocode in Figure 5.1, the algorithm selects the user-permission tuple with the highest (according to lexicographic order) value for the following quality metric Q_{up}, which maps user-permission tuples to triples. Informally, the first two components of $Q_{\mathrm{up}}(\langle u, r, o\rangle)$ are the frequency of permission p and user u, respectively, i.e., their numbers of occurrences in UP_0, and the third component is the string representation of $\langle u, r, o\rangle$ (a deterministic although somewhat arbitrary tie-breaker when the first two components of the metric are equal).

$$
\begin{aligned}
\mathrm{freq}(\langle r, o\rangle) &= |\{\langle u', r', o'\rangle \in UP_0 \mid r' = r \wedge o' = o\}| \\
\mathrm{freq}(u) &= |\{\langle u', r', o'\rangle \in UP_0 \mid u' = u\}| \\
Q_{\mathrm{up}}(\langle u, r, o\rangle) &= \langle \mathrm{freq}(\langle r, o\rangle), \mathrm{freq}(u), \mathrm{toString}(\langle \mathrm{u}, \mathrm{r}, \mathrm{o}\rangle)\rangle
\end{aligned}
$$

In the iterations over *Rules* in mergeRules and simplifyRules, the order in which rules are processed is deterministic in our implementation, because *Rules* is implemented as a linked list, loops iterate over the rules in the order they appear in the list, and newly generated rules are added at the beginning of the list. In mergeRules, the workset is a priority queue sorted in descending lexicographic order of rule pair quality, where the quality of a rule pair $\langle \rho_1, \rho_2\rangle$ is $\langle \max(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2)), \min(Q_{\mathrm{rul}}(\rho_1), Q_{\mathrm{rul}}(\rho_2))\rangle$.

A.4 Optimizations

Periodic Merging of Rules. Our algorithm processes UP_0 in batches of 1000 tuples, and calls mergeRules after processing each batch. Specifically, 1000 tuples are selected at random from uncovUP, they are processed in the order described in Section A.3, mergeRules(*Rules*) is called, and then another batch of tuples is processed.

This heuristic optimization is motivated by the observation that merging sometimes has the side-effect of generalization, i.e., the merged rule may cover more tuples than the rules being merged. Merging earlier (compared to waiting until uncovUP is empty) allows additional tuples covered by merged rules to be removed from uncovUP before those tuples are processed by the loop over uncovUP in the top-level pseudocode in Figure 5.1. Without this heuristic optimization, those tuples would be processed by the loop over uncovUP, additional rules would be generated from them, and those rules would probably later get merged with other rules, leading to the same policy.

Caching To compute $[\![\rho]\!]$ for a rule ρ, our algorithm first computes $[\![\mathrm{uae}(\rho)]\!]$ and $[\![\mathrm{rae}(\rho)]\!]$. As an optimization, our implementation caches $[\![\rho]\!]$, $[\![\mathrm{uae}(\rho)]\!]$, and $[\![\mathrm{rae}(\rho)]\!]$ for each rule ρ. Each of these values is stored after the first time it is computed. Subsequently, when one of these values is needed, it is recomputed only if some component of ρ, $\mathrm{uae}(\rho)$ or $\mathrm{rae}(\rho)$, respectively, has changed. In our experiments, this optimization improves the running time by a factor of approximately 8 to 10.

Early Stopping. In the algorithm without noise detection, in mergeRules, when checking validity of ρ_{merge}, our algorithm does not compute $[\![\rho_{\text{merge}}]\!]$ completely and then test $[\![\rho_{\text{merge}}]\!] \subseteq UP_0$. Instead, as it computes $[\![\rho_{\text{merge}}]\!]$, it immediately checks whether each element is in UP_0, and if not, it does not bother to compute the rest of $[\![\rho_{\text{merge}}]\!]$. In the algorithm with noise detection, that validity test is replaced with the test $|[\![\rho_{\text{merge}}]\!] \setminus UP_0| \div |[\![\rho_{\text{merge}}]\!]| \leq \alpha$. We incrementally compute the ratio on the left while computing $[\![\rho_{\text{merge}}]\!]$, and if the ratio exceeds 2α, we stop computing $[\![\rho_{\text{merge}}]\!]$, under the assumption that ρ_{merge} would probably fail the test if we continued. This heuristic decreases the running time significantly. It can affect the result, but it had no effect on the result for the problem instances on which we evaluated it.

A.5 Details of Sample Policies

This section describes the ABAC policies we developed as case studies to illustrate our policy language and evaluate our policy mining algorithm. The number of rules in these case studies is relatively small, but they express non-trivial policies and exercise all the features of our policy language, including use of set membership and superset relations in attribute expressions and constraints. The policy rules and illustrative user attribute data and resource attribute data for all case studies appear in the supplemental material.

The figures in this section contain all rules and some illustrative attribute data for each case study. The policies are written in a concrete syntax with the following kinds of statements. userAttrib($uid, a_1 = v_1, a_2 = v_2, \ldots$) provides user attribute data for a user whose "uid" attribute equals *uid* and whose attributes $a_1, a_2, \ldots$ equal $v_1, v_2, \ldots$, respectively. The resourceAttrib statement is similar. The statement rule(*uae*; *pae*; *ops*; *con*) defines a rule; the four components of this statement correspond directly to the four components of a rule as defined in Section 4.1. In the attribute expressions and constraints, conjuncts are separated by commas. In constraints, the superset relation "$\supseteq$" is denoted by ">", and the contains relation "$\ni$" is denoted by "]".

University Case Study Our university case study is a policy that controls access by students, instructors, teaching assistants, registrar officers, department chairs, and admissions officers to applications (for admission), gradebooks, transcripts, and course schedules. The policy appears in Figure A.1, except that most of the userAttrib and resourceAttrib statements are omitted, to save space. User attributes include position (applicant, student, faculty, or staff), department (the user's department), crsTaken (set of courses taken by a student), crsTaught (set of courses for which the user is the instructor (if the user is a faculty) or the TA (if the user is a student), and isChair (true if the user is the chair of his/her department). Resource attributes include type (application, gradebook, roster, or transcript), crs (the course a gradebook or roster is for, for those resource types), student (the student whose transcript or application this is, for type=transcript or type=application), and department (the department the course is in, for type $\in$ {gradebook, roster}; the student's major department, for type=transcript).

The manually written attribute dataset for this case study contains a few instances of each type of user and resource: two academic departments, a few faculty, a few gradebooks, several students, etc. We generated a series of synthetic datasets, parameterized by the number of

academic departments. The generated userAttrib and resourceAttrib statements for the users and resources associated with each department are similar to but more numerous than the userAttrib and resourceAttrib statements in the manually written dataset; for example, the synthetic datasets contain 20 students, 5 faculty, and 10 courses per academic department.

```
// Rules for Gradebooks
// 1. A user can read his/her own
// scores in gradebooks for
// courses he/she has taken.
rule(; type=gradebook;
   readMyScores; crsTaken ] crs)

// 2. A user (the instructor or TA)
// can add scores and read scores in
// the gradebook for courses
// he/she is teaching.
rule(; type=gradebook; {addScore,
    readScore}; crsTaught ] crs)

// 3. The instructor for a course
// (i.e., a faculty teaching the course)
// can change scores and assign
// grades in the gradebook for that
// course.
rule(position=faculty;
    type=gradebook; {changeScore,
    assignGrade}; crsTaught ] crs)

// Rules for Rosters
// 4. A user in registrar's office can
// read and modify all rosters.
rule(department=registrar;
    type=roster; {read, write}; )

// 5. The instructor for a course
// (i.e., a faculty teaching the course)
// can read the course roster.
rule(position=faculty; type=roster;
    {read}; crsTaught ] crs)

// Rules for Transcripts
// 6. A user can read his/her own
// transcript.
rule(; type=transcript; {read};
    uid=student)

// 7. The chair of a department can
// read the transcripts of all students in
// that department.
rule(isChair=true; type=transcript;
    {read}; department=department)

// 8. A user in the registrar's office can
// read every student's transcript.
rule(department=registrar;
    type=transcript;{read}; )

// Rules for applications and admissions
// 9. A user can check the status of
// his/her own application.
rule(; type=application; {checkStatus};
    uid=student)

// 10. A user in the admissions office can
// read, and update the status of, every
// application.
rule(department=admissions;
    type=application;{read, setStatus}; )

// An illustrative user attribute statement.
userAttrib(csStu2, position=student,
   department=cs, crsTaken={cs601},
   crsTaught={cs101 cs602})
// An illustrative resource attribute statement.
resourceAttrib(cs101gradebook,
   department=cs, crs=cs101,
   type=gradebook)
```

Figure A.1: University case study.

Health Care Case Study Our health care case study is a policy that controls access by nurses, doctors, patients, and agents (e.g., a patient's spouse) to electronic health records (HRs) and HR items (i.e., entries in health records). The policy appears in Figure A.2, except that most of the userAttrib and resourceAttrib statements are omitted, to save space.User attributes include position (doctor or nurse; for other users, this attribute equals $\perp$), specialties (the medical areas that a doctor specializes in), teams (the medical teams a doctor is a member of), ward (the ward a nurse works in or a patient is being treated in), and agentFor (the patients for which a user is an agent). Resource attributes include type (HR for a health record, or HRitem for a health record item), patient (the patient that the HR or HR item is for), treatingTeam (the medical team treating the aforementioned patient), ward (the ward in which the aforementioned patient is being treated), author (author of the HR item, for type=HRitem), and topics (medical areas to which the HR item is relevant, for type=HRitem).

The manually written attribute dataset for this case study contains a small number of instances of each type of user and resource: a few nurses, doctors, patients, and agents, two wards, and a few items in each patient's health record. We generated a series of synthetic datasets, parameterized by the number of wards. The generated userAttrib and resourceAttrib statements for the users and resources associated with each ward are similar to but more numerous than the userAttrib and resourceAttrib statements in the manually written dataset; for example, the synthetic datasets contain 10 patients and 4 nurses per ward.

Project Management Case Study Our project management case study is a policy that controls access by department managers, project leaders, employees, contractors, auditors, accountants, and planners to budgets, schedules, and tasks associated with projects. The policy appears in Figure A.3, except that most of the userAttrib and resourceAttrib statements are omitted, to save space. User attributes include projects (projects the user is working on), projectsLed (projects led by the user), adminRoles (the user's administrative roles, e.g., accountant, auditor, planner, manager), expertise (the user's areas of technical expertise, e.g., design, coding), tasks (tasks assigned to the user), department (department that the user is in), and isEmployee (true if the user is an employee, false if the user is a contractor). Resource attributes include type (task, schedule, or budget), project (project that the task, schedule, or budget is for), department (department that the aforementioned project is in), expertise (areas of technical expertise required to work on the task, for type=task) and proprietary (true if the task involves proprietary information, which is accessible only to employees, not contractors).

The manually written attribute dataset for this case study contains a small number of instances of each type of user (managers, accountants, coders, etc.) and each type of resource (two departments, two projects per department, three tasks per project, etc.). We generated a series of synthetic datasets, parameterized by the number of departments. The generated userAttrib and resourceAttrib statements for the users and resources associated with each department are similar to the userAttrib and resourceAttrib statements in the manually written dataset.

```
// Rules for Health Records
// 1. A nurse can add an item in
// a HR for a patient in the ward
// in which he/she works.
rule(position=nurse; type=HR;
    {addItem}; ward=ward)

// 2. A user can add an item in a
// HR for a patient treated by one
// of the teams of which he/she is
// a member.
rule(; type=HR; {addItem};
    teams ] treatingTeam)

// 3. A user can add an item with
// topic "note" in his/her own HR.
rule(; type=HR; {addNote};
    uid=patient)

// 4. A user can add an item with
// topic "note" in the HR of a
// patient for which he/she is an
// agent.
rule(; type=HR; {addNote};
    agentFor ] patient)

// Rules for Health Record Items
// 5. The author of an item can
// read it.
rule(; type=HRitem; {read};
    uid=author)

// 6. A nurse can read an item with
// topic "nursing" in a HR for a patient
// in the ward in which he/she works.
rule(position=nurse; type=HRitem,
      topics supseteqIn {{nursing}};
      {read}; ward=ward)

// 7. A user can read an item in a HR for
// a patient treated by one of the teams of
// which he/she is a member, if the topics of
// the item are among his/her specialties.
rule(; type=HRitem; {read};
      specialties > topics, teams ] treatingTeam)

// 8. A user can read an item with topic
// "note" in his/her own HR.
rule(; type=HRitem, topics supseteqIn {{note}};
      {read}; uid=patient)

// 9. An agent can read an item with topic
// "note" in the HR of a patient for which
// he/she is an agent.
rule(; type=HRitem, topics supseteqIn {{note}};
      {read}; agentFor ] patient)

// An illustrative user attribute statement.
userAttrib(oncDoc1, position=doctor,
    specialties={oncology},
    teams={oncTeam1, oncTeam2})
// An illustrative resource attribute statement.
resourceAttrib(oncPat1nursingItem,
    type=HRitem, author=oncNurse2,
    patient=oncPat1, topics={nursing},
    ward=oncWard, treatingTeam=oncTeam1)
```

Figure A.2: Health care case study.

Online Video Case Study Our online video case study is a policy that controls access to videos by users of an online video service. The policy appears in Figure A.4, except that most of the userAttrib and resourceAttrib statements are omitted, to save space. It is based on the policy in [YT05], where it is presented as an example of a policy that can be expressed concisely using ABAC but cannot be expressed concisely using RBAC. We modified the policy to use age groups instead of numeric ages. The policy has a more combinatorial character than our other case studies, since permissions depend on combinations of values of multiple user and resource attributes, but not on constraints relating the values of those

```
// 1. The manager of a department
// can read and approve the budget
// for a project in the department.
rule(adminRoles supseteqIn
    {{manager}}; type=budget;
    {read approve};
    department=department)

// 2. A project leader can read and
// write the project schedule and
// budget.
rule( ; type in {schedule, budget};
    {read, write};
    projectsLed ] project)

// 3. A user working on a project
// can read the project schedule.
rule( ; type=schedule; {read};
    projects ] project)
// A user can update the status of
// tasks assigned to him/her.
rule( ; type=task; {setStatus};
    tasks ] rid)

// 4. A user working on a project
// can read and request to work on
// a non-proprietary task whose
// required areas of expertise are
// amonghis/her areas of expertise.
rule( ; type=task, proprietary=false;
    {read request}; projects ] project,
    expertise > expertise)

// 5. An employee working on a
// project can read and request to
// work on any task whose required
// areas of expertise are among
// his/her areas of expertise.
rule(isEmployee=True; type=task;
    {read request}; projects ] project,
    expertise > expertise)

// 6. An auditor assigned to a project can
// read the budget.
rule(adminRoles supseteqIn {{auditor}};
    type=budget; {read}; projects ] project)

// 7. An accountant assigned to a project
// can read and write the budget.
rule(adminRoles supseteqIn {{accountant}};
    type=budget; {read, write};
    projects ] project)

// 8. An accountant assigned to a project
// can update the cost of tasks.
rule(adminRoles supseteqIn {{accountant}};
    type=task; {setCost};
    projects ] project)

// 9. A planner assigned to a project
// can update the schedule.
rule(adminRoles supseteqIn {{planner}};
    type=schedule; {write};
    projects ] project)

// 10. A planner assigned to a project
// can update the schedule (e.g., start
// date, end date) of tasks.
rule(adminRoles supseteqIn {{planner}};
    type=task; {setSchedule};
    projects ] project)

// An illustrative user attribute statement.
userAttrib(des11, expertise={design},
    projects={proj11}, isEmployee=True,
    tasks={proj11task1a, proj11task1propa})
// An illustrative resource attribute statement.
resourceAttrib(proj11task1a, type=task,
    project=proj11,department=dept1,
    expertise={design}, proprietary=false)
```

Figure A.3: Project management case study.

```
// Rules that apply to premium
// members.
// 1. Premium members of all
// ages can view movies rated G.
rule(memberType=premium;
    rating in=G; {view}; )

// 2. Premium teens can view
// movies rated PG.
rule(memberType=premium,
      ageGroup=teen;
      rating=PG; {view}; )

// 3. Premium adults can view
// movies with all ratings.
rule(memberType=premium,
      ageGroup=adult; ; {view}; )

// Rules that apply to all member
// types. These rules correspond
// 1-to-1 with the above rules,
// transformed by dropping the
// restriction to premium members
// and adding the restriction to old videos.
// 4. Members of all ages can
// view old movies rated G.
rule(; videoType=old, rating=G; {view}; )

// 5. Teens can view old movies rated PG.
rule(ageGroup=teen; videoType=old,
      rating=PG; {view}; )
// 6. Adults can view old movies with all
// ratings.
rule(ageGroup=adult; videoType=old;
      {view}; )

// An illustrative user attribute statement.
userAttrib(child1r, ageGroup=child,
              memberType=regular)
// An illustrative resource attribute statement.
resourceAttrib(TheLionKing, rating=G,
              videoType=old)
```

Figure A.4: Online video case study.

attributes. User attributes include ageGroup (child, teen, or adult) and memberType (regular or premium). Every resource is a video. Resource attributes include rating (G, PG-13, or R) and videoType (old or new).

A.6 Example: Processing of a user-permission tuple

Figure A.5 illustrates the processing of the user-permission tuple $t = \langle \text{csFac2}, \text{addScore}, \text{cs601gradebook} \rangle$ selected as a seed (i.e., selected in line 4 of Figure 5.1), in a smaller version of the university sample policy containing only one rule, namely, the second rule in Figure A.1. Attribute data for user csFac2 and resource cs601gradebook appear in Figure A.1.

The edge from t to cc labeled "candidateConstraint" represents the call to candidateConstraint, which returns the set of atomic constraints that hold between csFac2 and cs601gradebook; these constraints are shown in the box labeled cc. The two boxes labeled "addCandidateRule" represent the two calls to addCandidateRule. Internal details are shown for the first call but elided for the second call. The edges from t to e_{u} and from t to e_{r} represent the calls in addCandidateRule to computeUAE and computeRAE, respectively. The call to computeUAE returns a user-attribute expression e_{u} that characterizes the set s_{u} containing users u' with permission $\langle \text{addScore}, \text{cs601gradebook} \rangle$ and such that candidateConstraint(

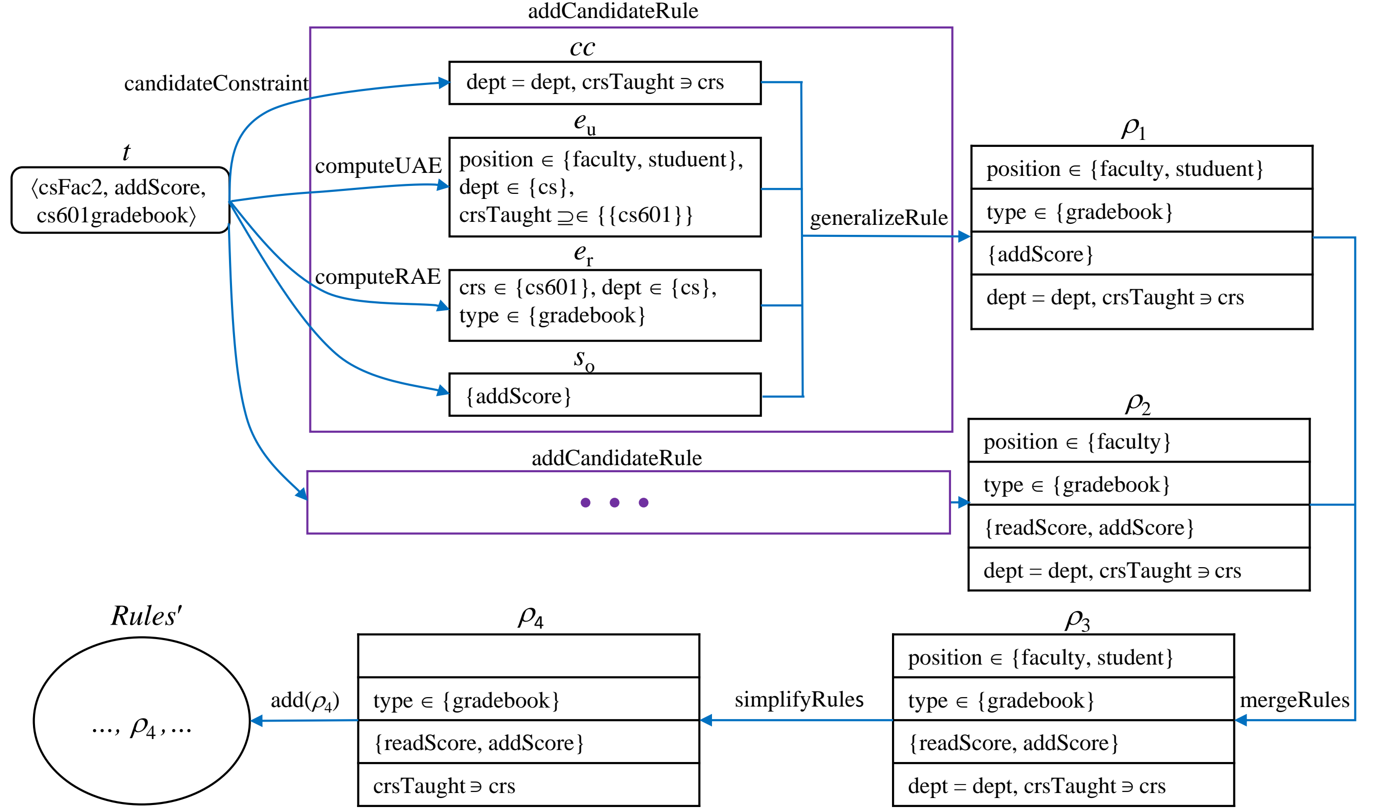

Figure A.5: Diagram representing the processing of one user-permission tuple selected as a seed, in the university sample policy. Rules are depicted as rectangles with four compartments, corresponding to the four components of a rule tuple.

$\text{cs601gradebook}, u') = cc$. The call to computeRAE returns a resource-attribute expression that characterizes {cs601gradebook}. The set of operations considered in this call to addCandidateRule is simply $s_{\rm o} = \{\text{addScore}\}$. The call to generalizeRule generates a candidate rule ρ_1 by assigning $e_{\rm u}$, $e_{\rm r}$ and $s_{\rm o}$ to the first three components of ρ_1, and adding the two atomic constraints in cc to ρ_1 and eliminating the conjuncts in $e_{\rm u}$ and $e_{\rm r}$ corresponding to the attributes mentioned in cc. Similarly, the second call to addCandidateRule generates another candidate rule ρ_2. The call to mergeRules merges ρ_1 and ρ_2 to form ρ_3, which is simplified by the call to simplifyRules to produce a simplified rule ρ_4, which is added to candidate rule set $Rules'$.

A.7 Syntactic Similarity

Syntactic similarity of policies measures the syntactic similarity of rules in the policies. The *syntactic similarity* of rules ρ and ρ', denoted $s_{\rm syn}(\rho, \rho')$, is defined by

$$s_{\rm uae}(e, e') \quad = \quad |A_{\rm u}|^{-1} \sum_{a \in A_{\rm u}} J(e(a), e'(a))$$

$$
\begin{aligned}
s_{\text{rae}}(e, e') &= |A_{\text{r}}|^{-1} \sum_{a \in A_{\text{r}}} J(e(a), e'(a)) \\
s_{\text{syn}}(\rho, \rho') &= \text{mean}(s_{\text{uae}}(\text{uae}(\rho), \text{uae}(\rho')), s_{\text{rae}}(\text{rae}(\rho), \text{rae}(\rho')), \\
&\qquad J(\text{ops}(\rho), \text{ops}(\rho')),\ J(\text{con}(\rho), \text{con}(\rho')))
\end{aligned}
$$

where the Jaccard similarity of two sets is $J(S_1, S_2) = |S_1 \cap S_2|/|S_1 \cup S_2|$.

The *syntactic similarity* of rule sets *Rules* and *Rules'* is the average, over rules ρ in *Rules*, of the syntactic similarity between ρ and the most similar rule in *Rules'*. The *syntactic similarity* of policies is the maximum of the syntactic similarity of the sets of rules in the policies, considered in both orders (this makes the relation symmetric).

$$
\begin{aligned}
s_{\text{syn}}(\mathit{Rules}, \mathit{Rules}') &= |\mathit{Rules}|^{-1} \times \\
&\quad \textstyle\sum_{\rho \in \mathit{Rules}} \max(\{s_{\text{syn}}(\rho, \rho') \mid \rho' \in \mathit{Rules}'\})) \\
s_{\text{syn}}(\pi, \pi') &= \max(s_{\text{syn}}(\text{rules}(\pi), \text{rules}(\pi')), \\
&\qquad s_{\text{syn}}(\text{rules}(\pi'), \text{rules}(\pi)))
\end{aligned}
$$

A.8 ROC Curves for Noise Detection Parameters

When tuning the parameters α and τ used in noise detection (see Section 4.3.1), there is a trade-off between true positives and false positives. To illustrate the trade-off, the Receiver Operating Characteristic (ROC) curve in Figure A.6 shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on α and τ for synthetic policies with 20 rules and 6% noise, split between under-assignments and over-assignments as described in Section 4.4.4. Figure A.7 shows the TPR and FPR for over-assignments. Each data point is an average over 10 synthetic policies. In each of these two sets of experiments, true positives are reported noise (of the specified type, i.e., over-assignments or under-assignments) or that are also actual noise; false negatives are actual noise that are not reported; false positives are reported noise that are not actual noise; and true negatives are user-permission tuples that are not actual noise and are not reported as noise.

Generally, we can see from the ROC curves that, with appropriate parameter values, it is possible to achieve very high TPR and FPR simultaneously, so there is not a significant inherent trade-off between them.

From the ROC curve for under-assignments, we see that the value of τ does not affect computation of under-assignments, as expected, because detection of under-assignments is performed before detection of over-assignments (the former is done when each rule is generated, and the latter is done at the end). We see from the diagonal portion of the curve in the upper left that, when choosing the value of α, there is a trade-off between the TPR and FPR, i.e., having a few false negatives and a few false positives.

From the ROC curve for over-assignments, we see that the value of α affects the rules that are generated, and hence it affects the computation of over-assignments based on those rules at the end of the rule generation process. For $\alpha = 0.01$, when choosing τ, there is some trade-off between the TPR and FPR. For $\alpha \geq 0.02$, the FPR equals 0 independent of τ, so there is no trade-off: the best values of τ are the ones with the highest TPR.

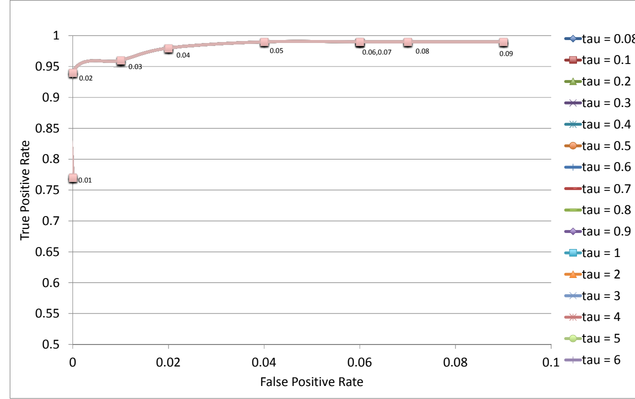

Figure A.6: ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for under-assignments on α and τ.

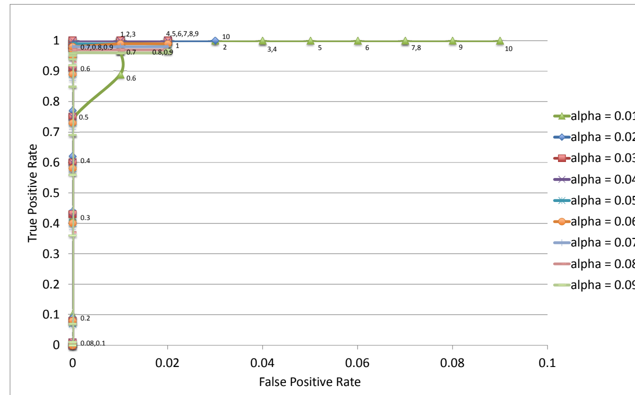

Figure A.7: ROC curve showing shows the dependence of the true positive rate (TPR) and false positive rate (FPR) for over-assignments on α and τ.

A.9 Graphs of Results from Experiments with Permission Noise and Attribute Noise

For the experiments with permission noise and attribute noise described in Section 4.4.4, Figure A.8 shows the Jaccard similarity of the actual and reported over-assignments and the Jaccard similarity of the actual and reported under-assignments, and Figure A.9 shows the semantic similarity of the original and mined policies. Each data point is an average over 10 policies. Error bars show 95% confidence intervals using Student's t-distribution.

A.10 Translation to Inductive Logic Programming

This section describes our translation from the ABAC policy mining problem to inductive logic programming (ILP) as embodied in Progol [MB00, MF01]. Given an ACL policy and attribute data, we generate a Progol input file, which contains type definitions, mode declarations, background knowledge, and examples.

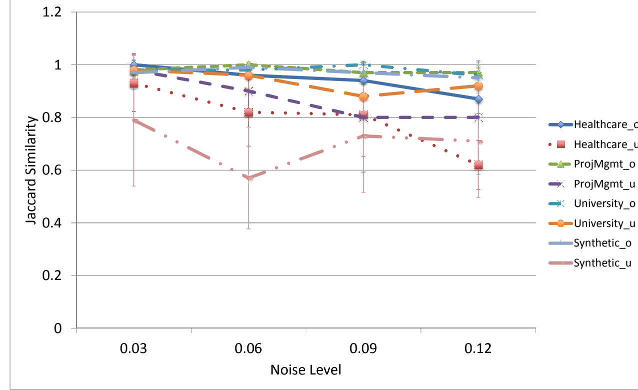

Figure A.8: Jaccard similarity of actual and reported under-assignments, and Jaccard similarity of actual and reported over-assignments, as a function of permission noise level due to permission noise and attribute noise.

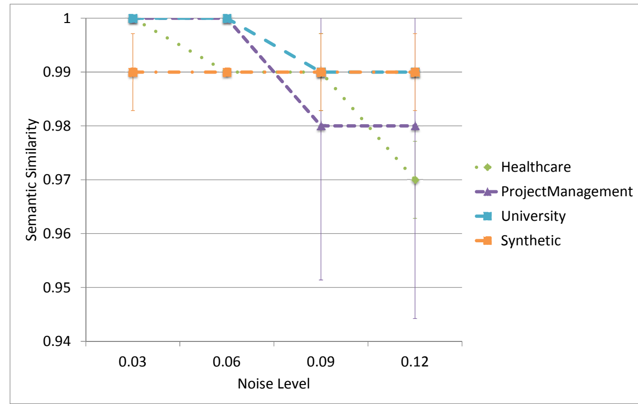

Figure A.9: Semantic similarity of the original policy and the mined policy, as a function of permission noise level due to permission noise and attribute noise.

Type Declarations Type definitions define categories of objects. The types `user`, `resource`, `operation`, and `attribValAtomic` (corresponding to $\mathrm{Val_s}$) are defined by a statement for each constant of that type; for example, for each user u, we generate the statement `user(`u`)`. The type `attribValSet` (corresponding to $\mathrm{Val_m}$) is defined by the rules

```
attribValSet([]).
attribValSet([V|Vs]) :- attribValAtomic(V),
                        attribValSet(Vs).
```

For each attribute a, we define a type containing the constants that appear in values of that attribute in the attribute data; for example, for each value d of the "department" attribute, we generate the statement `departmentType(`d`)`.

Mode Declarations Mode declarations restrict the form of rules that Progol considers, by limiting how each predicate may be used in learned rules. Each head mode declaration `modeh(...)` describes a way in which a predicate may be used in the head (conclusion) of a

learned rule. Each body mode declaration `modeb(...)` describes a way in which a predicate may be used in the body (premises) of a learned rule. Each mode declaration has two arguments. The second argument specifies, for each argument a of the predicate, the type of a and whether a may be instantiated with an input variable (indicated by "`+`"), an output variable (indicated by "`-`"), or a constant (indicated by "`#`"). The first argument, called the *recall*, is an integer or `*`, which bounds the number of values of the output arguments for which the predicate can hold for given values of the input arguments and constant arguments; "`*`" indicates no bound. The specification of predicate arguments as inputs and outputs also limits how variables may appear in learned rules. In a learned rule h `:-` $b_1, \ldots, b_n$, every variable of input type in each premise b_i must appear either with input type in h or with output type in some premise b_j with $j < i$.

We generate only one head mode declaration:

```
modeh(1, up(+user, +resource, #operation))
```

This tells Progol to learn rules that define the user-permission predicate `up`.

For each single-valued user attribute a, we generate a body mode declaration `modeb(1,` a`U(+user, #`a`Type))`. For example, the mode declaration for a user attribute named "department" is `modeb(1, departmentU(+user, #departmentType))`. We append "`U`" to the attribute name to prevent naming conflicts in case there is a resource attribute with the same name. Mode declarations for multi-valued user attributes are defined similarly, except with "`*`" instead of `1` as the recall. Mode declarations for resource attributes are defined similarly, except with `R` instead of `U` appended to the attribute name. We tried a variant translation in which we generated a second body mode declaration for each attribute, using `-`a`Type` instead of `#`a`Type`, but this led to worse results.

We also generate mode declarations for predicates used to express constraints. For each single-valued user attribute a and single-valued resource attribute $\bar{a}$, we generate a mode declaration `modeb(1,` a`U_equals_`$\bar{a}$`R(+user,+resource))`; the predicate aU_equals_$\bar{a}$R is used to express atomic constraints of the form $a = \bar{a}$. The mode declarations for the predicates used to express the other two forms of atomic constraints are similar, using user and resource attributes with appropriate cardinality, and with "`contains`" (for $\ni$) or "`superset`" (for $\supseteq$) instead of "`equals`" in the name of the predicate.

Background Knowledge The attribute data is expressed as background knowledge. For each user u and each single-valued user attribute a, we generate a statement a`U(`u, v`)` where $v = d_{\mathrm{u}}(u, a)$. For each user u and each multi-valued user attribute a, we generate a statement a`U(`u, v`)` for each $v \in d_{\mathrm{u}}(u, a)$. Background knowledge statements for resource attribute data are defined similarly.

Definitions of the predicates used to express constraints are also included in the background knowledge. For each equality predicate a_equals_$\bar{a}$ mentioned in the mode declarations, we generate a statement a`U_equals_`$\bar{a}$`R(U,R) :- `a`U(U,X), `$\bar{a}$`R(R,X)`. The definitions of the predicates used to express the other two forms of constraints are

```
aU_contains_āR(U,R) :- aU(U,X), āR(R,X).
aU_superset_āR(U,R) :- setof(X, aU(U,X), SU),
                       setof(Y, āR(R,Y), SR),
```

```
                        superset(SU,SR),
                        not(SR==[]).
superset(Y,[A|X]) :- element(A,Y),
                     superset(Y,X).
superset(Y,[]).
```

The premise `not(SR==[])` in the definition of a`U_superset_`$\bar{a}$`R` is needed to handle cases where the value of $\bar{a}$ is $\perp$. The predicates `setof` and `element` are built-in predicates in Progol.

Examples A *positive example* is an instantiation of a predicate to be learned for which the predicate holds. A *negative example* is an instantiation of a predicate to be learned for which the predicate does not hold. For each $\langle u, r, o\rangle \in U \times R \times Op$, if $\langle u, r, o\rangle \in UP_0$, then we generate a positive example `up`(u, r, o), otherwise we generate a negative example `:-` `up`(u, r, o) (the leading "`:-`" indicates that the example is negative). The negative examples are necessary because, without them, Progol may produce rules that hold for instantiations of `up` not mentioned in the positive examples.

Appendix B

Supplemental Material on ABAC Mining from Logs

B.1 Rule Quality Metric Based On Inductive Logic Programming

We review the theory quality metric used in Progol [MB00, MF01], a well-known ILP system, and then describe our design of a rule quality metric based on it. Progol's IPL algorithm works as follows. At the outermost level, Progol uses a loop that repeatedly generalizes an example to a hypothesized rule and then removes examples which are redundant relative to (i.e., covered by) the new rule, until no examples remain to be generalised. When generalizing an example, Progol uses a metric, called a *compression metric*, to guide construction of the hypotheses. When mining ABAC policies from operation logs, user-permissions tuples in $UP(L)$ are positive examples, and no negative examples are available. Thus, this corresponds to the case of learning from only positive data. When learning from only positive data, Progol's compression metric *pcomp* is defined as follows [Mug95].

$$\begin{aligned} f_m(H) &= c \times 2^{-|H|}(1-g(H))^m && \text{(B.1)} \\ pcomp(H,E) &= \log_2 \frac{f_m(H)}{f_m(E)} && \text{(B.2)} \\ &= |E| - |H| - m(\log_2(1-g(E)) \\ &\quad - \log_2(1-g(H))) \\ &\approx |E| - |H| + m\log_2(1-g(H)) \end{aligned}$$

where E is the set of positive examples, H is the entire theory (i.e., ABAC policy, in our context) being generated, including the part not generated yet, $m = |E|$, $|H|$ is the size of H, measured as the number of bits needed to encode H, and c is a constant chosen so that $\sum_{H \in \mathcal{H}} f_m(H) = 1$, where $\mathcal{H}$ is the set of all candidate theories (note that f, like *pcomp* is a function of H and E, since $m = |E|$, but we adopt Muggleton's notation of $f_m(H)$ instead of using the more straightforward notation $f(H,E)$). Let X be the set of all possible well-formed examples (in our context, X is the set of all user-permission tuples). $g(H)$ (the "generality" of H) is the probability that an element of X, randomly selected following a

uniform distribution, satisfies H. f_m and *pcomp* can be regarded as policy quality metrics. The term $2^{-|H|}$ in the definition of f_m causes policies with smaller size to have higher quality, and the term $(1 - g(H))^m$ causes policies with larger meaning to have higher quality. In [Mug95], f_m is used to guide the search for rules. Using *pcomp* to guide the search would have the same effect, because in the definition of *pcomp* in equation (B.2), $f_m(E)$ is a constant, and $\log_2 \frac{f_m(H)}{f_m(E)}$ is a monotonic function of $f_m(H)$, so maximizing f_m is equivalent to maximizing *pcomp*.

A difficulty with using f_m to guide generation of a rule to add to a partly generated theory is that the entire theory H is not yet known. To overcome this difficulty, the quality of the entire theory is estimated by extrapolation. Let C_i denote the i'th rule added to the theory, and let $H_i = \{C_1, \ldots, C_i\}$. Let n denote the number of rules in the entire theory being generated (of course, n is not known until the algorithm terminates). When generating C_i, the policy quality $f_m(H_n)$ is estimated as follows.

$$c \times 2^{-(\frac{m}{p} \times |C_i|)} \times (1 - \frac{m}{p} * (g(H_i) - g(H_{i-1})))^m \tag{B.3}$$

where p is the number of examples in E that are implied by C_i and not by H_{i-1}.

Now we describe how to modify our policy mining algorithm to use f_m as a rule quality metric. In the loop in the top-level pseudocode in Figure 5.1 that builds the set *Rules* of candidate rules, rule quality is computed using equation B.3 with $H_{i-1} = \mathit{Rules}$ and $C_i = \rho$. Specifically, in the definition of generalizeRule in 6.3, $Q_{\mathrm{rul}}(\rho, \mathrm{uncovUP})$ is replaced with f_m computed using equation B.3 with $H_{i-1} = \mathit{Rules}$ and $C_i = \rho$. This closely corresponds to the usage in Progol, although it is slightly different, because some of the candidate rules in *Rules* will not be included in the final set of rules Rules'.

In the loop in the top-level pseudocode in Figure 5.1 that builds the set Rules' of final rules, we take the same approach, except using Rules' instead of *Rules*. Specifically, $Q_{\mathrm{rul}}(\rho, \mathrm{uncovUP})$ is replaced with f_m computed using equation B.3 with $H_{i-1} = \mathit{Rules}'$ and $C_i = \rho$.

To compute rule quality after building the set *Rules* of candidate rules and before building the set Rules' of final rules, the algorithm is modifying a set of rules, not extending a set of rules, so f_m can be evaluated using equation B.1, with *Rules* as an estimate of the entire policy H. Specifically, in the calls to elimConjuncts and elimConstraints from simplifyRules, $Q_{\mathrm{rul}}(\rho'', UP_0)$ is replaced with f_m computed using equation B.1 with $H = \mathit{Rules} \backslash \rho \cup \{\rho''\}$, and $Q_{\mathrm{rul}}(\rho_{\mathrm{best}}, UP_0)$ is replaced with f_m computed using equation B.1 with $H = \mathit{Rules} \backslash \rho \cup \{\rho_{\mathrm{best}}\}$.

Bibliography

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[BGR08] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proc. 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 185–194. ACM, 2008.

[BM13] Matthias Beckerle and Leonardo A. Martucci. Formal definitions for usable access control rule sets—From goals to metrics. In *Proceedings of the Ninth Symposium on Usable Privacy and Security (SOUPS)*, pages 2:1–2:11. ACM, 2013.

[CDPOV09] Alessandro Colantonio, Roberto Di Pietro, Alberto Ocello, and Nino Vincenzo Verde. A formal framework to elicit roles with business meaning in RBAC systems. In *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 85–94, 2009.

[CDV12] Alessandro Colantonio, Roberto Di Pietro, and Nino Vincenzo Verde. A business-driven decomposition methodology for role mining. *Computers & Security*, 31(7):844–855, October 2012.

[EHM+08] Alina Ene, William G. Horne, Nikola Milosavljevic, Prasad Rao, Robert Schreiber, and Robert Endre Tarjan. Fast exact and heuristic methods for role minimization problems. In *Proc. 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 1–10. ACM, 2008.

[FSBB09] Mario Frank, Andreas P. Streich, David A. Basin, and Joachim M. Buhmann. A probabilistic approach to hybrid role mining. In *ACM Conference on Computer and Communications Security (CCS)*, pages 101–111. ACM, 2009.

[GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proc. 2nd ACM Workshop on Role Based Access Control (RBAC'97)*, pages 153–159, November 1997.

[GO04] Mei Ge and Sylvia L. Osborn. A design for parameterized roles. In *Research Directions in Data and Applications Security XVIII, IFIP TC11/WG 11.3 Eighteenth Annual Conference on Data and Applications Security*, pages 251–264. Kluwer, 2004.

[GOGY+11] Nurit Gal-Oz, Yaron Gonen, Ran Yahalom, Ehud Gudes, Boris Rozenberg, and Erez Shmueli. Mining roles from web application usage patterns. In *Proc. 8th Int'l. Conference on Trust, Privacy and Security in Digital Business (TrustBus)*, pages 125–137. Springer, 2011.

[GVA08] Qi Guo, Jaideep Vaidya, and Vijayalakshmi Atluri. The role hierarchy mining problem: Discovery of optimal role hierarchies. In *Proc. 2008 Annual Computer Security Applications Conference (ACSAC)*, pages 237–246. IEEE Computer Society, 2008.

[HFK+13] Vincent C. Hu, David Ferraiolo, Rick Kuhn, Arthur R. Friedman, Alan J. Lang, Margaret M. Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (abac) definition and considerations (final draft). NIST Special Publication 800-162, National Institute of Standards and Technology, September 2013.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[Lim10] Yow Tzu Lim. *Evolving Security Policies.* PhD thesis, University of York, 2010.

[LLM+07] Ninghui Li, Tiancheng Li, Ian Molloy, Qihua Wang, Elisa Bertino, Seraphic Calo, and Jorge Lobo. Role mining for engineering and optimizing role based access control systems. Technical Report 2007-60, CERIAS, Purdue University, November 2007.

[LM07] Ninghui Li and Ziqing Mao. Administration in role based access control. In *Proc. ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, pages 127–138. ACM, March 2007.

[LS97] Emil Lupu and Morris Sloman. Reconciling role based management and role based access control. In *Proc. 2nd ACM Workshop on Role Based Access Control (RBAC'97)*, pages 135–141, November 1997.

[LVA08] Haibing Lu, Jaideep Vaidya, and Vijayalakshmi Atluri. Optimal Boolean matrix decomposition: Application to role engineering. In *Proc. 24th International Conference on Data Engineering (ICDE)*, pages 297–306. IEEE, 2008.

[MB00] Stephen Muggleton and Christopher H. Bryant. Theory completion using inverse entailment. In James Cussens and Alan M. Frisch, editors, *Proc. 10th International Conference on Inductive Logic Programming (ILP)*, pages 130–146. Springer, 2000.

[MCL+10] Ian Molloy, Hong Chen, Tiancheng Li, Qihua Wang, Ninghui Li, Elisa Bertino, Seraphin B. Calo, and Jorge Lobo. Mining roles with multiple objectives. *ACM Trans. Inf. Syst. Secur.*, 13(4), 2010.

[MF01] Stephen H. Muggleton and John Firth. CProgol4.4: a tutorial introduction. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 160–188. Springer-Verlag, 2001.

[MLC11] Ian Molloy, Jorge Lobo, and Suresh Chari. Adversaries' holy grail: Access control analytics. In *Proc. First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS 2011)*, pages 52–59, 2011.

[MLL+09] Ian Molloy, Ninghui Li, Tiancheng Li, Ziqing Mao, Qihua Wang, and Jorge Lobo. Evaluating role mining algorithms. In *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 95–104. ACM, 2009.

[MLQ+10] Ian Molloy, Ninghui Li, Yuan (Alan) Qi, Jorge Lobo, and Luke Dickens. Mining roles with noisy data. In *Proceeding of the 15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 45–54. ACM, 2010.

[Mol11] Ian Molloy. Private communication, December 2011.

[MPC12a] Ian Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 45–56. ACM, 2012.

[MPC12b] Ian Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM, 2012.

[MSAV13] Barsha Mitra, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya. Toward mining of temporal roles. In *Proc. 27th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy (DBSec)*, volume 7964 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2013.

[Mug95] Stephen H. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13:245–286, 1995.

[NLC+09] Qun Ni, Jorge Lobo, Seraphin Calo, Pankaj Rohatgi, and Elisa Bertino. Automating role-based provisioning by learning from examples. In *Proc. 14th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 75–84. ACM, 2009.

[RZCG+10] Michal Rosen-Zvi, Chaitanya Chemudugunta, Thomas Griffiths, Padhraic Smyth, and Mark Steyvers. Learning author-topic models from text corpora. *ACM Transactions on Information Systems*, 28(1):4:1–4:38, January 2010.

[SBM99] Ravi S. Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Trans. Inf. Syst. Secur.*, 2(1):105–135, 1999.

[SR05] Gabor J. Szekely and Maria L. Rizzo. Hierarchical clustering via joint between-within distances: Extending ward's minimum variance method. *J. Classification*, 22(2):151–183, 2005.

[VAG07] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: finding a minimal descriptive set of roles. In *Proc. 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 175–184. ACM, 2007.

[VAGA08] Jaideep Vaidya, Vijayalakshmi Atluri, Qi Guo, and Nabil Adam. Migrating to optimal RBAC with minimal perturbation. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 11–20. ACM, 2008.

[VAW06] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. RoleMiner: Mining roles using subset enumeration. In *Proc. 13th ACM Conference on Computer and Communications Security (CCS)*, pages 144–153. ACM, 2006.

[VAWG10] Jaideep Vaidya, Vijayalakshmi Atluri, Janice Warner, and Qi Guo. Role engineering via prioritized subset enumeration. *IEEE Trans. Dependable Secur. Comput.*, 7(3):300–314, 2010.

[VVAC12] Nino Vincenzo Verde, Jaideep Vaidya, Vijay Atluri, and Alessandro Colantonio. Role engineering: From theory to practice. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 181–192. ACM, 2012.

[XAC] eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xacml/`.

[XS12] Zhongyuan Xu and Scott D. Stoller. Algorithms for mining meaningful roles. In *Proc. 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–66. ACM, 2012.

[XS13a] Zhongyuan Xu and Scott D. Stoller. Mining attribute-based access control policies. *Computing Research Repository (CoRR)*, abs/1306.2401, June 2013. Revised January 2014. `http://arxiv.org/abs/1306.2401`.

[XS13b] Zhongyuan Xu and Scott D. Stoller. Mining parameterized role-based policies. In *Proc. Third ACM Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2013.

[YT05] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *Proc. 2005 IEEE International Conference on Web Services (ICWS)*, pages 561–569. IEEE Computer Society, 2005.

[ZCG+13] Wen Zhang, You Chen, Carl A. Gunter, David Liebovitz, and Bradley Malin. Evolving role definitions through permission invocation patterns. In *Proc. 18th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 37–48. ACM, 2013.

[ZGL+11] Wen Zhang, Carl A Gunter, David Liebovitz, Jian Tian, and Bradley Malin. Role prediction using electronic medical record system audits. In *AMIA Annual Symposium Proceedings*, pages 858–867. American Medical Informatics Association, 2011.

[ZRE07] Dana Zhang, Kotagiri Ramamohanarao, and Tim Ebringer. Role engineering using graph optimisation. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, pages 139–144, 2007.