On the Design and Testing of Authorization Systems

by

Alireza Sharifi

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

© Alireza Sharifi 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Authorization deals with the specification and management of accesses principals have to resources. In the design of an authorization system, sometimes we just implement the access-enforcement without having a precise semantics for it. In this dissertation we show that, there exists a precise semantics that improves the efficiency of access-enforcement over the access-enforcement without precise semantics. We present an algorithm to produce an Access Control List (ACL), in a particular authorization system for version control syatems called `gitolite`, and we compare the implementation of our algorithm against the implementation that is already being used.

As another design problem, we consider least-restrictive enforcement of the Chinese Wall security policy. We show that there exists a least-restrictive enforcement of the Chinese Wall Security Policy. Our approach to proving the thesis is by construction; we present an enforcement that is least-restrictive. We also prove that such an enforcement mechanism cannot be subject-independent.

We also propose a methodology that tests the implementation of an authorization system to check whether it has properties of interest. The properties may be considered to be held in the design of an authorization system, but they are not held in the implementation. We show that there exist authorization systems that do not have the properties of interest.

Acknowledgements

I would like to thank all the people who made this possible.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Access Control 1
1.2 Life cycle for an authorization system 2
1.3 Version Control System Design 2
1.4 Chinese Wall Security Policy 3
1.5 Property Testing 4
1.5.1 Thesis Statements 4

2 The Design and Semantics of an Authorization Scheme for Version Control Systems 6

2.1 Introduction 6
2.1.1 Version Control Systems 6
2.2 Authentication and Access Enforcement 7
2.3 Authorization Scheme 8
2.3.1 Classes of Users 9
2.3.2 Administrative and Authorization Specifications 10
2.3.3 Compilation 11
2.4 Semantics 15
2.4.1 A Term Rewriting System (TRS) representing `gitolite` 16
2.4.2 Producing the ACL using rewrite rules 17
2.4.3 Authorization Priority Factor (APF) 18

2.4.4 TRS Algorithm 19
2.5 Experience from Deployments 30
2.5.1 Experimental Setup 30
2.5.2 Impact of number of users with access request 30
2.5.3 Impact of deny rules with increasing resources 31
2.6 Conclusion 32

3 Least-Restrictive Enforcement of the Chinese Wall Security Policy 33
3.1 Introduction 33
3.2 A Formalism and the Chinese Wall Security Policy 36
3.2.1 Authorization State 36
3.2.2 The Chinese Wall Security Policy 37
3.2.3 Changes to the Authorization State 39
3.3 Enforcement 40
3.3.1 Axes for Trade-offs 43
3.4 Our Approach 45
3.4.1 Soundness 46
3.4.2 Least-Restriction 49
3.4.3 Other Axes 51
3.4.4 Uniqueness of Our Enforcement Mechanisms 52
3.5 The Brewer-Nash Approach 53
3.5.1 New Properties 55
3.5.2 Bishop's Rendition [32] 56
3.6 Related Work 57
3.7 Conclusion 59

4 Property-Testing Real-World Authorization Systems 60
4.1 Introduction 60
4.2 More Details on the Testing Methodology 63
4.2.1 Relationship to Model-Checking 65
4.2.2 A Testing System 66

4.3 Application 1: the $\mathcal{D}$ System . . . 67
4.3.1 Properties of interest . . . 69
4.3.2 Traces . . . 70
4.3.3 Relating Properties and Traces . . . 72
4.3.4 Results for the $\mathcal{D}$ system . . . 74
4.3.5 Summary of results . . . 75
4.4 Application 2: the $\mathcal{I}$ System . . . 77
4.4.1 Properties of interest . . . 78
4.4.2 Traces . . . 80
4.4.3 Relating Properties and Traces . . . 82
4.4.4 Results for the $\mathcal{I}$ System . . . 84
4.5 Related Work . . . 84
4.6 Conclusion . . . 85

5 Conclusion **87**

References **89**

List of Tables

4.1 Temporal operators that we use. 63

4.2 Our complete results for our tests for the $\mathcal{D}$ System with $m = 3$. For each case (e.g., $\mathsf{def}(p_1), \mathsf{def}(p_2), \neg\mathsf{def}(p_3)$), we have five rows that correspond to the rights of the users $u_0, \ldots, u_4$, respectively. We have four columns, that correspond to the table p_0, and procedures $p_1, \ldots, p_3$. A "1" indicates that that user needs to have the privilege to that procedure/table so the necessity and security properties are satisfied. $\mathcal{D}_1$ corresponds to the system immediately after we create all the procedures, and $\mathcal{D}_2$ corresponds to the system at a later time, after we perform some actions on it. 76

List of Figures

2.1 A version control system in BNF. The notation $::=$ stands for "comprises," and $\{\cdot\}$ means 0 or more occurrences. 7

2.2 Access enforcement to version control system objects using the `gitolite` reference monitor. 7

2.3 We have four classes of users. The transitive closure of the arrows represents "may delegate to." 9

2.4 The Scheme 27

2.5 Access times when increasing the number of users accessing allowed access to the resource. 31

2.6 Access times when varying the number of resources while denying a certain number of users access to the every resource. 5 gitolite means that there are 5 users that are denied access to every resource. 32

3.1 An authorization state with two subjects and six objects. As the legend expresses, boxes with sharp corners are Conflict of Interest classes and boxes with rounded edges are Company Datasets. An arrow indicates read or write at a particular time. The Conflict of Interest class to the far right, and the Company Dataset within it contain sanitized objects only. 34

3.2 Changes to the authorization state starting at the state in Figure 3.1. We show only the Conflict of Interest class that contains o_1 and o_2. In the state to the far left, the subject s_3 is created. In the next state, s_3 reads o_2. In the next state, o_2 is destroyed. In the final state to the far right, s_3 writes to o_1 causing an object-violation. All these states are reachable from the state in Figure 3.1. 37

3.3 A subject-violation example. The policy is not satisfied for subjects in this state. . 38

3.4 An object-violation example. The policy is not satisfied for objects in this state. . 39

3.5 Our enforcement algorithm for the state-changes from Figure 3.2. We denote $\delta(o_1) = \delta_1$ and $\delta(o_2) = \delta_2$. The enforcement state that corresponds to each subject and object is shown in "{ }." The figure to the far left indicates that no information from any Company Dataset other than the one to which each belongs has flowed into o_1 and o_2. As s_3 is created in that state, the set that corresponds to it is empty. The write by s_3 in the final state is denied because the "if" condition in WRITE evaluates to true. 43

3.6 An example of non-transitive conflict of interest relation 45

4.1 The four stages in our testing methodology. 61

4.2 Our testing system. The "interactor," "trace-instance-validator," and "trace-instance-checker" are automata (programs or humans). In our current implementation, the interactor and the trace-instance-validator are human-driven. We discuss the components in Section 4.2.2. 65

4.3 An example with four users in the $\mathcal{D}$ System, Alice, Bob, Carl and Dan. Alice owns the table `t`, and grants insert privilege over it to Bob. Bob defines the procedure `def` to be definer's rights and grants Carl execute privilege to it. The procedure `def` inserts a row into Alice's table. Carl creates an invoker's rights procedure `inv` and grants Dan execute privilege to it. The procedure `inv` invokes Bob's procedure `def`. Finally, Dan attempts to execute Carl's procedure `inv`. 68

Chapter 1

Introduction

This dissertation addresses three important problems in authorization systems. The first two problems are in the context of design. The challenge in design is proposing a precise semantics for the designed system or modifying the system based on the proposed semantics. In this dissertation, we undertake the challenge of designing a version control system and proposing its precise semantics using term rewriting rules. In the second problem in design, we give a least-restrictive enforcement of the Chinese Wall security policy. The third problem we address is in the context of testing an implementation of an authorization system. Authorization systems that allow authorizing a principal indirectly are interesting and thus considered. We need to test such systems to ensure that they posses the properties of interest. In section 1.1 we consider authorization systems as an important aspect of security. In section 1.2 we talk about life cycle of an authorization system. In Sections 1.3 through 1.5 we give a brief description of problems we consider in this dissertation.

1.1 Access Control

Access control is one of the most important aspects of the security of a system. With it, one is able to regulate who may perform certain actions such as read or write on resources. Anderson in [1] characterizes Access Control as "... the traditional center of gravity of computer security. It is where security engineering meets computer science. Its function is to control which principals (persons, processes, machines, . . .) have access to which resources in the system, which files they can read, which programs they can execute, how they share data with other principals, and so on." Access control deals with the specification and management of accesses principals have to resources. It is an important aspect of security.

Authorization systems can be complex; this is the underlying technical challenge in our work. The complexity arises from a feature that realistic authorization systems have: it is possible to not only directly authorize a principal to a resource, but also to do so indirectly. Examples of indirect ways are delegation, groups and roles. The reason such indirect ways are allowed is to balance

the scalability needs of enterprise authorization systems, and security. Enterprise authorization systems must typically support large numbers of principals (e.g., users) and resources (e.g., files and documents). Providing indirect authorizations eases administrative burden. Granting a right to a group, for example, is less burdensome than granting it to each member of the group.

1.2 Life cycle for an authorization system

To have an authorization system for a specific application, There are several challenges in the procedure of design:

1. Identify needs and requirements: Before designing an authorization system, we have to have a good understanding of what we need. For example, for designing an access control model that is used for an enterprise, we should provide a set of entities and the structure needed to express the access restriction for all resources [2].
2. Design: After knowing about needs and requirements, we design the required authorization scheme with a precise semantics to generate an Access Control List (ACL). Sometimes in this step, you compare your design with previous designs and estimate the improvements. In all pieces of work we do in this dissertation we compare our new designs with their previous designs.
3. Verification: When we design an authorization system, the output is what we specify but it does not necessarily match the need. Like software development we use formal verification in authorization systems to check the differences between what we want and what we get.
4. Testing: This step ascertains whether the implementation of the authorization system has the desired security and availability properties. Considering only the verification step is not enough because the implementation may not necessarily satisfy your verified authorization system features.

In this dissertation we focus on steps 2 and 4. We give a brief justification why we choose these two steps. We argue that in step 2, we can identify needs. In step 4, we can use many techniques that have been proposed in step 3. Also testing is more considerable in practice than verification, because in testing we consider a real system rather than a model.

1.3 Version Control System Design

We present `gitolite` [3, 4], a new authorization scheme for version control systems. Authorization deals with the specification and management of the rights users have to resources. It is an important aspect of the security of a system. A version control system is used when resources

(e.g., software programs) are in flux, and the history of changes need versioning. Typically, a version control system needs to allow multiple users to access, modify and update the resources.

`gitolite` has been implemented for the Git VCS [5] and is sufficiently general to be applicable to other version control systems such as SVN [6]. Our particular focus in this dissertation is our design of controlled delegation that is part of the scheme. Traditional authorization schemes for version control systems have a single administrator for an instance of a version control system that specifies authorization rules for accessors. Our scheme is more flexible and scalable as it has a finer-grained delegation model.

In the context of designing an authorization scheme for version control systems, we have set a broader goal for ourselves: to design a scheme that simultaneously addresses expressive power, usability and correctness. In this context, our intent is only for our scheme to be applicable to version control systems; it is not designed to be as expressive as more general schemes from the research literature. A natural question then is whether schemes that have been proposed previously, such as those for Discretionary Access Control (DAC) [7], Role-Based Access Control (RBAC) [8, 9] and Trust Management [10, 11], are expressive enough to subsume our scheme. Our observation is that other schemes do not capture `gitolite` in a manner that is useful for real-world deployments. This should not be surprising: it has been observed before that even from a more rigorous standpoint, given two authorization schemes, neither may be as expressive as the other [12]. Furthermore, the application-domain has a significant influence on the design of an authorization scheme and its expressive power. Therefore, it is likely that schemes from different application-domains are incomparable with one another.

1.4 Chinese Wall Security Policy

Conflict of interest is related to flow of information: if two pieces of information are to be confidential from one another, then a subject should not have access to both.

The importance of conflict of interest has been recognized in the past in various settings: legal, financial and governmental [13], and technical [14]. In particular, in recent research, its importance has been recognized in cloud-computing environments [15, 16]. Conflict of interest has been addressed in past research in information security. The work of Brewer and Nash [17] articulates a conflict of interest policy, and proposes its enforcement with what it calls Chinese Walls. To our knowledge, it is the first piece of work on conflict of interest in information security research. Their work is generally recognized as important to the foundations of computer security, and is cited widely in the research literature [18].

We revisit conflict of interest from the standpoint of enforcement [19]. Work subsequent to that of Brewer and Nash (see, for example, [20, 21, 22]) has pointed out that the Brewer-Nash approach to enforcement can be highly restrictive. What this means is that the approach can unnecessarily preclude a system from reaching authorization states that do not violate conflict of interest. Such work then proposes new approaches to enforcement that are seemingly less restrictive.

We make several observations about past approaches to enforcement [19]. For example, the Brewer-Nash approach is more restrictive than previously thought, and subsequent approaches are also restrictive in that they preclude states that do not violate conflict of interest. We consider enforcement of conflict of interest with what we call *least-restrictive* enforcement. By least-restrictive, we mean that every state that does not violate conflict of interest is reachable. We devise an approach, and show that it is least-restrictive. Of course, there are trade-offs in achieving this least-restriction. To precisely identify what the trade-offs are, we articulate several axes of goodness for an approach to enforcement, and analyze our approach and those from prior work along these axes. Examples of the axes we consider are time-efficiency, space-efficiency and whether the actions of a subject can impact the prospective actions of another subject.

1.5 Property Testing

Authorization deals with the specification and management of accesses principals have to resources. It is an important aspect of security. We address the problem of testing implementations of authorization systems for security properties that are of interest [23].

Given an implementation of an authorization system that supports indirect authorizations, one may want to test it. The objective is to ascertain whether the implementation has certain security and availability properties of interest. For example, in an authorization system that supports the notion of groups, we may want to ask, "if Alice is a member of the group G at the time G is authorized to read file f, then she should be authorized to read file f. Furthermore, this authorization should hold henceforth, so long as both the direct authorizations hold." The two direct authorizations, in the preceding example, refer to Alice's membership in G, and G's authorization to f.

Prior foundational work suggests that verifying that an authorization system has such a property may be intractable or even undecidable [24, 25, 26]. This is the case even if we want to carry out such verification "on paper," i.e., considering only the design of the system, and not any implementation. Notwithstanding this foundational issue, the problem we address, that of testing an implementation for properties, is clearly an important one. If an implementation does not have a property we seek, we either need to redesign it, or address some implementation bug that causes it to not have the property.

1.5.1 Thesis Statements

Our thesis statements are as follow:

- In the design of an authorization system, there exists a precise semantics that improves the efficiency of access-enforcement over the access-enforcement without precise semantics. Our approach to proving the thesis is by construction; we present an algorithm to produce

an Access Control List (ACL), in a particular authorization system for version control syatems called `gitolite`, and we compare the implementation of our algorithm against the implementation that is already being used.

- There exists a least-restrictive enforcement of the Chinese Wall Security Policy. Our approach to proving the thesis is by construction; we present an enforcement that is least-restrictive. We also prove that such an enforcement mechanism cannot be subject-independent.

- There exists a methodology that tests the implementation of an authorization system to check whether it has properties of interest. The properties may be considered to be held in the design of an authorization system, but they are not held in the implementation. Our approach to proving the thesis is by construction; we present a methodology to check whether an implementation of an authorization system has properties of interest and show that there exist authorization systems that do not have the properties of interest.

Chapter 2

The Design and Semantics of an Authorization Scheme for Version Control Systems

2.1 Introduction

Authorization deals with the specification and management of the rights users have to resources. It is an important aspect of the security of a system. A version control system is used when resources (e.g., software programs) are in flux, and the history of changes need versioning. Typically, a version control system needs to allow multiple users to access, modify and update the resources.

`gitolite` has been implemented for the Git version control system [5] and is sufficiently general to be applicable to other version control systems such as SVN [6]. Our particular focus in this dissertation is our design of controlled delegation that is part of the scheme. Traditional authorization schemes for version control systems have a single administrator for an instance of a version control system that specifies authorization rules for accessors. Our scheme is more flexible and scalable as it has a finer-grained delegation model.

2.1.1 Version Control Systems

A version control system provides the ability to version resources such as data files. (We use the phrase "a version control system" to mean "an instance of a version control system.") Considerable functionality is associated with modern version control systems; some examples are distributed development, non-linear development and the maintenance of history (versions) [5, 6]. A version control system comprises objects such as repositories and branches that help realize such functionality. An authorization scheme for a version control system specifies how these objects are protected, while respecting the semantics of the relationships between the objects.

$$\begin{aligned}
&\text{Version Control System} ::= \{\text{Repo}\}\\
&\text{Repo} ::= \text{Branch}\,\{\text{Branch}\}\\
&\text{Branch} ::= \{\text{Folder}\}\,\{\text{File}\}\,\{\text{Tag}\}\\
&\text{Folder} ::= \{\text{Folder}\}\,\{\text{File}\}
\end{aligned}$$

Figure 2.1: A version control system in BNF. The notation ::= stands for "comprises," and $\{\cdot\}$ means 0 or more occurrences.

In Figure 2.1, we show the components of a version control system from the standpoint of authorization in Backus-Naur Form (BNF) [27], and give an example below. A version control system is a collection of repositories. A repository can be seen as a collection of data for a single project. A repository comprises branches. A branch is a thread of work on the data in a repository. A repository has at least one branch – its main branch. After a user edits some of the data in a branch, she may merge her changes with another branch.

A branch comprises folders, files and tags. Files contain data, and folders contain files and other folders; they are similar to folders and files in conventional file systems. A tag represents a "commit point" in a branch. Thus, a branch can be seen as comprising tags. From the standpoint of our authorization scheme, there is a function from the set of branches to the set of repositories, the set of tags to the set of branches, and the set of files and folders to the set of branches.

2.2 Authentication and Access Enforcement

Before we discuss the `gitolite` authorization scheme, we describe authentication and access enforcement with `gitolite`. The `gitolite` approach is quite general; however, there are some implementation aspects that are specific to Git. Our main point with this section is to discuss how we have been able to realize `gitolite` as middleware, and largely agnostic to changes to Git. `gitolite` works with version 1.6.2 and later versions of Git "out of the box."

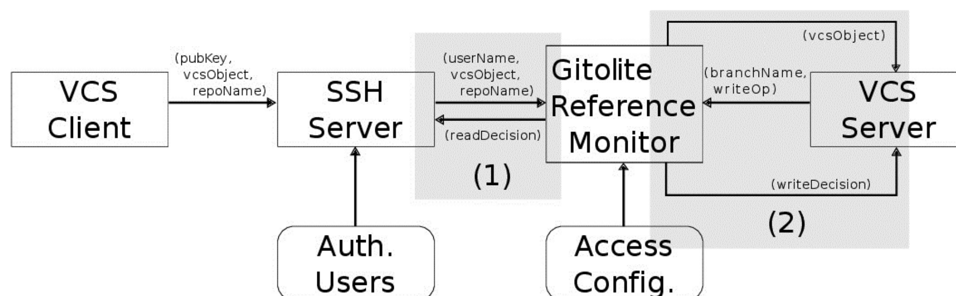

Figure 2.2: Access enforcement to version control system objects using the `gitolite` reference monitor.

In Figure 2.2, we show the relevant entities and the process. Git relies on SSH for access by a client, and authentication. As we show in the figure, four entities are relevant in the context of authentication and access enforcement. One is the version control system client, that issues access requests to objects maintained by the version control system server. The communication is mediated by the SSH server, and access requests are mediated by the `gitolite` Reference Monitor. A version control system client communicates a command that is intended for `gitolite` and Git, that is opaque to the SSH server. The SSH server authenticates the client based on its public key (pubKey in Figure 2.2). The SSH server is configured to map the client's pubKey to its `gitolite` username. (Every Git client has to have a username to be authorized by `gitolite`– see Section 2.3.)

A command from a version control system client comprises two components: the name of a repository (repoName), and what we call a version control system object (vcsObject). The vcsObject is opaque to the SSH server and the `gitolite` Reference Monitor, and must be "unwrapped" by the version control system server. From the standpoint of access enforcement, the vcsObject contains a branchName, the data within the branchName to that the access request pertains, and the manner in that the client wishes to access the data.

The modes of access are classified into two: read and write. There is only one kind of read access; there are several kinds of write access. As Figure 2.2 indicates, access enforcement is split into two stages based on whether the request pertains to a read or write access. We label these (1) and (2) in the figure.

Any write right to a branch of a repository implies read access to the repository. Consequently, in stage (1) of access enforcement, the `gitolite` Reference Monitor first checks whether userName is authorized to read some branch of repoName. If the check succeeds, the access check labelled (1) passes. As we explicate in Section 2.3.3, a client needs to have read access to only any branch in the repository to have read access to the entire repository. Consequently, there is no need for the `gitolite` reference monitor to access any part of vcsObject to make an access decision related to read.

The access check labelled (2) is triggered by the version control system server, if necessary. It is not necessary if the client wishes read access only. The manner in that this is implemented in `gitolite` is by what is called a "hook" in Git. A hook allows one to associate a call-back function with Git. In our case, the hook is used for access enforcement. The version control system server is able to interpret the contents of vcsObject. It invokes the call-back function within `gitolite` with the name of the branch (branchName) and the particular write operation (writeOp) to that the request pertains. The `gitolite` reference monitor consults its policy and issues a binary decision on the access request.

2.3 Authorization Scheme

Our authorization scheme is discretionary [7]. We discuss in Section 2.3.2 how users specify rules. These rules are compiled into an ACL that is used for access enforcement; we discuss this

in Section 2.3.3. An ACL, in our context, is a set of triples, each of the form ⟨User, Resource, Right⟩. It indicates that User possesses the Right to the Resource.

The authorization rules include administrative actions such as delegations. We begin, in the following section, with a discussion of our classification of users into four user-classes, three of that comprise users that are allowed to specify authorization rules that pertain to administration.

2.3.1 Classes of Users

Traditional version control systems have only two classes of users: an administrator, and an accessor. An administrator configures authorization rules, and an accessor is allowed access based on those rules. In `gitolite`, we have four classes of users, as we show in Figure 2.3. These classes and their relationships are fixed. Consequently, the depth of delegation is at most four. This design choice of a fixed delegation depth is similar to some schemes from prior work such as ARBAC97 [28] that has a delegation depth of two, and dissimilar to schemes such as RT [11] that allows delegation of arbitrary depth.

As we express in the figure, our four classes of users are: VCS Admin, Repo Admin, Repo Owner and Accessor. A solid arrow represents "may delegate to." We point out that the transitive closure of that relation is appropriate in our context. For example, we indicate in Figure 2.3 that a VCS Admin may delegate to a Repo Admin. However, he may delegate also to Repo Owner and Accessor.

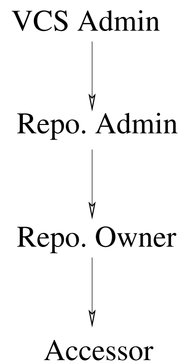

Figure 2.3: We have four classes of users. The transitive closure of the arrows represents "may delegate to."

We give a broad characterization of each user-class here, and make it more precise in the following sections. A VCS Admin administers an entire installation of a version control system. In a typical enterprise, we anticipate that this is a classical systems administrator. We anticipate that a VCS Admin is not involved in projects that are represented by repositories.

A Repo Admin administers a set of repositories. The set typically represents a large project within that may exist smaller projects. The term "within" has no meaning in the context of repositories in a version control system such as Git; there is no such thing as a "sub-repository."

Consequently, each smaller project is associated with a repository, but the authorization scheme must somehow capture the intuition that a set of repositories are administered together. This is why we have designed our scheme to have the notion of a Repo Admin.

A Repo Owner administers a repository. In `gitolite`, we have the notion of the creator of a repository as well. We assume in this dissertation that the Repo Owner is the creator of the repository. A Repo Admin determines who may create (and hence, own) repositories, and delimits the identities of those repositories. Only one user may be the Repo Owner of a repository. An Accessor accesses version control system objects such as repositories and branches.

A typical chronological flow of actions is as follows. A version control system is installed by the VCS Admin. Initially, and over time the VCS Admin adds users that may access parts of the version control system. The VCS Admin also broadly divides the version control system into sets of repositories for administration by Repo Admins. These repositories may not exist yet; the specification is done using regular expressions. A Repo Admin is involved in projects in an intimate way. He designates Repo Owners for sub-projects that have some limited discretion in handing out access rights to other users. We discuss more concrete uses of `gitolite` in Section 2.5.

2.3.2 Administrative and Authorization Specifications

In this section, we discuss the specifications that a user in each class that we present in the previous section may make. The `gitolite` user-classes are organized in a hierarchy (see Figure 2.3). Anything a user at a particular class can do may also be done by a user at a higher class for that instance of the hierarchy. That is, a user at a higher class of the hierarchy may delegate some power directly to a user at a class lower than the one immediately below his.

We call each specification of authorization for an entire version control system a *configuration*, and an element of a configuration a *rule*. For example, the VCS Admin may designate *Alice* to be a valid user. He does so using a rule in the configuration for that version control system.

VCS Admin The following are rules that only a VCS Admin may specify.

- The set of valid usernames for the version control system. This set is the domain to that the SSH Server maps public keys at the time of access (see Section 2.2).
- A set of mnemonics. Examples of such mnemonics are READERS and WRITERS. The intent is for these to be used by a Repo Owner to specify who may read and write objects. A mnemonic may be seen as a group or role. However, there are some differences between what we call a mnemonic, and groups and roles. Indeed, `gitolite` has a separate notion of a group that may be specified in place of a user in an ACL entry.
- Who may act as a Repo Admin. The VCS Admin is able to delimit the set of repositories over that a Repo Admin has purview. He does this by specifying regular expressions for the names of the repositories. (Repositories within a version control system are identified

uniquely by name.) More than one Repo Admin may be associated with a set of repositories. This may happen, for example, if two regular expressions match the same string. The VCS Admin assigns a strict order to a Repo Admin's authorization rules.

From the standpoint of authorization, we need certain conditions to hold with regards to delegations and definitions that the VCS Admin has made. We capture these in the effective rights that users have, that we discuss in Section 2.3.3.

Repo Admin The following are rules that only a Repo Admin or VCS Admin may specify.

- Associate a right with a mnemonic. For example, a Repo Admin may associate the right "read" with the mnemonic READERS, and "read" and "write" with WRITERS.
- Who may act as a Repo Owner, and of that repositories a user may be the Repo Owner. A Repo Admin may also delimit a Repo Owner's ability to give out rights. One manner in that he does this is by designating a repository to be "private." We discuss this below. Another manner is that he can deny a particular user some rights, that a Repo Owner cannot override.

Repo Owner The following are rules and actions that only a Repo Owner or Repo Admin of a repository, or the VCS Admin may specify and perform.

- Create a repository. A repository may be created only if one with the same (fully-resolved) name does not exist, and a corresponding Repo Admin has authorized it.
- Assign users to mnemonics. As we mention above in our discussions on Repo Admins, we constrain a Repo Owner in that he is unable to directly assign rights to users. All he is allowed to do is assign a user to a mnemonic, via that the user may acquire rights. This is a design choice that balances flexibility (expressive power) and security.

2.3.3 Compilation

As we mention in Section 2.3.2, in `gitolite`, rules are specified in a configuration. There is one configuration per version control system; the configuration is split across one or more files. The VCS Admin has a particular configuration file in that he specifies his rules, such as the legitimate usernames and the definitions of mnemonics. Similarly, each Repo Admin has his own configuration file.

EXAMPLE 1 *Let us say that Alice is a VCS Admin and she defines in the repository gitolite-admin:*

```
@1 = p.* q.*
repo    gitolite-admin
        RW+                        = Alice
        RW                         = Bob Mike
        RW+  NAME/                 = Alice
        RW+  NAME/conf/fragments/1 = Mike
        RW+  NAME/conf/fragments/1 = Bob
```

and she knows the following users' public keys and mnemonics:

- *Carl, Bill, Ann and Ted as users,*
- *READERS, WRITERS, CREATOR as mnemonics.*

In a configuration file we use "@" to indicate groups of regular expressions (resources) or users. In this example we have "@1" that is a group of regular expressions "p." and "q.*". In the repository gitolite-admin, we have Alice as VCS Admin in line 3. Line 3 says that Alice has* `rewind` *(RW+) access to gitolite-admin. Bob and Mike as Repo Admins have* `write` *(RW) access to gitolite-admin. In line 5 through 7 Alice is making an order over Repo Admins. Using the above syntax Mike has the priority over Bob on all regular expressions in group "@1". It means that if Mike denies a user to write to any regular expressions (resources) in "@1", the user will not have access, even if Bob has given him access.*

We have the following configuration file as a Repo Admin's (Bob's) configuration file:

```
@students   =   Carl Bill
repo    p.*
    C    =   @students
    -    =   Ted
    RW+ =   CREATOR
    RW   =   WRITERS
    R    =   READERS
```

In Bob's configuration file we have the group "@students" of users. He gives `create` *(C) permission over regular expression "p.*" to the members of the group "@students". Also he denies Ted to have* `write` *(RW) access to "p.*". He assigns some permissions to mnemonics as well.*

Now Carl creates the repository "p1" and becomes the Repo Owner:

```
$ git clone git@server:p1
```

Carl sets the following permissions:

```
$ ssh git@server setperms p1 < myperms
New perms are:
READERS Ann
WRITERS Ted
```

The question here is who has access to repository "p1" ans which kinds of access it has. We try to answer this question with proposing our semantics.

In this section, we discuss how the configurations are compiled into an ACL. Compilation, therefore, translates a configuration into a list of effective rights that each user has to resources, encoded as an ACL. Compilation takes as input a specification of the authorization scheme as we discuss in the previous two sections, and outputs an ACL, that comprises ⟨User, Resource, Right⟩ triples. The reason for this intermediate step of compilation, rather than directly checking a request against the configuration rules is efficiency at the time of access enforcement.

`gitolite` uses Git to administer Git. That is, the configuration files are maintained as part of the Git repository that is the version control system instance. As we mention in Section 2.3.2, an aspect of `gitolite` is the prioritization of Repo Admins and rules within a Repo Admin's specification. Our prioritization is implemented by sequencing the configuration files in a particular order during compilation. This is simple, but effective in achieving what we need, that is imposing a strict priority on Repo Admins and rules respectively. All a VCS Admin does, for instance, is specify in what order the `gitolite` compiler should process the configuration files, and this expresses his prioritization, as we see in Example 1.

For User to have Right over Resource, the following two conditions are necessary.

1. The User must be valid, as specified by VCS Admin, and,
2. The user must either (a) directly be assigned Right (by Repo Admin), or, (b) there must exist a valid mnemonic (as specified by the VCS Admin) to that User is assigned (by the Repo Owner), and that mnemonic must have in its effective set of rights, Right.

Consequently, the first set of processes that the compiler runs comprises assembling the following from the configuration rules.

3. The set of valid users.

4. The set of valid mnemonics.
5. The set of authorizations of rights directly assigned to users, and the corresponding priority.
6. The set of authorizations of rights assigned to mnemonics and the corresponding priority.
7. The set of user to mnemonic assignments per resource.

The pieces of information (3)–(7) are read by the compiler directly from the configuration rules. The compiler must then process the above information and make inferences regarding the effective rights per user to resources. This comprises the following steps.

8. Infer the rights a user may have to resources via mnemonics. This involves putting the information from Steps (6) and (7) together. The compiler also records the priority with that each such assignment is associated.
9. For each right to that a user is authorized, either directly or indirectly via a mnemonic, infer other rights. We discuss below how this inference is done. An example of such an inference is that any (positive) right to a resource implies read access to the repository that corresponds to the resource.
10. Decide whether the user indeed has a right to a resource by considering the highest priority rule that grants him the right. If there is a "deny" counterpart of this right in a rule of higher priority, then the user does not have the right.

After Steps (8)–(10) are completed, the compiler is able to output ACL entries of the form ⟨Resource, Right, User⟩ for each Right and Resource to that the User is authorized. The final ACL comprises only those positive rights that a user possesses. Consequently, checking whether a user has a right or not against the ACL is efficient.

In Step (9), we refer to inferring that a user has a right to a resource if he has another right to a resource. Following is our specification.

a. If a user has any positive right to a resource, then he has `read` access to the repository associated with that resource.
b. Every right other than `read` is of type write. If a user has any right of type write to a resource, then he has `write` access to that resource.

The reason that underlies the above inferences regarding rights is that it does not make sense to possess a right and not another in the specific cases that we address with the inferences. For example, it does not make sense for a user to have any right of type write to a resource unless he is able to read that resource. Also, in practice, there is no need for the `read` access to be at a finer granularity than to an entire repository.

2.4 Semantics

In this section we consider the semantics of `gitolite` more precisely. In the following example we consider one of the issues we captured in `gitolite` [3, 4] that motivates us to specify a precise semantics for /gol.

EXAMPLE 2 `deny` *permission does not affect* `create` *permission but affects* `write` *permission*

*Assume that a **Repo Owner** creates a repository but he cannot update the repository, because of a* `deny` *permission from a **Repo Admin**. It is reasonable that a* `create` *permission cannot be independent from a* `write` *permission, because when the user is the owner of a repository and he has created it, so he should be able to modify it. We have this observation in **gitolite-admin** configuration file as well. When a user creates a repository, we give the creator* `rewind` *permission. So implicitly* `create` *permission results in* `write` *permission. If we assume the statement* `create` $\rightarrow$ `write`*, which is reasonable, we easily conclude:*

$$(\texttt{create} \rightarrow \texttt{write}) \leftrightarrow (\neg\texttt{write} \rightarrow \neg\texttt{create})$$

When a `deny` *permission affects* `write` *permission, reasonably it should affect* `create` *permission, but it does not.*

We use term rewriting rules in a Term Rewriting System (TRS) [29] to produce the ACL. We consider only those environments in that variables are mapped meaningfully to concrete values that are meaningful.

Definition 1 (A Semantics for an Authorization System) *A semantics provides a 'yes' or 'no' answer to the question, "Does user, u, have permission, p, over the resource r?"*

Definition 2 (The ACL) *An ACL is a set of triples $\langle u, r, p\rangle$, such that for any resource (regular expression) ρ, there exists at most one ACL entry for user u, $\langle u, r, p\rangle$ such that $\rho \subseteq r$. Also for any pair of $\langle u, p\rangle$ there exists at most one resource r and for any pair of $\langle u, r\rangle$ there exists at most one permission p.*

We consider a regular expression as a set of strings. So by $\rho \subseteq r$, we mean that ρ as a set of strings is a subset of r as a set of strings.

Definition 3 (Semantics of an ACL) *Given $\langle u, r, p\rangle$, user, u, has permission, p, over the resource r if and only if $\langle u, r, p\rangle$ matches an entry in ACL.*

2.4.1 A Term Rewriting System (TRS) representing `gitolite`

A Term Rewriting System (TRS) is a pair (Σ, R) of an alphabet or signature Σ and a set of rewrite rules R. The alphabet Σ in `gitolite` consists of:

- A_p: The power set of permissions.
- A_r: The set of all resources. A resource is a 3-tuple $\langle r, b, t \rangle$, where:
  - r is a name of a repository that can be a regular expression.
  - b is a name of a branch that can be a refex, where a refex is a regular expression but a refex is prefix-matched or an empty set.
  - t is a name of a tag that can be a refex or an empty set.
- A_u: The set of all users defined by VCS Admin.
- A_m: The set of all mnemonics defined by VCS Admin.

Definition 4 $K = A_m \cup A_u$ *denotes the set of all entries that can delegate authority to a mnemonic or a user.*

We define two different kinds of rules in `gitolite`:

Definition 5 (Delegation Rules (DR)) *A delegation rule is a definition of a user or a mnemonic $\in K$ over a specified repository $\in A_r$, where the definition belongs to the relation DR: $K \times A_r \cup \{\epsilon\} \rightarrow K \times A_r \times A_p \cup \{\epsilon\}$. When we use ϵ we mean that we are not specifying any kind of resources or permissions.*

Definition 6 (Name Rewrite Rules (NRR)) *A name rewrite rule is a definition of a group of users $\subseteq A_u$ or a group of repositories $\subseteq A_r$, where the definition belongs to the relation NRR: $A_u \rightarrow A_u$ or $A_r \rightarrow A_r$.*

There is no need of closed set of rewrite rules to define the TRS (For example in [29] rules are defined based on the certificates in a string rewriting system). As we will see we can define related ruleset based on different configuration files in different levels of delegation.

2.4.2 Producing the ACL using rewrite rules

In this section, we use rewrite rules to produce the ACL.

A rewrite ruleset R representing configuration files in Example 1 is:

1. From VCS Admin's configuration file we have:

   1. Alice $\mapsto$ Bob p.*
   2. Alice $\mapsto$ Bob q.*

   that means that Alice as the VCS Admin is delegating the administration of repositories p.* and q.* to Bob. We do not consider permissions in this case, because we assume that Bob has full permissions to the repositories of that he is administrator.

2. In Repo Admin's configuration file, Bob as a Repo Admin is assigning permissions to users and mnemonics:

   1. Bob p.* $\mapsto$ Carl p.* C,
   2. Bob p.* $\mapsto$ Bill p.* C,
   3. Bob p.* $\mapsto$ Ted p.* -,
   4. Bob p.* $\mapsto$ CREATOR p.* RW+,
   5. Bob p.* $\mapsto$ WRITERS p.* RW,
   6. Bob p.* $\mapsto$ READERS p.* R.

3. In Repo Owner's myperms file, the owner is assigning each user to a mnemonic and she is the creator of the repository. So we have:

   1. CREATOR p1 $\mapsto$ Carl p1,
   2. READERS p1 $\mapsto$ Ann p1,
   3. WRITERS p1 $\mapsto$ Ted p1.

Now using rewrite rules, we produce all rules having Alice as VCS Admin in the left side. As an example, by concatenating rewrite rules 1.1, 2.4 and 3.1, we can produce: **Alice $\mapsto$ Carl p1 RW+**

After checking all possible concatenations of rules and producing new rules, we have the following in that Alice is in the left side:

1.Alice $\mapsto$ Bob p.*
2.Alice $\mapsto$ Carl p.* C
3.Alice $\mapsto$ Bill p.* C
4.Alice $\mapsto$ Ted p1 -
5.Alice $\mapsto$ Carl p1 RW+
6.Alice $\mapsto$ Ann p1 R
7.Alice $\mapsto$ Ted p1 RW
8.Alice $\mapsto$ Bob q.*

Definition 7 *Authorized Rule*

A rule that:

- *has VCS Admin in the left side,*
- *does not have mnemonics in right side,*
- *does not have any kinds of groups (Using NRRs we can rewrite groups of repositories and users).*

We show an authorized rule as

r: VCS Admin $\mapsto$ ***user*** $\in A_u$ ***resource*** $\in A_r$ ***permission*** $\in A_p$.

Remark.

From [3] we have: "when using deny rules, the order of your rules starts to matter, where earlier it did not. If you're just starting to add a deny rule to an existing ruleset, it's a good idea to review the entire ruleset once, to make sure you're doing it right."

It means that order of rules (configuration file lines) is not important when we do not have deny rules. But in general case we keep the order of lines in configuration files to be sure that in case of deny rules we have an ordered list.

2.4.3 Authorization Priority Factor (APF)

What should we do when we have two rules from two different authorized rules contradict? When does this happen?

It happens when we have a contradiction between two rules that are issued by two different Repo Admins. By contradiction we mean that one rule gives a user a write access but the other one forbids the user from writing. It cannot happen for two different Repo Owner, because the intersection of the repositories that they own is empty set. We have a priority function over Repo Admins in `gitolite`.

Definition 8 (Authorization Priority Factor (APF)) *The function $APF(r)$ is the strict priority that the authorized rule r has. When r is not an authorized rule, $APF(r) = 0$. An authorized rule r, that is produced from other rules, has $APF(r) = APF(r')$, where r' is the rule deducted from gitolite-admin configuration file and is first rule in the chain of rules that produce rule r.*

Sometimes rules contradict and because of that we define some criteria to solve the contradiction s. So we should recognize contradictions and solve them before producing the ACL.

2.4.4 TRS Algorithm

Two regular expressions can be different from one another, and have a non-empty intersection. We assume that ACL entries are being produced one by one from the produced authorized rules from the TRS presenting `gitolite` based on the APF. Whenever an ACL entry is being produced, we check whether there is a contradiction or not. The time complexity is $1+2+3+...+n = O(n^2)$ where n is the number of the ACL entries. Because in `gitolite` we are processing configuration files based on APF, the newer entry should not contradict with previous ones. If it does, we change rules to get new rules (and sometimes previous rules) that do not contradict with previous rules and finally produce the ACL entry. But the main problem is how can we check and solve the contradictions. We check the contradiction between two rules, such that one of them has been considered before and the other one is ready to be considered and be inserted to the ruleset. From definition 7, we know that an authorized rule is:

r: VCS Admin $\mapsto \quad u_r \in A_u \quad reg_r \in A_r \quad p_r \in A_p.$

To make it easier to work with, we consider the right side of an authorized rule as a three tuple authorized rule and we do not mention the VCS Admin name in left side.

Definition 9 (ACL that corresponds to `gitolite` rulesets) *We say "A" corresponds to the* `gitolite` *ruleset R if and only if:*

1. *"A" is an ACL,*
2. *For any regular expression ρ, such that there is no regular expression $\rho' \subset \rho$, for each $\langle u, reg_j, p_j \rangle$ in ACL such that $\rho \subseteq reg_j$, there exist entries $\langle u, reg_i, p_i \rangle$ in R such that $\rho \subseteq reg_i$ and $p_j = \cup p_i$.*
3. $\bigcup_R reg_i = \bigcup_{ACL} reg_j$

Lemma 1 *The ACL that corresponds to a* `gitolite` *ruleset R is unique.*

PROOF.

Assume by way of contradiction that we have at least two different access control lists, ACL_1 and ACL_2 correspond to the `gitolite` ruleset R. There exists at least one entry in ACL_1, $\langle u, reg_1, p_1 \rangle$, that is not in ACL_2. From Definition 9, there should be an entry $\langle u, reg_2, p_2 \rangle$ in ACL_2 such that $reg_1 \cap reg_2 \neq \emptyset$. There can be two cases:

1. $reg_1 = reg_2$, but $p_1 \neq p_2$. From Definition 9, if we consider ACL_1 we conclude that $p_1 = \cup p_i$, where for all ρ, such that there is no $\rho' \subset \rho$, $\rho \subseteq reg_1$, there exist entries of the form $\langle u, reg_i, p_i \rangle$ such that $\rho \subseteq reg_i$. But because $reg_1 = reg_2$ then we have same rules from R where $p_2 = \cup p_i$, then $p_1 = p_2$.

2. $reg_1 \neq reg_2$. Without loss of generality assume that $reg_1 \backslash reg_2 \neq \emptyset$ and $\rho \subseteq reg_1 \backslash reg_2$. Let say that $\rho \subseteq reg_3$ in ACL_2 and the related entry is $\langle u, reg_3, p_3 \rangle$. From Definition 9 we have $p_1 = p_3$. Now choose ρ' in Definition 9 from $reg_1 \cap reg_2$, again we conclude that $p_1 = p_2$. So $p_2 = p_3$, but it contradicts, because from Definition 2 we know that for a pair of a user and a permission we cannot have more than one regular expression (resource).

□

Definition 10 (Confluent Term Rewriting System) *We have $\rightarrow^*$ as a reflexive transitive closure of relation $\rightarrow$ [30]. Now consider the rule $s \rightarrow^* d$. We say that s is confluent if and only if with any terms t_1 and t_2 such that $s \rightarrow^* t_1$ and $s \rightarrow^* t_2$ imply $t_1 \rightarrow^* d$ and $t_2 \rightarrow^* d$. A term rewriting system is confluent if and only if every term in the system is confluent.*

Corollary 1 *Using Lemma 1, our term rewriting system is confluent.*

In our TRS algorithm we need to recognize contradictions may happen between two rules r_i and r_j in R. We propose the following algorithm as one part of TRS algorithm to do that. Assume that $R = \{r_1, ..., r_k\}$ is a ruleset that is the list of authorized rules and is sorted in increasing order based on APF. Another assumption we make is that R only includes the authorized rules for a specific user u. The intuition behind this assumption is from the cause of a contradiction. Contradiction never happens when we have two authorized rules with two different users. One user assumption is not making the problem easier to solve.

The following algorithm is used to solve the contradiction between two rules r_i and r_j in the ruleset R.

MATCHING-FUNCTION(R, r_i, r_j)
    if $\{-\} \subseteq p_{r_i}$ **then**
        $p_{r_{k+1}} = (p_{r_i} \cup p_{r_j}) \backslash \{W\}$
    else
        $p_{r_{k+1}} = (p_{r_i} \cup p_{r_j})$
    $r_{k+1} = \left\langle u_{r_i}, reg_{r_i} \cap reg_{r_j}, p_{r_{k+1}} \right\rangle$
    $reg_{r_i} = reg_{r_i} \backslash reg_{r_j}$ **and** $reg_{r_j} = reg_{r_j} \backslash reg_{r_i}$
    return $R = R \cup \{r_{k+1}\}$

Where $\{-\}$ is the `deny` permission and $\{W\}$ is the `write` permission.

Lemma 2 *MATCHING-FUNCTION is sound and complete.*

PROOF.

In MATCHING-FUNCTION we should have no contradiction between output rules. It means we should have three rules in the output with mutually exclusive resources. Because deny permission makes some constraints on other permissions, so there are two different cases. Function ensures that the rule with intersection of reg_{r_i} and reg_{r_j} as its resource, does not have `write` permission, when r_i that has higher priority has `deny` permission. After that function makes a union of permissions of input rules for new rule. The function is complete because it has a case analysis over permissions and resources.

□

Now we propose an algorithm for keeping track of all permissions in the ruleset R. In PERMISSIONS-SET, for each subset of power set of permissions we define a new rule that has all the resources with the chosen permission and remove rules with the chosen permission from the ruleset. The intuition behind this algorithm is bounding the number of rules in the ruleset R to the number of permissions we have in R. With this bound we guarantee that with a bounded number of permissions, Our TRS algorithm works in polynomial.

PERMISSIONS-SET(R)
  foreach $p \in A_p$ **do**
    $reg_p = \emptyset$
    foreach $r_i \in R$ **do**
      if $p_{r_i} = p$ **then**
        $reg_p = reg_p \cup reg_{r_i}$
        $R = R \backslash r_i$
    if $reg_p \neq \emptyset$ **then**
      $R = R \cup \{r_P = \big(u, reg_p, p\big)\}$

Now we propose the algorithm CONTRADICTION-SOLUTION-GITOLITE that uses MATCHING-FUNCTION and PERMISSIONS-SET to solve all contradictions in the ruleset R.

CONTRADICTION-SOLUTION-GITOLITE(R)
  $S = \{r_1\}$;
  for $j = 2$ *to* k **do**
    $S = S \cup r_j$;
    for $i = 1$ *to* $|S|$ **do**
      MATCHING-FUNCTION(S, r_i, r_j);
    PERMISSIONS-SET(S);
  return $R = S$

In this algorithm we return the ruleset R that does not have any resources overlap (contradiction). It also has other specifications that ACL entries should have.

Proposition 1 *Contradiction-Solution-Gitolite is sound.*

Proof.

To show that Contradiction-Solution-Gitolite is sound, we need to show that the output of Contradiction-Solution-Gitolite(R) is an ACL and because the ACL is unique from Lemma 1, the output is the ACL corresponding to `gitolite` rulesets R. We show by induction on number of rules in R that S as the output of Permissions-Set(S) in line 6 of Contradiction-Solution-Gitolite is the ACL for $R = \{r_1, ..., r_j\}$. The base case is trivial, because we have only one rule, we have the ACL with one entry. Now assume that S is the ACL for $R = \{r_1, ..., r_{k'}\}$, we show that S in next iteration of the for loop is the ACL for $R = \{r_1, ..., r_{k'+1}\}$. In $k' + 1$st iteration Matching-Function($S, r_i, r_{k'+1}$) guarantees that resource of $r_{k'+1}$ has no intersection with previous resources in S. Meanwhile Matching-Function($S, r_i, r_{k'+1}$) does not remove any resource from S and new rule $r_{k'+1}$, so union of resources in updated S will be union of resources in S and $r_{k'+1}$. Also Permissions-Set(S) guarantees that for a pair of $\langle u, p\rangle$, we have only one rule in the ACL.

□

Proposition 2 *Contradiction-Solution-Gitolite is complete.*

Proof.

Permissions-Set is complete, because we assume that A_p and R are two finite sets then Permissions-Set terminates for the input set R. In line 3 of Contradiction-Solution-Gitolite we add a rule from R to the set S. For loop in lines 2-6 terminates, because Matching-Function and Permissions-Set are complete functions, and consequently Contradiction-Solution-Gitolite is complete.

□

Contradiction-Solution-Gitolite Computational Complexity

In the worst case, we have a ruleset that has mutually non-empty intersection resources and none of them has same permission. This case is the worst case because when we call Matching-Function, the number of rules in R in the output of the Matching-Function is added by one.

- Lines 4-5: Each time that we want to check a new rule with existing rules, the number of rules doubles up. In lines 4-5 in Contradiction-Solution-Gitolite, the algorithm is replacing the set S by a new one with size at most $2 \cdot |S| - 1$. Assume each time the algorithm wants to check the intersection between two regular expressions (repositories), it is doing that in time I. Consequently the running time is $(2 \cdot |S| - 1) \cdot I$.

- Line 6: In line 6 we have the subroutine Permissions-Set that is unifying all rules which have same permission in A_p. So the running time of the subroutine Permissions-Set is $(2 \cdot |S| - 1) \cdot |A_p|$.
- Line 2-6: We have a for loop over all elements in R. When $j = 2$, then $|S| = 2$ and the running time is $(2 \cdot 2 - 1) \cdot I + (2 \cdot 2 - 1) \cdot |A_p|$. Then in lth iteration of for loop the running time is roughly $2 \cdot 2^l \cdot I + 2 \cdot 2^l \cdot |A_p|$

In the algorithm Contradiction-Solution-Gitolite any of the following cases can happen:

- $|A_p| > 2^k$: Then the total running time is: $\Sigma_{j=2}^{j=k} 2^j \cdot I + 2^j \cdot |A_p| = (2^{k+1} - 1) \cdot I + (2^{k+1} - 1) \cdot |A_p| \simeq 2^{k+1} \cdot (I + |A_p|)$.
- $|A_p| \leq 2^k$: Assume that $2^m < |A_p| \leq 2^{m+1}$, then the total running time is: $\Sigma_{j=2}^{j=m} 2^j \cdot I + 2^j \cdot |A_p| + (|A_p| \cdot I + |A_p|^2) \cdot (k - m) \simeq 2^{m+1} \cdot (I + |A_p|) + (|A_p| \cdot I + |A_p|^2) \cdot (k - m)$. It happens because when the number of the rules in S is not less then $|A_p|$, the algorithm does not double up the number of rules anymore.

Now to capture both above mentioned cases we define the following slope function: $r(n) = n$ when $n > 0$ and $r(n) = 0$ when $n < 0$. We can assign $m = \lfloor \log |A_p| \rfloor$. Also by $min(.,.)$ we mean the minimum number between two inputs of the function. Consequently the time complexity of Contradiction-Solution-Gitolite is:

$$T(k, |A_p|, I) = O(2^{min(k, \lfloor \log |A_p| \rfloor)+1} \cdot (I + |A_p|) + (|A_p| \cdot I + |A_p|^2) \cdot r(k - \lfloor \log |A_p| \rfloor))$$

Proposition 3 *Generating the ACL that corresponds to* `gitolite` *ruleset is in* ***EXPTIME*** ^{O_C,O_I}*, where* O_C *is an oracle that decides whether one regular expression is a subset of another and* O_I *is an oracle that decides whether two given regular expressions have non-empty intersection.*

Proof. The reason we assume two oracles O_C and O_I is that in [31] it has been shown that deciding that one regular expression is a subset of another and deciding whether two given regular expressions have non-empty intersection are **EXPTIME-complete**. The algorithm Contradiction-Solution(R) produces an ACL in exponential time in the size of input in the worst case. From Lemma 1 the produced ACL is the corresponding ACL to the ruleset R.

□

The *ACL generation problem* is the problem of generating the ACL that corresponds to a given `gitolite` ruleset. In other words, it computes the (one-to-one) function $g \colon \mathcal{R} \longrightarrow \mathcal{A}$, where $\mathcal{R}$ is the set of all `gitolite` rulesets and $\mathcal{A}$ is the set of all ACLs. We now consider a decision problem that is a lower-bound in computational complexity to g.

Definition 11 *CORR-ACL = $\{\langle R, n, p \rangle : R$ is a `gitolite` ruleset, and $n, p \in \mathbb{Z}^+$, such that there exists an ACL that corresponds to R that has fewer than n entries each of whose permission sets is of size $p\}$.*

Proposition 4 *If g can be computed in polynomial-time, then CORR-ACL can be decided in polynomial-time.*

PROOF.

Given $\langle R, n, p \rangle$, we invoke the algorithm that computes g with input R. The output A must be polynomial in R. We now count the number of entries in A that have permission sets of size p and check that it is at most n. □

Proposition 4 establishes that the computational complexity of deciding CORR-ACL is a lower-bound for the computational complexity of computing g.

Proposition 5 *CORR-ACL $\in$ **EXPTIME**.*

PROOF.

The algorithm CONTRADICTION-SOLUTION(R) produces an ACL in exponential time in the size of input in the worst case. Also we have a cook reduction from CORR-ACL to the generating the ACL problem. The cook reduction is: When we can generate the ACL, we can extract the size of the ACL and then solve CORR-ACL.

□

Proposition 6 *CORR-ACL $\in$ **NP**-hard.*

PROOF.

To prove the proposition, we Karp-reduce the following version of VERTEX-COVER to CORR-ACL.

VERTEX-COVER $= \{\langle G, k \rangle : G$ is an undirected graph and $k \in \mathbb{Z}^+$, such that $k \geq 2$ and G has a vertex cover of size $k\}$. Our restriction that $k \geq 2$ does not affect computational complexity; i.e., VERTEX-COVER $\in$ **NP**-complete.

Let $G = \langle V, E \rangle$ with $V = \{v_1, \ldots, v_n\}$. Corresponding to each edge, $\langle v_i, v_j \rangle \in E$, we associate a resource $reg_{i,j}$. We specify that the $reg_{i,j}$'s are distinct from one another.

With each vertex v_i, we associate a permission $p_i \in P$. We then specify the `gitolite` ruleset R as follows. For each vertex $v_i \in V$, let $\{e_{i,1}, \ldots, e_{i,l}\}$ be the set of edges each of that is not incident on v_i. We specify that R contains entries of the form $\langle u, \{reg_{i,1}, \ldots, reg_{i,l}\}, \{p_i\} \rangle$. We specify $n = \binom{|V|}{k}$ and $p = k$. We now assert that $\langle G, k \rangle \in$ VERTEX-COVER if and only if $\langle R, n, p \rangle \in$ CORR-ACL.

For the "only if" direction, assume that $\langle G, k \rangle \in$ VERTEX-COVER. This means that there exists a set of $k \geq 2$ vertices that covers all the edges in G. Let $\{v_1, \ldots, v_k\}$ be such a set. For each v_i, let $reg_i = \left\{reg_{i,1}, \ldots, reg_{i,l}\right\}$ that is the set of resources that corresponds to the entry for v_i in R. We first observe that $reg_1 \cap \ldots \cap reg_k = \emptyset$. The reason is that $v_1, \ldots, v_k$ cover all edges, and each reg_i corresponds to the edges not covered by v_i. $reg_1 \cap \ldots \cap reg_k$ identifies the set of edges not covered by $v_1, \ldots, v_k$ together, that is empty.

The set of permissions in the ACL A that would correspond to the resource-set $reg_1 \cap \ldots \cap reg_k$ is $\{p_1, \ldots, p_k\}$. As the resource-set is empty, we know that such an entry is not in A. Therefore, the number of entries in A with a k-sized set of permissions has to be fewer than $n = \binom{|V|}{k}$.

For the "if" direction, consider the case that $\langle G, k \rangle \notin$ VERTEX-COVER. Then, given any k-sized subset of V, at least one edge is not covered by it. Let $\{v_1, \ldots, v_k\} \subset V$. Then, for the corresponding resource-sets in R, $reg_1 \cap \ldots \cap reg_k \neq \emptyset$, and the ACL A has to contain an entry $\langle \{reg_1 \cap \ldots \cap reg_k\}, \{p_1, \ldots, p_k\} \rangle$. And we would have exactly $n = \binom{|V|}{k}$ such entries in A whose permission-sets are of size $p = k$. Thus, $\langle R, n, p \rangle \notin$ CORR-ACL. □

Parameterized Computational Complexity

We commence this section by some examples.

EXAMPLE 3 *Assume that we have* $A_p = \{\{\texttt{read}\}, \{\texttt{read}, \texttt{write}\}\}$. *As you can see in this case we never have more than two choices for permission of rules.*

EXAMPLE 4 *Now assume that we have:*

- $A_p = \{\{\}, \{\texttt{read}, \texttt{write}\}, \{\texttt{write}\}\}$,
- $r_1 = (Alice, reg_1, \{\{\texttt{read}\}\})$ *and* $r_2 = (Alice, reg_2, \{\{\texttt{write}\}\})$ *where two regular expressions* reg_1 *and* reg_2 *has intersection* reg_{12}.
- *Due to having the worst case situation, we assume that all regular expressions of new rules have intersections with all regular expressions and previous intersections of regular expressions. For example* $r_3 = (Alice, reg_3, \{\texttt{read}\})$ *has resource* reg_3 *that has intersections with* reg_1, reg_2 *and* reg_{12}.

Now whatever you want to add to the list of authorized rules, you will add at most half of the number of existing processed rules. It is easy to show that the time complexity of proposed solution of contradiction algorithm is quadratic in the number of authorized rules.

From the above examples we have a clue such that making $|A_p|$ bounded by a constant number will effect the computational complexity of the problem. Now we define the following language.

Definition 12 *Let* CORR-ACL$_p$ $= \{\langle R, n\rangle : R$ *is a* `gitolite` *ruleset,* $n \in \mathbb{Z}^+$, $\exists$ *ACL of* n *rules that corresponds* R *with bounded* $|A_p|\}$.

Proposition 7 CORR-ACL$_p$ $\in \boldsymbol{P}^{O_C,O_I}$, *where* O_C *is an oracle that decides whether one regular expression is a subset of another and* O_I *is an oracle that decides whether two given regular expressions have non-empty intersection.*

PROOF.

From CONTRADICTION-SOLUTION-GITOLITE Computational Complexity we have:

$$T(k, |A_p|, I) = O(2^{min(k,\lfloor\log|A_p|\rfloor)+1} \cdot (I + |A_p|) + (|A_p| \cdot I + |A_p|^2) \cdot r(k - \lfloor\log|A_p|\rfloor))$$

By having the two oracles O_I and O_C we assume that I is a constant number:

$$T(k, |A_p|) = O(2^{min(k,\lfloor\log|A_p|\rfloor)+1} \cdot |A_p| + |A_p|^2 \cdot r(k - \lfloor\log|A_p|\rfloor))$$

As we said before there are two cases for $|A_p|$ comparing to k.

- $|A_p| > 2^k : T(k, |A_p|) = O(2^{k+1} \cdot |A_p|) = O(|A_p|^2)$
- $|A_p| \leq 2^k : T(k, |A_p|) = O(2^{\lfloor\log|A_p|\rfloor+1} \cdot |A_p| + |A_p|^2 \cdot r(k - \lfloor\log|A_p|\rfloor)) = O(k \cdot |A_p|^2)$

So the running time of the algorithm with two assumptions: 1) $|A_p|$ bounded by a constant number and 2) having two oracles O_C and O_I, is the following:

$$T(k) = O(k \cdot |A_p|^2)$$

Using PERMISSIONS-SET, we guarantee that size of the ruleset R in `gitolite` never exceeds $|A_p|$. It implies that the time complexity of proposed solution of contradiction algorithm is quadratic in the number of authorized rules in R.

□

TRS Algorithm

An ACL has an important role in access control. In previous sections we give an algorithm to compute an ACL from a configuration file "c" that is presenting a `gitolite` system of authorization. We need an enforcement algorithm that has an ACL and a request as its input and the response that allows the user (issuer) of the request to access the resource that he asked or disallows him as its output. The whole scheme has two algorithms as we see in 2.4.

Now we present TRS algorithm and prove its soundness and completeness. To prove that the TRS algorithm is sound and complete we show that in each step of computing the ACL, we

are using some mappings that result in a sound and complete algorithm. After that based on definition of soundness and completeness we show that the enforcement algorithm is sound and complete.

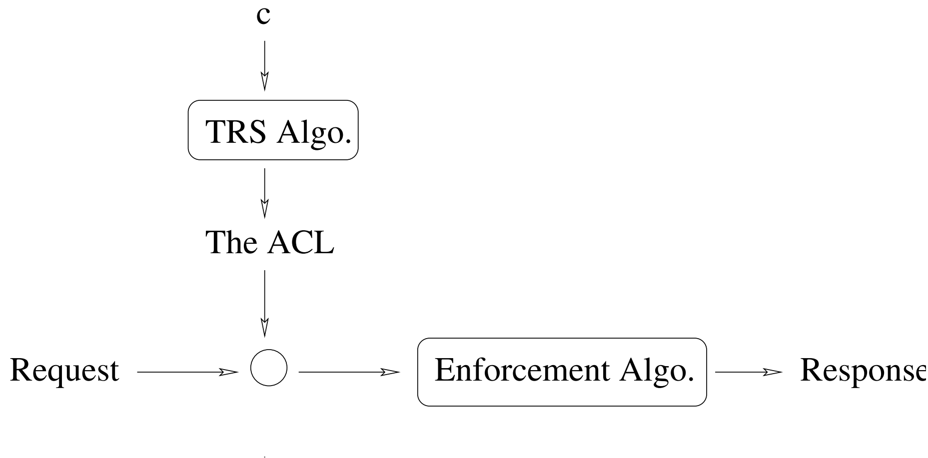

Figure 2.4: The Scheme

TRS algorithm is a function

We divide the whole TRS algorithm into three consecutive steps of mappings:

$$TRSalgorithm(c) = m_3 \circ m_2 \circ m_1(c)$$

As we see in Figure 2.4 we assume that the input of the TRS algorithm is a `gitolite` configuration file and the output is the ACL. The three mappings are as follow:

1. $m_1(.)$: A configuration file $\rightarrow$ Sorted list of rules: In first mapping we syntactically translate a configuration file to a sorted list of rules. By sorted list we mean that the order of inserting the rules to the list matters and we do not change it. We categorize all lines in a configuration file into three different groups:

    - A repo line in that we mention a name of a repository in the configuration file. We do not have an independent rule related to this kind of lines in configuration file, but it has an important role in specifying the resources of the following rules that are being extracted from the following lines in the configuration file. Actually it determines that the resource a user will have access or will not have access. More precisely we can say whenever a line has the word "repo", it is going to determine the resource of upcoming rules in the list of rules till getting next repo line in the configuration file.
    - A definition line in that we define a group of resources or a group of users. The translation of these kinds of lines are Name Rewrite Rules (NRR). More precisely we say whenever a line has the sign "@", it is translated as a NRR.

- A delegation line in that a user, a mnemonic or a group of users are being assigned to a resource and a permission. The translation of these kinds of lines is Delegation Rules (DR). More precisely we say whenever a line has the sign "=", it is translated as a DR.

2. $m_2(.)$: Sorted list of rules $\rightarrow$ Sorted list of authorized rules (ruleset): Using this mapping we want to produce a sorted list of authorized rules in that we do not have any groups, so we do not have any NRR. The purpose of having authorized rules in this step is having rules that are certified by checking the issuers. We mean that firstly we do not consider groups and we want to have a list of rules for users. Secondly we do not consider the rules that are not authorized by VCS Admin implicitly or explicitly. We have the two following steps:

   - We replace all groups in DR rules using NRR rules by individual users. In many cases we should replace a DR rule by many, because reasonably the number of members in a group is more than one. To have the new list we remove NRR rules as well, because we do not need them anymore.
   - In this step we replace each rule in the list by its related authorized rule. By related authorized rule we mean the authorized rule such that the last rule in it, is the assumed rule. At the end of this step we have a sorted list of authorized rules.

3. $m_3(.)$: Sorted list of authorized rules $\rightarrow$ The ACL: In this part we apply CONTRADICTION-SOLUTION-GITOLITE to the sorted list of authorized rules (ruleset). As we know that CONTRADICTION-SOLUTION-GITOLITE is sound and complete, and there is no indeterminism in the algorithm, we conclude that we can consider m_3 as a mapping.

We showed that the TRS algorithm can be presented as a three level mappings composition.

Theorem 5 *The TRS algorithm computes the function $f(.) : C \rightarrow A$, where C is the domain of all `gitolite` configuration files and A is the domain of ACL based on the definition 2.*

PROOF.

Assume by way of contradiction that $f(.)$ is not a function. So for a configuration file in different time we should have different ACL. The important factor of the TRS algorithm is being time invariant. We do not have any kind of randomness in the three steps of constructing the TRS algorithm. Because in each step we have just a mapping from a sorted input to a sorted output, there is no choice in doing the mappings. So we conclude that it contradicts and we cannot have two different outputs for same input.

□

Enforcement algorithm is sound and complete

In this section we show that the enforcement algorithm is sound and complete. As we see in 2.4 inputs of enforcement algorithm are a request and an ACL.

Definition 13 (A request) *A request is a three tuple vector of a name of a user, a name of an existing resource and a permission.*

Now we define completeness as it follows:

Definition 14 (Completeness) *An enforcement algorithm is complete when it terminates and outputs either true or false for a given input.*

Theorem 6 *The enforcement algorithm is complete.*

PROOF.

There are only two different cases in the enforcement algorithm:

- Whenever the request matches no or more than one entry in the ACL, the enforcement algorithm outputs false.
- When the request matches one entry the output is true.

So we can easily conclude that the algorithm is complete.

□

Definition 15 (Soundness) *What we mean by soundness is that a request matches at most one ACL entry.*

Theorem 7 *The enforcement algorithm is sound.*

PROOF.

There is at most one entry in the ACL that potentially matches the request. Assume by way of contradiction that there are two entry in the ACL that have the user and the resource same as the request. So there should be two authorized rules in the procedure of the producing the ACL with non-empty intersection of their resources.But it contradicts, because we assume that the ACL entries are CONTRADICTION-SOLUTION-GITOLITE outputs where they do not have resources with non-empty intersection. □

2.5 Experience from Deployments

In this section we test the performance of the proposed algorithm, TRS, in real world. We name the implementation of our algorithm an gitoliteACL , because the result of our algorithm is an ACL. Also we compare gitoliteACL against the release version 2.0 of `gitolite` [3] that is already being used. Henceforce, we call this release as gitolite .

2.5.1 Experimental Setup

We perform our experimental evaluation on a machine that has the following setup: Intel(R) Pentium(R) 4 CPU 3.00GHz with 3GB RAM, Ubuntu Linux 12.04 with kernel version 3.2.0-49. Both gitoliteACL and gitolite use the PERL programming language for its implementation.

All our experiments use standard read and write access requests. We only report the results of write access requests. This is because denying accesses to a user in gitolite strictly apply to write permissions. To minimize the impact of network latency, we ensure that the access requests originate from the same server hosting the git server. Thus, measuring the time an access request takes does not include the time to transfer the data. Instead, we only account for the time taken to perform the access check. That is, the time it takes for gitoliteACL and gitolite to decide whether to grant or deny an access request. This is typically followed by transferring the appropriate data, that we do not include in the measured time. This focuses our measurement only on the access check. However, this makes individual access requests take a negligible amount of time to complete. In order to collect meaningful results, we measure the time it takes to complete 1000 of the same access request. We execute each experiment fifteen times, and average the measured time to produce the average access latency.

2.5.2 Impact of number of users with access request

The first experiment evaluates the impact on latency of an access request to a resource (a single repository) when increasing the number of users permitted access to the resource. Figure 2.5 shows a graph with the average latency of access request for gitoliteACL and gitolite. We vary the number of users permitted access to the resource (a single repository) from 0 to 295. Notice that for 0 users, an access check to a resource is simply denied. The results indicate that the average access latency for gitoliteACL is less than that of gitolite. The primary reason for the lower access latency for gitoliteACL is that gitoliteACL generates an ACL that only constitutes entries that allow access to the resource. gitolite on the other hand, must construct this information per access request. Furthermore, the latency for gitoliteACL does not change significantly as we increase the number of users allowed access to the resource. The reason is that, searching in gitoliteACL to check whether the query matches an entry or not can be done using binary search. Binary search is logarithmic in the number of ACL entries. Because we are considering only 300 users in this graph, the latency for gitoliteACL looks constant as we increase the number of users

allowed access to the resource. Also because we have checked the access for the first user in the list of users, the latency even looks more unchanged.

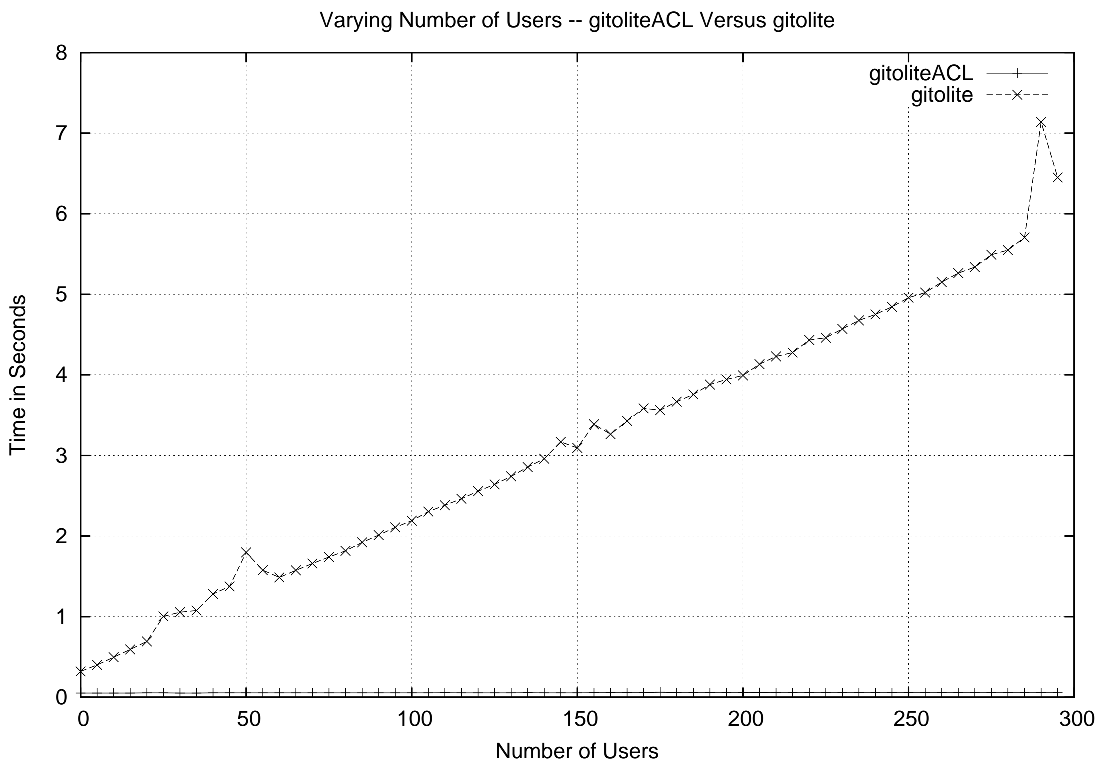

Figure 2.5: Access times when increasing the number of users accessing allowed access to the resource.

2.5.3 Impact of deny rules with increasing resources

The second experiment investigates the impact of increasing the number of deny rules for some randomly chosen users while increasing the number of resources. We experiment with three configurations of $0, 5$ and 10 users denied access to the resources. For gitolite, we notice that the average access latency does not increase across the three configurations. The reason is that in gitoliteno matter how many deny rules we have, we check if the query matches any of rules in the configuration file. The average access latency does not increase in gitoliteACL as well. That is because after producing the ACL, the only complexity we undertake is searching for a matched entry and it is not related to a deny or allow permission.

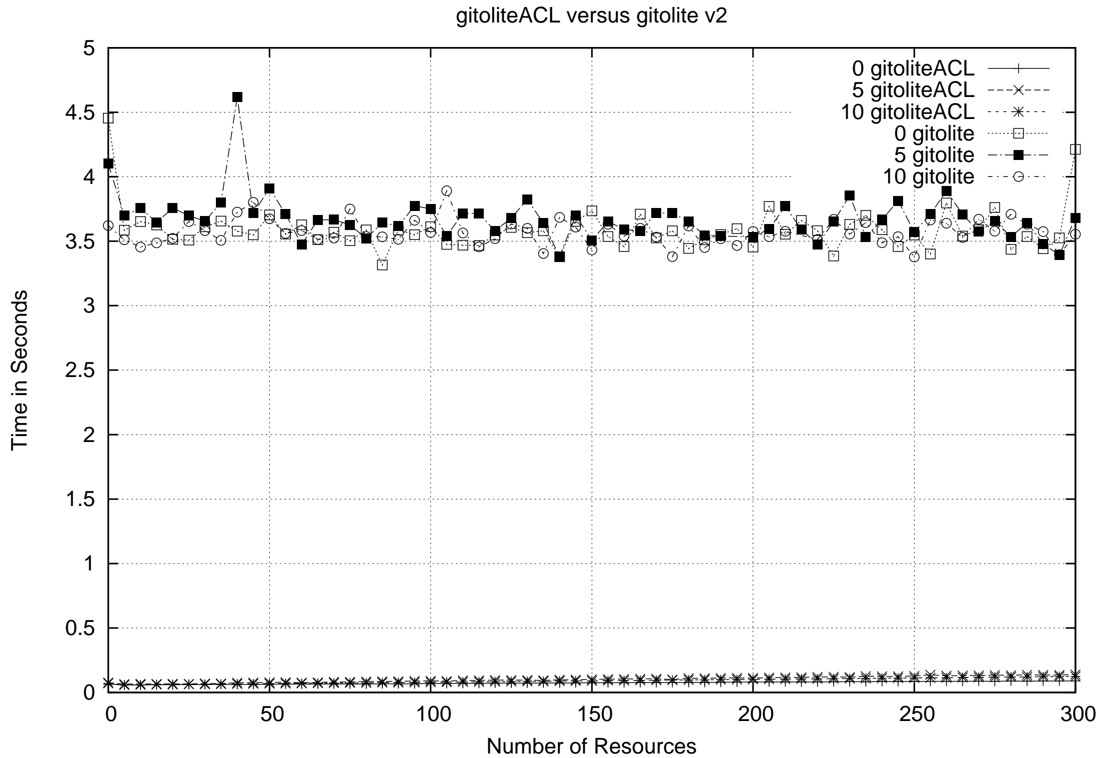

Figure 2.6: Access times when varying the number of resources while denying a certain number of users access to the every resource. 5 gitolite means that there are 5 users that are denied access to every resource.

2.6 Conclusion

We have proposed a semantics for an authorization system that is used for a version control system. Another important challenge we have solved is producing an Access Control List (ACL) based on the proposed authorization system. In access control enforcement we need to have an ACL based on the authorization system to respond the query. To produce the ACL we need to have a well defined semantics for the system to result in an ACL. We have proposed an algorithm to produce the ACL and showed complexity result for that. Also we have shown that it is sound and complete. We have shown in real world experiment the proposed algorithm works properly. We have discussed that our algorithm is considerably faster than already being used implementation of `gitolite` in many situations.

Chapter 3

Least-Restrictive Enforcement of the Chinese Wall Security Policy

3.1 Introduction

The *Chinese Wall security policy* relates to conflict of interest a subject may have over two pieces of information. The policy states that information from two objects that are to be confidential from one another should not flow to a subject. Conflict of interest is recognized as important in several settings – legal, financial and governmental [13], and technical, including in the emergent area of cloud-computing [14, 15, 16].

We address an issue at the foundations of the policy — its enforcement [19]. We begin with some discussions on the policy itself. To our knowledge, the policy was first articulated in information security research by Brewer and Nash [17]. Their work is cited extensively in the research literature [18], and included in popular books on information security [1, 32, 33].

In articulating the Chinese Wall security policy, Brewer-Nash adopt an abstraction for the authorization state of systems in which the policy is relevant. We show an example in Figure 3.1. A system comprises objects and subjects; the former is a passive container of information, and the latter actively accesses objects. There are only two kinds of access, read and write. When a subject reads an object, information flows from the object to the subject. When a subject writes an object, information flows from the subject to the object. Apart from such direct flows of information, there can be indirect flows of information. In Figure 3.1 for example, the subject s_2 read object o_6, and later wrote o_4; therefore, information has flowed from o_6 to o_4.

Each object belongs to a Company Dataset, and Company Datasets are grouped into Conflict of Interest classes. An object that contains confidential information is called an unsanitized object. If two unsanitized objects belong to different Company Datasets within the same Conflict of Interest class, then, for the Chinese Wall security policy to be maintained, information from both those objects should not flow to a subject. An exception is a distinguished Conflict of Interest class for what are called sanitized objects. Such objects do not contain confidential

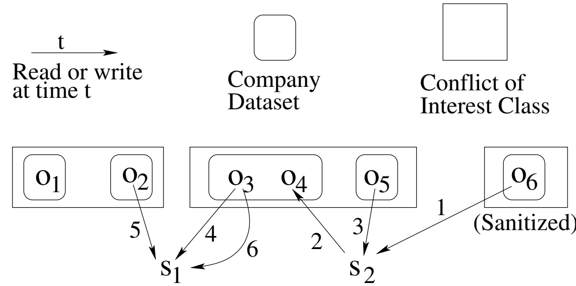

Figure 3.1: An authorization state with two subjects and six objects. As the legend expresses, boxes with sharp corners are Conflict of Interest classes and boxes with rounded edges are Company Datasets. An arrow indicates read or write at a particular time. The Conflict of Interest class to the far right, and the Company Dataset within it contain sanitized objects only.

information at the start, and therefore data that is initially contained in them is allowed to flow freely.

In Figure 3.1 for example, there are three Conflict of Interest classes, one of which is for sanitized objects. Objects o_3 and o_4 belong to the same Company Dataset within a Conflict of Interest class, and are therefore not in conflict. The objects o_2 and o_3 that belong to different Company Datasets that are in different Conflict of Interest classes are also not in conflict. The objects o_3 and o_5 that belong to different Company Datasets within the same Conflict of Interest class are in conflict and therefore we need to guard against data flow from those two objects that may be in violation of the policy.

Apart from articulating the policy, Brewer-Nash propose an enforcement mechanism for it. Their enforcement mechanism comprises two rules, the simple security rule and the $*$-rule [17]. A rule is a precondition on the authorization state of the system that must be satisfied before access is granted to a subject.

The simple security rule is: "Access is only granted if the object requested (a) is in the same Company Dataset as an object already accessed by that subject, or, (b) belongs to an entirely different Conflict of Interest Class." "Access" means read or write. The $*$-rule is: "Write access is only permitted if (a) access is permitted by the simple security rule, and, (b) no object can be read which is in a different Company Dataset to the one for which write access is requested and contains unsanitized information."

In Figure 3.1, for example, if the subject s_1 attempts to read o_1, it is disallowed by the simple security rule as o_1 belongs to the same Conflict of Interest class as another object, o_2, that it has already read. If it attempts to read o_2 or o_3 again, it is allowed to do so by the simple security rule. The subject s_1 is disallowed from writing any object by the $*$-rule because it is allowed to read both o_2 and o_3, which are in different Company Datasets.

We point out, however, that if s_1 is allowed to write to o_2 or o_3, no violation of the policy results. This is an example of the restrictiveness of the Brewer-Nash enforcement mechanism. While the enforcement mechanism (simple-security and $*$-rules) is sufficient to preclude conflict of interest, it is not necessary.

We point out also that in Figure 3.1, at time 2, the $*$-rule of Brewer-Nash disallows s_2 from writing to o_4. No flow of information to a subject from objects that are in conflict results from such an action by s_2. However, at time 7, such a write results in information flow from o_5 to o_4, which are in conflict. It is unclear whether Brewer-Nash intend for this to be a violation of the policy. We address it as well, as some subsequent work appears to adopt this as a policy violation. (We call it an "object violation" of the policy, as opposed to a "subject violation" — see Definition 17 in Section 3.2.2.)

Work subsequent to Brewer-Nash has pointed out that their enforcement mechanism can be highly restrictive. Specifically, Kessler [20] and Sandhu [21, 22] establish the following two properties:

P_1: A subject that has read objects from two or more Company Datasets cannot write to any object.

P_2: A subject that has read objects from exactly one Company Dataset can write to that Dataset only.

Our work We address the following questions that, to our knowledge, have not been addressed previously. Can we articulate a notion of least-restrictive enforcement for the Chinese Wall security policy? If so, does there exist an enforcement mechanism that is least-restrictive? What are the trade-offs it incurs?

We answer the above questions by first proposing a notion of least-restrictive enforcement: under such enforcement, an authorization state is reachable if and only if it does not violate the policy. We then propose an enforcement mechanism and show that it is least-restrictive. We identify that the trade-off we incur regards the independence of the actions of a subject from another. That is, the read or write actions of a subject can constrain prospective read or write actions of another subject. We then show that this trade-off in inherent to least-restrictive enforcement — any enforcement mechanism that is least-restrictive must incur this trade-off.

Separately, we show that the Brewer-Nash enforcement mechanism is even more restrictive than previous work establishes. We establish two new results. (1) The second sub-rule of the $*$-rule implies the simple security rule (the first sub-rule of the $*$-rule). (2) If a subject is authorized to write to an unsanitized object, then all unsanitized objects must belong to the same Conflict of Interest class.

Our work not only provides a least-restrictive enforcement mechanism of the Chinese Wall security policy that incurs only the trade-off it must, but also sheds new light on Brewer-Nash, which is generally considered to be important work in information security.

In the next section, we discuss a graph-based formalism to represent the authorization state and express the Chinese Wall security policy precisely.

3.2 A Formalism and the Chinese Wall Security Policy

We seek to carefully distinguish policy from enforcement mechanism. In this section, we articulate the policy precisely. As the Chinese Wall security policy pertains to accesses subjects have made to objects, we adopt a graph-based formalism for what we call authorization states that capture such accesses.

We first describe our graph-based formalism in Section 3.2.1. Then, in Section 3.2.2 we express the Chinese Wall security policy, and in Section 3.2.3, we discuss ways in which an authorization state can change. (Henceforth, we sometimes simply say "state" for "authorization state," and "policy" for "Chinese Wall security policy.")

3.2.1 Authorization State

An authorization state is a tuple $\langle D, \delta, C, G \rangle$, where $G = \langle V, \mathit{exists}, E \rangle$ is a graph, and:

- D is a finite set of company datasets $\{D_{[s]}, D_1, \ldots, D_k\}$. The Dataset $D_{[s]}$ is the one reserved for sanitized objects. All other Datasets contain unsanitized objects.
- $V = S \cup O$ is the set of vertices which represent the subjects and objects. We assume that $S \cap O = \emptyset$, and:
  - $S = \{s_1, s_2, \ldots\}$ is a countably infinite set of subjects. These are all the subjects that can possibly exist.

    We use the term "subject," rather than "user" or "principal" intentionally. Sandhu [21, 22] points out some confusion in Brewer-Nash on this issue. We adopt the mindset from his work. A subject may be created and destroyed, and is an agent of a user; for example, an operating system process.
  - $O = \{o_1, o_2, \ldots\}$ is a countably infinite set of objects. These are all the objects that can possibly exist.
- The function *exists*: $V \longrightarrow \{0, 1\}$ associates each vertex with a bit that indicates whether it exists. We assume that in any given state, only finitely many subjects and objects exist.

  One may wonder why we model subjects and objects in this manner, and not simply have only those subjects and objects that exist as vertices in an authorization state. The reason is that, as we discuss in Section 3.2.3, subjects and objects that exist can be destroyed later. Even after a subject or object is destroyed, it can be part of an information flow that leads to a violation of the Chinese Wall security policy (see the discussion after Definition 17 in Section 3.2.2). Consequently, we keep track of all subjects and objects that can possibly exist, and annotate the ones that exist using the *exists* function.
- The function δ: $O \longrightarrow D$ maps an object to a Company Dataset.

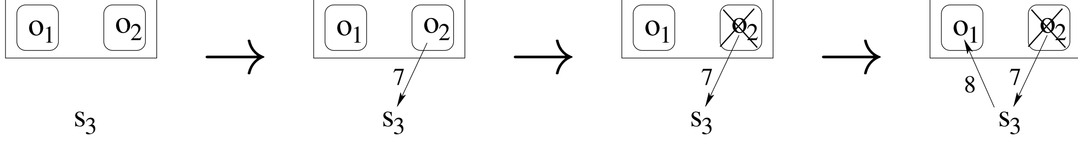

Figure 3.2: Changes to the authorization state starting at the state in Figure 3.1. We show only the Conflict of Interest class that contains o_1 and o_2. In the state to the far left, the subject s_3 is created. In the next state, s_3 reads o_2. In the next state, o_2 is destroyed. In the final state to the far right, s_3 writes to o_1 causing an object-violation. All these states are reachable from the state in Figure 3.1.

- The function $C\colon D \times D \longrightarrow \{0,1\}$ identifies Conflict of Interest. That is, $C(D_i, D_j) = 1$ if and only if $D_i \neq D_j$, $D_i \neq D_{[s]}$, $D_j \neq D_{[s]}$ and D_i and D_j are in conflict (i.e., in the same Conflict of Interest class).

  It may seem that we could simply partition D into Conflict of Interest classes instead of specifying the function C. We choose this approach to address Lin's generalization [34] to the Brewer-Nash characterization of Conflict of Interest classes. Lin's work points out that the Brewer-Nash characterization requires the Conflict of Interest relation to be transitive, when in practice, it does not have to be. In our case, the relation $\{\langle D_i, D_j\rangle : C(D_i, D_j) = 1\}$ is irreflexive, symmetric and not necessarily transitive.

- $E \subseteq V \times V \times \mathbb{Z}^+$ is the set of edges, where $\mathbb{Z}^+$ is the set of positive integers. Each $\langle v_1, v_2, t\rangle \in E$ represents an access that occurred at time t. If $v_1 \in S$, then $v_2 \in O$, and the access was a write. If $v_1 \in O$, then $v_2 \in S$, and the access was a read.

We can represent the system in Figure 3.1 using our formalism. In the graph, we would have $|D| = 5$, and $C(D_i, D_j)$ would correspond to the Conflict of Interest classes shown in the figure. For example, $C(\delta(o_1), \delta(o_2)) = 1$, but $C(\delta(o_3), \delta(o_4)) = 0$. The vertices for which *exists* $= 1$ are those shown in the figure, and the only edges that exist are those between vertices that exist.

3.2.2 The Chinese Wall Security Policy

In this section, we first introduce the notion of an information flow in a state, which is a kind of path in the graph. We then characterize the Chinese Wall security policy.

Definition 16 (Information Flow) *In an authorization state (a graph) G, an information flow from vertex v_1 to v_n, denoted $v_1 \rightsquigarrow v_n$, is a path with edges $\langle v_1, v_2, t_{1,2}\rangle$, $\langle v_2, v_3, t_{2,3}\rangle$, …, $\langle v_{n-1}, v_n, t_{n-1,n}\rangle$ such that $t_{1,2} \leq t_{2,3} \leq \ldots \leq t_{n-1,n}$.*

For example, in the state that corresponds to Figure 3.1, $o_6 \rightarrow s_2 \rightarrow o_4$ is an information flow. Subjects and objects alternate in a path that is an information flow; i.e., information can

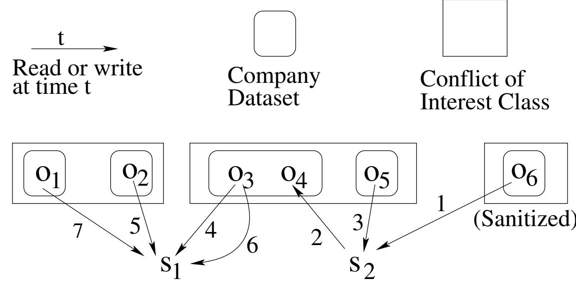

Figure 3.3: A subject-violation example. The policy is not satisfied for subjects in this state.

flow between two objects only via a subject, and information can flow between two subjects only via an object.

Brewer and Nash [17] express the Chinese Wall security policy as, "[Subjects] are only allowed access to information that is not held to conflict with any other information that they already possess." We express this in our graph-based formalism as follows. As we discuss in Section 3.1, we capture also the policy for flows between objects.

Definition 17 (Chinese Wall Security Policy) *The Chinese Wall security policy is said to be satisfied for subjects in a state if there exists no pair of information flows, $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ such that $s \in S$, $o_1, o_2 \in O$, and, $C(\delta(o_1), \delta(o_2)) = 1$. The policy is said to be satisfied for objects if there exists no pair of information flows, $o_1 \rightsquigarrow o$ (or $o_1 = o$) and $o_2 \rightsquigarrow o$ (or $o_2 = o$) such that $o, o_1, o_2 \in O$, and, $C(\delta(o_1), \delta(o_2)) = 1$.*

If the policy is not satisfied for subjects in a state, then we say that there is a *subject-violation* of the policy in the state. As an example in Figure 3.3 the policy is not satisfied, because s_1 has read access to both objects o_1 and o_3 which are in conflict.

If the policy is not satisfied for objects, then we say that there is an *object-violation* of the policy. As an example, the policy is not satisfied in the authorization state that corresponds to Figure 3.4, because there is an information flow from o_5 to o_3 which are in conflict.

The Chinese Wall security policy is satisfied for both subjects and objects in the state that corresponds to Figure 3.1. However, if s_2 writes to o_4 at time 7, we have an object-violation from the flow $o_5 \rightsquigarrow o_4$. If s_1 reads o_1, there is a subject-violation from the flows $o_1 \rightsquigarrow s_1$ and $o_2 \rightsquigarrow s_1$.

We point out that for our definitions above, a violation can be caused by a flow in which there is a subject or object that has been destroyed. This is reasonable: even if an object is destroyed, for example, if a subject has previously read that object, we need to consider that flow of information in any future determination of a violation of the policy.

We point out also that subject- and object-violations are mutually exclusive in that one is neither a necessary nor a sufficient condition for the other to exist. However, if an object-violation exists, then there exists a read action by one of a particular set of subjects that causes a subject-violation. Similarly, if a subject-violation exists, then there exists a write action by one of a particular set of subjects that causes an object-violation.

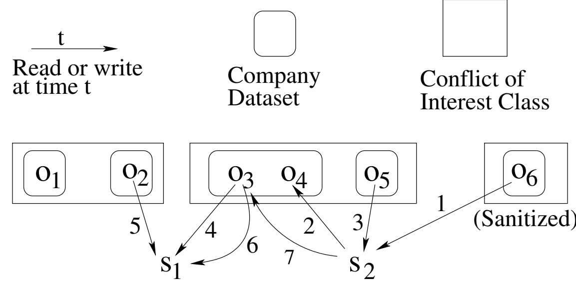

Figure 3.4: An object-violation example. The policy is not satisfied for objects in this state.

3.2.3 Changes to the Authorization State

Of course, the Chinese Wall policy is interesting only if we consider changes to the authorization state as actions are performed. The only part of the state that can change is the graph, G. Within G, the only possible changes are to *exists*, and the set of edges E. The former happens when a subject or object is created or destroyed, and the latter when a subject reads or writes an object. Edges can only be added, and never removed, because an edge represents a read or write action, which cannot be revoked once it is performed.

Brewer-Nash explicitly consider only the actions read and write, and not the creation and destruction of subjects and objects. However, their rules for creation and destruction are straightforward to intuit (see Section 3.5). Also, some of the work subsequent to theirs such as that of Sandhu [21, 22] considers creation and destruction of subjects and objects explicitly.

In each of the following, we denote as G the current state, and as G_{new} the state that results from the change.

- A subject or object may be created. If $\alpha \in S \cup O$ is the subject/object that is created, *exists*$(\alpha) = 0$ in G (α does not already exist in G). The only difference between G and G_{new} is that in the latter, *exists*$(\alpha) = 1$.

  In the context of enforcement in the next section, we make the additional assumption that once a subject or object is destroyed, it cannot be recreated. We defer a discussion on this to the next section.

- A subject or object may be destroyed. If $\alpha \in S \cup O$ is destroyed, *exists*$(\alpha) = 1$ in G, and *exists*$(\alpha) = 0$ in G_{new}. That is the only difference between G and G_{new}.

- A subject may read or write an object. Suppose $s \in S$ is a subject that reads or writes $o \in O$. Then, in G, *exists*(s) = *exists*$(o) = 1$; that is, both s and o exist in G. The only difference between G and G_{new} is that an edge that corresponds to the action additionally exists in G_{new}, with a timestamp that corresponds to the time the action occurs.

  We assume that time is discretized, and only one action occurs at a time. The latter assumption is needed for clarity of exposition only. We can allow multiple actions at the

same time without affecting any of our contributions. Our use of "$\leq$" in relating the time-stamps $t_{1,2}, t_{2,3}, \ldots, t_{n-1,n}$ in Definition 16 accounts for this. If only one action is allowed to happen at a time, then we can change that to "$<$."

Definition 18 (Reachable State) *Given two states σ_1 and σ_n, we say that σ_n is reachable from σ_1 if and only if either $\sigma_1 = \sigma_n$, or there exists a state σ_{n-1} that is reachable from σ_1 and σ_n results from a state-change to σ_{n-1}.*

In Figure 3.2, we show examples of reachable states starting at the state in Figure 3.1. One may ask why we have chosen state-changes at the above granularity. Why not allow, for example, multiple subjects to be created in a single state-change? The granularity is important for reads and writes only. That is, from the standpoint of enforcing the Chinese Wall security policy, a read and a write must each constitute a state-change, and is atomic. The reason, as we discuss in the next section, is that we may need to prevent a particular read or write from happening to ensure that the policy is met in all reachable states.

We point out that the simple security and $*$-rules of Brewer-Nash are articulated with exactly such a granularity of state-changes in mind for reads and writes. As for the other state-changes (creation and destruction of subjects and objects), the granularity is unimportant. That is, we can allow multiple subjects and objects to be created and destroyed simultaneously without affecting any of our contributions. The reason is that those state-changes by themselves cannot cause the satisfaction of the policy to be affected.

3.3 Enforcement

We now consider enforcement of the Chinese Wall security policy. Enforcement is the process by which we allow or disallow a prospective change to the authorization state. An enforcement mechanism specifies, given a particular state, which changes are allowed. We define it more precisely below.

In the context of the Chinese Wall security policy, an attempt to create or destroy a subject or object always succeeds, so long as the subject/object being created has not been created before, and the subject/object being destroyed exists. The reason, as we discuss in the previous section, is that those state-changes do not affect satisfaction of the policy. Any conditions on such actions is beyond the scope of the policy. Of course, an enforcement mechanism may want to keep track of which subjects/objects exist. The main focus of an enforcement mechanism is whether an attempt by a subject that exists to read or write an object that exists is allowed.

Definition 19 (Enforcement mechanism) *An* enforcement mechanism *is an algorithm that either allows or denies a request for a state-change (creation, destruction, read and write). Some persistent data that we call* enforcement state *may be consulted and updated by the algorithm.*

We have kept the above definition somewhat informal. We could, instead, have defined an enforcement mechanism by characterizing it as a particular kind of automaton, as is done in some prior work on access enforcement (see, for example, [35, 36]). We have adopted the above definition for clarity of exposition but without compromising rigor or correctness.

Enforcement state The enforcement state "shadows" a corresponding authorization state. Indeed, an enforcement state could be exactly the corresponding authorization state. However, that would not correspond to an efficient enforcement mechanism. The reason is that we would then be maintaining data even for subjects and objects that do not exist — that size can be unbounded in the number of subjects and objects that exist.

It is reasonable that the size of the enforcement state is a non-constant (e.g., linear) function of the number of subjects and objects that exist because in any realistic system, some data is associated with something that exists to indicate that it exists.

An example A simple example of an enforcement mechanism is one that maintains, as the enforcement state, a set of subjects and objects that exist, allows the creation of any subject or object that does not already exist, allows any subject that exists to write any object that exists, and disallows all reads. When a subject or object is created, it is added to the set that is the enforcement state, and when a subject or object is destroyed, it is removed from the set.

Such an enforcement mechanism is time and space efficient — the enforcement state is linear in the number of subjects and objects that exist, and the algorithms are each at worst linear-time in the number of subjects and objects that exist. It also ensures that the Chinese Wall security policy is always met. However, it is highly restrictive as a consequence of disallowing any attempt to read. Large numbers of states that do not violate the Chinese Wall security policy are precluded from being reached.

Least-restrictive enforcement This leads us to our notion of a least-restrictive enforcement mechanism, and axes for trade-offs other than restrictiveness. We define the former in this section, and discuss the latter in the next section. Before we define what it means for an enforcement mechanism to be least-restrictive, we define the notion of an empty authorization state.

Definition 20 (Empty Authorization State) *An empty authorization state is a graph as characterized in Section 3.2.1 in which for all $v \in V$, exists$(v) = 0$ and $E = \emptyset$.*

An empty state can be seen as characterizing a system at the start. No subjects or objects have been created yet, and nothing has been accessed. We discuss why we need it after our following definition on least restrictive enforcement.

Definition 21 (Least-restrictive) *Let σ_0 be the empty state. Let $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ be a sequence of states such that σ_{i+1} is reachable from σ_i via a state-change. Let σ_n be such that the Chinese Wall security policy is satisfied in it. An enforcement mechanism is said to be least-restrictive if it allows enforcement state related to σ_n to be reached from σ_0 via some sequence of state-changes.*

The above definition of a least-restrictive enforcement mechanism captures our intent that every state that does not violate the policy should be reachable from σ_0 in enforcement. The reason we insist that the start state of any sequence we consider is the empty state σ_0 is because we want the particular enforcement mechanism under consideration to be used from the start of the system. An enforcement mechanism maintains an enforcement state as we specify in Definition 19. We expect that the enforcement state that corresponds to σ_0 for any enforcement mechanism is empty. If we allow a sequence of state-changes to begin at some arbitrary authorization state, we would have to allow a particular enforcement mechanism to initialize its enforcement state to be consistent with that start state. We simply require the start state to be σ_0 instead.

In the above definition, it may seem interesting that we do not require the intermediate states $\sigma_1, \ldots, \sigma_{n-1}$ to also satisfy the policy. The reason is captured by the following lemma.

Lemma 3 *Let $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n$ be a sequence of authorization states each reachable from the previous by a state-change, where σ_0 is the empty state. Let σ_k for $k \in [0, n]$ violate the Chinese Wall security policy. Then, each of the states $\sigma_{k+1}, \ldots, \sigma_n$ violates the policy as well.*

PROOF. Let some state σ_i where $i \in [k+1, n]$, not violate the policy when some prior state σ_k does. We use induction on $i - k$. For the base case, $i = k + 1$. We do a case-analysis on the state-change $\sigma_k \rightarrow \sigma_{k+1}$ and produce a contradiction for each case in a straightforward manner. If the state-change is the creation of a subject or object, then this cannot affect the violation of the policy that exists in σ_k. We observe similarly that the destruction of a subject or object, or a read or write action by a subject on an object do not affect the policy violation that exists in σ_k.

We adopt the induction assumption that for $i = k + m - 1$, where $m \geq 1$, all the states $\sigma_k, \ldots, \sigma_i$ violate conflict of interest. For the step, we adopt $i = k + m$, and perform a case-analysis similar to that for the base case above for the state-change $\sigma_{i-1} \rightarrow \sigma_i$ and produce a contradiction. □

As a consequence of the above lemma, in Definition 21, given that σ_n satisfies the Chinese Wall security policy, we know that each of the prior states $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ satisfies it as well. Furthermore, we are able to assert the following lemma that states that if a state σ satisfies the policy, then an enforcement mechanism that is least-restrictive allows *every* possible sequence of states that reaches σ from the empty state. (We recognize that we could have simply defined "least-restrictive" to be this. However, we argue that ours is a better enforcement mechanism — adopt the notion that appears weaker as the definition, and then assert this stronger result as a lemma.)

Lemma 4 *Let $\mathcal{A}$ be a least-restrictive enforcement mechanism for the Chinese Wall security policy. Let the policy be satisfied in the state σ_n and let $\Psi = \sigma_0 \rightarrow \ldots \rightarrow \sigma_n$, where σ_0 is the empty state, be some sequence of states via which σ_n is reached. Then, $\mathcal{A}$ allows the sequence Ψ to occur.*

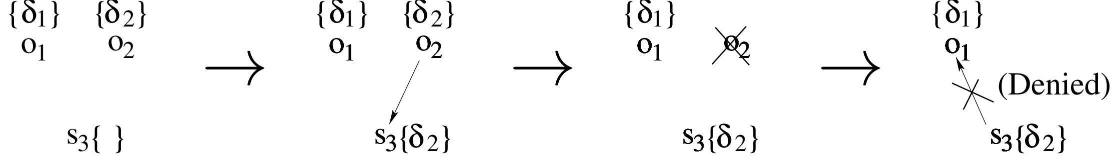

Figure 3.5: Our enforcement algorithm for the state-changes from Figure 3.2. We denote $\delta(o_1) = \delta_1$ and $\delta(o_2) = \delta_2$. The enforcement state that corresponds to each subject and object is shown in "{ }." The figure to the far left indicates that no information from any Company Dataset other than the one to which each belongs has flowed into o_1 and o_2. As s_3 is created in that state, the set that corresponds to it is empty. The write by s_3 in the final state is denied because the "if" condition in WRITE evaluates to true.

PROOF. From Lemma 3 we know that the policy is satisfied for any σ_k in Ψ. Therefore, σ_k is reachable under $\mathcal{A}$. Induction on k completes the proof. □

3.3.1 Axes for Trade-offs

While the main focus of this dissertation is least-restrictive enforcement, we recognize that an enforcement mechanism that is least-restrictive should not be unduly poor in other properties of interest such as time- and space-efficiency. Indeed, in Section 3.4.2, we ask along what axis we (need to) trade-off to achieve least-restriction.

Example For example, it is possible to conceive an enforcement mechanism that is least-restrictive that maintains as its enforcement state the authorization state (the graph). The enforcement mechanism would work as follows. If there is an attempt by a subject to read or write, we simply simulate that action by changing the graph as though the attempt is allowed — that is, we add the corresponding edge to the graph. We then run a graph-search algorithm similar to Floyd-Warshall's [37, 38] to determine whether the policy is violated in the new graph. Such an algorithm would correctly tell us whether the policy is violated or not because information flow demonstrates optimal substructure similar to shortest-paths in graphs — a sub-path of an information flow is an information flow.

If we discover that the policy is violated in the new graph, we disallow the request and retain the current authorization state as our enforcement state. If it is not, we allow it and adopt the new state as the enforcement state. It is possible to show that this enforcement mechanism is indeed least-restrictive. However, as we discuss in the previous section, it is inefficient from the standpoint of space and time. The space is unbounded in the number of subjects and objects that exist. And this in turn causes an algorithm such as Floyd-Warshall's [37, 38] to be inefficient, as it is polynomial-time in the input graph.

The axes we consider

- *Soundness* – we want an enforcement mechanism to be sound — an attempt to read or write is denied if it would cause a violation of the policy. This captures the basic security property that we need to maintain.

- *Time- and Space-efficiency* – we would want an enforcement mechanism to be time- and space-efficient. By "space," we mean what we call enforcement state, and any other auxiliary space used by an algorithm in the enforcement mechanism.

- *Least-restriction* – we would want an enforcement mechanism to be least-restrictive in the authorization states it allows to be reached.

- *Independence of subjects* – an issue with which prior work [20] deals is whether the prospective actions of a subject can be constrained by the actions of other subjects. This can be deemed undesirable. We make this notion of independence of subjects more precise in the following definition so we can articulate Theorem 10 in Section 3.4.3. That theorem asserts that it is impossible to achieve least-restriction without trading-off independence of subjects.

  Definition 22 (Independence of subjects) *Let σ_0 be the empty state and $\mathcal{A}$ be an enforcement mechanism. Starting at σ_0, suppose $\Psi_a = \langle \psi_1, \psi_2, \ldots, \psi_n \rangle$ is a sequence of state-changes that is allowed by $\mathcal{A}$, but the state-change ψ_{n+1} that is the read or write by a subject s on some object is not allowed by $\mathcal{A}$ immediately after Ψ_a. Let Ψ_b be a subsequence of Ψ_a that we get by removing one or more read and write actions of a subject $s' \neq s$ such that $\mathcal{A}$ allows ψ_{n+1} immediately after Ψ_b. Then we say that under $\mathcal{A}$, the actions of a subject are not independent of the actions of other subjects. If no such pair $\langle \Psi_a, \Psi_b \rangle$ exists, then we say that under $\mathcal{A}$, the actions of a subject are independent.*

  We point out that our above definition has similarities to non-interference [39, 40].

- *Access to sanitized objects* – given that the intent of sanitized objects is to contain public information, it is desirable that subjects are able to read such objects freely so long as there has been no flow of information into a sanitized object from an unsanitized object.

- *Transitivity of the conflict of interest relation* – as Lin [34] points out, the conflict of interest relation is not necessarily transitive, and therefore an enforcement mechanism should not assume that it is. As an example in Figure 3.6 o_5 is in conflict with o_2 and o_4. But o_2 and o_4 are not in same conflict of interest class.

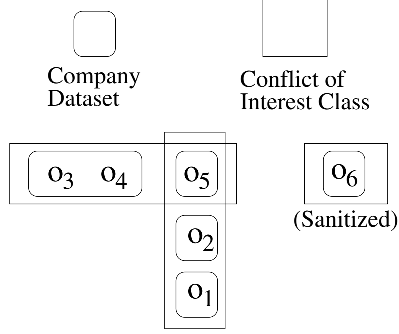

Figure 3.6: An example of non-transitive conflict of interest relation

3.4 Our Approach

In this section, we present our enforcement mechanism and show that it is least-restrictive. We discuss also how well it does along the other axes for trade-offs that we discuss in the previous section.

Enforcement state The enforcement state in our approach, $\mathcal{V}$, is a set of sets. We maintain a set for every subject and object that exists; that is, has been created and not destroyed. Let $\Delta_v \in \mathcal{V}$ be associated with $v \in S \cup O$ where *exists*$(v) = 1$. Then $\Delta_v \subseteq D$, where D is the set of Company Datasets (see Section 3.2.1). $\mathcal{V}$ is initialized to $\emptyset$.

Algorithm Our enforcement algorithm is conceptually simple. When information flows from v_1 to v_2 via a read or write, labeling enforcement propagates some "control data" as well — the set of Company Datasets with which v_1 is associated. We present the algorithm as a set of sub-routines, each corresponding to the kind of state-change that is attempted.

CREATE (v)

> **if** $\exists \Delta_v \in \mathcal{V}$ **then** Error;
> **if** $v \in O$ **then** $\Delta_v \leftarrow \{\delta(v)\}$ **else** $\Delta_v \leftarrow \emptyset$;
> $\mathcal{V} \leftarrow \mathcal{V} \cup \Delta_v$;

DESTROY (v)

> $\mathcal{V} \leftarrow \mathcal{V} - \Delta_v$;

READ (s, o)

> **if** $\exists d_i, d_j$ *with* $\{d_i, d_j\} \subseteq \Delta_s \cup \Delta_o$ *and* $C(d_i, d_j) = 1$ **then** deny;
> **else** $\Delta_s \leftarrow \Delta_s \cup \Delta_o$; allow;

WRITE (s, o)

if $\exists d_i, d_j$ *with* $\{d_i, d_j\} \subseteq \Delta_s \cup \Delta_o$ *and* $C(d_i, d_j) = 1$ **then** deny;
else $\Delta_o \leftarrow \Delta_s \cup \Delta_o$; allow;

When a subject or object v is created, CREATE initializes a set Δ_v and adds it to the enforcement state, $\mathcal{V}$. Δ_v is set to $\emptyset$ if v is a subject, and the Company Dataset to which v belongs if v is an object. When a subject or object is destroyed, DESTROY removes the set that corresponds to it from the enforcement state, $\mathcal{V}$.

When a subject s attempts to read or write an object o, it may be denied by the corresponding "if" condition, which examines $\Delta_s \cup \Delta_o$ for the existence of company datasets that are in conflict. If a read by s of o succeeds, then Δ_s is updated, and if a write succeeds, then Δ_o is updated.

The "if" condition in READ above is needed only if we wish to prevent subject-violations. The "if" condition in WRITE is needed only if we wish to prevent object-violations. If, for example, we seek to prevent subject-violations only and not object-violations, then we can remove the "if" condition in WRITE (thereby allowing all write requests), and only update Δ_o as specified under "else" in WRITE.

The conceptual simplicity of our approach ensures its efficiency, and belies its power. In the next sections, we discuss how our approach performs along each axis from Section 3.3.1. In particular, in Section 3.4.2 we show that it is least-restrictive, and in Section 3.4.3 we show that any enforcement mechanism that is least-restrictive must incur the same trade-off that our enforcement mechanism does. In Section 3.4.4, we address the question as to whether ours is the only possible enforcement mechanism to enforcing the Chinese Wall security policy that is least-restrictive.

3.4.1 Soundness

The soundness of our enforcement mechanism is based on the following lemma regarding the set Δ_v that corresponds to a vertex v in the enforcement state $\mathcal{V}$ in our enforcement mechanism.

Lemma 5 *Let $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ be an authorization state-change sequence that is allowed by our enforcement mechanism. Let $v_1 \rightsquigarrow v_2$ be an information flow in σ_n where $v_1 \in O$ and exists$(v_2) = 1$. Then, in the enforcement state, $\mathcal{V}$, that corresponds to σ_n, (1) $\exists \Delta_{v_2} \in \mathcal{V}$, and, (2) $\delta(v_1) \in \Delta_{v_2}$.*

PROOF. For (1), we observe that Δ_{v_2} is created and added to $\mathcal{V}$ in CREATE, and is removed in DESTROY only. Therefore, given that v_2 is created, but not destroyed in the sequence $\sigma_0 \to \ldots \to \sigma_n$, we know (1) is true. For (2), we use induction on n.

For the base cases with $n = 0$ and $n = 1$, the assertion is trivially true as there is no information flow in either σ_0 or σ_1 — there are no vertices in σ_0, and at most one vertex in σ_1.

For the step, we assume that the assertion is true for all n between 0 and k. We consider the case that $n = k + 1$, and perform a case-analysis on the state-change $\sigma_k \rightarrow \sigma_{k+1}$.

If the state-change is a creation or destruction of a subject or object, no new information flows are added to σ_k, and we rely directly on the induction assumption for the proof. Consider the case that the state-change is a read by subject s on object o. Then, the only new information flows that are added are into s. We observe also that the only component of the enforcement state that changes in READ is Δ_s. Consider an information flow $v \rightsquigarrow s$, for some $v \in O$, in σ_{n+1} that does not exist in σ_n. As the only new edge in E is from o to s, there must have existed an information flow $v \rightsquigarrow o$ in σ_n. By the induction assumption, in the enforcement state that corresponds to σ_n, $\delta(v) \in \Delta_o$. The component Δ_o does not change between σ_n and σ_{n+1}, and in READ, we perform $\Delta_s \leftarrow \Delta_s \cup \Delta_o$, and therefore $\delta(v) \in \Delta_s$ in the enforcement state that corresponds to σ_{n+1}, as desired.

Now consider the case that the state-change $\sigma_n \rightarrow \sigma_{n+1}$ is a write by s to o. Then, the only new information flows that are added are into o. We observe also that the only component of the enforcement state that changes in WRITE is Δ_o. Consider an information flow $v \rightsquigarrow o$, for some $v \in O$, in σ_{n+1} that does not exist in σ_n. As the only new edge in E is from s to o, there must have existed an information flow $v \rightsquigarrow s$ in σ_n. By the induction assumption, in the enforcement state that corresponds to σ_n, $\delta(v) \in \Delta_s$. The component Δ_s does not change between σ_n and σ_{n+1}, and in WRITE, we perform $\Delta_o \leftarrow \Delta_s \cup \Delta_o$, and therefore $\delta(v) \in \Delta_o$ in the enforcement state that corresponds to σ_{n+1}, as desired.

□

Theorem 8 (Soundness) *Let $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n$ be an authorization state-change sequence that is allowed by our enforcement mechanism such that σ_n satisfies the Chinese Wall security policy. Let $\sigma_n \rightarrow \sigma_{n+1}$ be a state-change such that the policy is violated in σ_{n+1}. Then our enforcement mechanism disallows the sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma_{n+1}$.*

PROOF.

We have only two cases. (1) one of the state-changes $\sigma_i \rightarrow \sigma_{i+1}$ for $i \in [0, n-1]$ is disallowed by our enforcement mechanism. In this case, we are done as the sequence is disallowed by our enforcement mechanism. (2) all of the state-changes $\sigma_i \rightarrow \sigma_{i+1}$ for $i \in [0, n-1]$ are allowed by our enforcement mechanism. In this case, we need to show that $\sigma_n \rightarrow \sigma_{n+1}$ is disallowed by our enforcement mechanism. If the policy is subject-violated in σ_{n+1} but not σ_n, we know that the state-change $\sigma_n \rightarrow \sigma_{n+1}$ is a read by some subject s on some object o, and all causes for the policy being subject-violated in σ_{n+1} are pairs of paths that terminate at s.

If the policy is object-violated in σ_{n+1} but not σ_n, we know that the state-change $\sigma_n \rightarrow \sigma_{n+1}$ is a write by some subject s on some object o, and all causes for the policy being object-violated in σ_{n+1} is a path that terminate at o.

- Subject-violation: Assume, for the purpose of contradiction, that our enforcement mechanism allows the state-change, and in σ_{n+1} we have the two information flows $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ with $C(\delta(o_1), \delta(o_2)) = 1$. As our enforcement mechanism allows the state-change, we know that the "if" condition in READ evaluates to false.

  We claim that there are only two possibilities in the enforcement state that corresponds to σ_n. Either (a) $\{\delta(o_1), \delta(o_2)\} \subseteq \Delta_o$. Or, (b) one of $\delta(o_1)$ or $\delta(o_2)$ is in Δ_s, and the other is in Δ_o. If we are able to prove this claim, then we have the desired contradiction, as the "if" condition in READ would evaluate to true when s attempts to read o.

  To prove the claim, we first observe that we have only one of the following two possibilities. Either one of the information flows $o_1 \rightsquigarrow s$ or $o_2 \rightsquigarrow s$ is in σ_n, or neither is. We point out that both information flows cannot be in σ_n as then σ_n contains a subject-violation, which contradicts our assumption. In the former case, assume without loss of generality that $o_1 \rightsquigarrow s$ is in σ_n. Then, by Lemma 5, $\delta(o_1) \in \Delta_s$ in the enforcement state that corresponds to σ_n. The only edge added to E between σ_n and σ_{n+1} is from o to s. As we have a new path $o_2 \rightsquigarrow s$ in σ_{n+1}, we must have $o_2 \rightsquigarrow o$ or $o_2 = o$ in σ_n. By Lemma 5 (for the case that $o_2 \rightsquigarrow o$) and how CREATE works for objects (for the case that $o_2 = o$), $\delta(o_2) \in \Delta_o$ in the enforcement state that corresponds to σ_n. This corresponds to Case (b) above.

  In the latter case that neither $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ is in σ_n, both paths are created newly in σ_{n+1} by the addition of the edge from o to s in E. Therefore, in σ_n, $o_1 \rightsquigarrow o$ and $o_2 \rightsquigarrow o$, and by Lemma 5, in the enforcement state that corresponds to σ_n, $\{\delta(o_1), \delta(o_2)\} \subseteq \Delta_o$. This is Case (a) above.

- Object-violation: Assume, for the purpose of contradiction, that our enforcement mechanism allows the state-change, and in σ_{n+1} we have the two information flows $o_1 \rightsquigarrow o$ (or $o_1 = o$) and $o_2 \rightsquigarrow o$ (or $o_2 = o$) with $C(\delta(o_1), \delta(o_2)) = 1$. As our enforcement mechanism allows the state-change, we know that the "if" condition in WRITE evaluates to false.

  We claim that there are only two possibilities in the enforcement state that corresponds to σ_n. Either (a) $\{\delta(o_1), \delta(o_2)\} \subseteq \Delta_s$. Or, (b) one of $\delta(o_1)$ or $\delta(o_2)$ is in Δ_s, and the other is in Δ_o. If we are able to prove this claim, then we have the desired contradiction, as the "if" condition in WRITE would evaluate to true when s attempts to write o.

  To prove the claim, we first observe that we have only one of the following two possibilities. Either one of the information flows $o_1 \rightsquigarrow o$ (or $o_1 = o$) or $o_2 \rightsquigarrow o$ (or $o_2 = o$) is in σ_n, or neither is. We point out that both information flows cannot be in σ_n as then σ_n contains an object-violation, which contradicts our assumption. In the former case, assume without loss of generality that $o_1 \rightsquigarrow o$ is in σ_n. Then, by Lemma 5, $\delta(o_1) \in \Delta_o$ in the enforcement state that corresponds to σ_n. The only edge added to E between σ_n and σ_{n+1} is from s to o. As we have a new path $o_2 \rightsquigarrow o$ in σ_{n+1}, we must have $o_2 \rightsquigarrow s$ in σ_n. By Lemma 5, $\delta(o_2) \in \Delta_s$ in the enforcement state that corresponds to σ_n. This corresponds to Case (b) above.

  In the latter case that neither $o_1 \rightsquigarrow o$ and $o_2 \rightsquigarrow o$ is in σ_n, both paths are created newly in

σ_{n+1} by the addition of the edge from s to o in E. Therefore, in σ_n, $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$, and by Lemma 5, in the enforcement state that corresponds to σ_n, $\{\delta(o_1), \delta(o_2)\} \subseteq \Delta_s$. This is Case (a) above.

□

3.4.2 Least-Restriction

In this section, we show that our enforcement mechanism is least-restrictive.

Lemma 6 *Let $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ be a sequence of authorization states that is allowed by our enforcement mechanism. In the enforcement state that corresponds to σ_n, suppose $d_i \in \Delta_v$ for $v \in V$. Then, either $v \in O$ and $d_i = \delta(v)$, or there exists some $o \in O$ with $\delta(o) = d_i$ and an information flow $o \rightsquigarrow v$ in σ_n.*

PROOF. We use induction on $n \geq 1$. For the base case, $n = 1$, we know that the state-change $\sigma_0 \to \sigma_1$ is the creation of a subject or object. If a subject s is created, then the enforcement state $\mathcal{V} = \{\Delta_s\}$ where $\Delta_s = \emptyset$, and the assertion is trivially true. If an object o is created, then there is only one entry $d_i \in \Delta_o$ where $d_i = \delta(o)$, and the assertion is true. For the step, we make the induction assumption that the assertion is true for all sequences with $n \leq k$. For the case that $n = k + 1$, we do a case-analysis on the kind of state-change $\sigma_k \to \sigma_{k+1}$. We adopt the notation $\mathcal{V}_j$ for the enforcement state that corresponds to the state σ_j.

If the state-change is the creation of a subject, then $d_i \in \Delta_v$ in $\mathcal{V}_{k+1}$ implies $d_i \in \Delta_v$ in $\mathcal{V}_k$, and we rely directly on the induction assumption for the proof. If it is the creation of an object o, then only $\delta(o) \in \Delta_o$ is a new entry in any Δ_v between $\mathcal{V}_k$ and $\mathcal{V}_{k+1}$. In this case, with the induction assumption, we have the proof. If it is a read by subject s on object o, only Δ_s changes between $\mathcal{V}_k$ and $\mathcal{V}_{k+1}$. For all others, the induction assumption provides the proof. Let d_i be a new entry in Δ_s. Then, we know that $d_i \in \Delta_o$ in $\mathcal{V}_k$. By the induction assumption, either $d_i = \delta(o)$, or there is information flow $o' \rightsquigarrow o$ in σ_k where $\delta(o') = d_i$. The addition of the edge from o to s into E results in information flows $o \rightsquigarrow s$ and $o' \rightsquigarrow s$, and hence the proof.

If it is a write by subject s on object o, only Δ_o changes between $\mathcal{V}_k$ and $\mathcal{V}_{k+1}$. For all others, the induction assumption provides the proof. Let d_i be a new entry in Δ_o. Then, we know that $d_i \in \Delta_s$ in $\mathcal{V}_k$. By the induction assumption there is information flow $o' \rightsquigarrow s$ in σ_k where $\delta(o') = d_i$. The addition of the edge from s to o into E results in information flows $s \rightsquigarrow o$ and $o' \rightsquigarrow o$, and hence the proof.

□

Lemma 7 *Let $\sigma_0 \to \sigma_1 \to \ldots \to \sigma_n$ be a sequence of authorization states that is allowed by our enforcement mechanism. Let $\sigma_n \to \sigma_{n+1}$ be a state-change that is denied by our enforcement*

mechanism. If the state-change from σ_n to σ_{n+1} is a read, then allowing σ_{n+1} results in a subject-violation of the Chinese Wall security policy. If the state-change from σ_n to σ_{n+1} is a write, then allowing σ_{n+1} results in an object-violation of the Chinese Wall security policy.

PROOF.

By Theorem 8, we know that σ_n does not violate the policy.

- Subject-violation: Let the state-change $\sigma_n \to \sigma_{n+1}$ be an attempt by a subject s to read an object o for which the "if" condition in READ evaluates to true. Let d_i, d_j be a pair that causes the "if" condition to evaluate to true. One of the following two cases is true for the enforcement state that corresponds to σ_n. (1) $d_i \in \Delta_s$ and $d_j \in \Delta_o$, or, (2) $\{d_i, d_j\} \subseteq \Delta_o$. We point out that the case $\{d_i, d_j\} \subseteq \Delta_s$ in the enforcement state that corresponds to σ_n cannot happen. The reason is that if that is the case, then there exist information flows $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ in σ_n with $\delta(o_1) = d_i$ and $\delta(o_2) = d_j$. This is a subject-violation of the policy, and by Theorem 8, this cannot happen.

  If case (1) is true, then by Lemma 6, there exists $o_1 \in O$ such that $d_i = \delta(o_1)$ and the information flow $o_1 \rightsquigarrow s$ exists in σ_n. Also, there exists o_2 such that $d_j = \delta(o_2)$ and either $o_2 = o$ or there exists information flow $o_2 \rightsquigarrow o$ in σ_n. The addition of the edge from o to s would result in information flows $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$, which results in a subject-violation.

  In case (2), we know from Lemma 6 that there exist $o_1, o_2 \in O$ such that there exist information flows $o_1 \rightsquigarrow o$ and $o_2 \rightsquigarrow o$ in σ_n, or one of o_1, o_2 is the same as o. Thus, the addition of the edge from o to s to E results in information flows $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ which results in a subject-violation.

- Object-violation: Let the state-change $\sigma_n \to \sigma_{n+1}$ be an attempt by a subject s to write an object o for which the "if" condition in WRITE evaluates to true. Let d_i, d_j be a pair that causes the "if" condition to evaluate to true. One of the following two cases is true for the enforcement state that corresponds to σ_n. (1) $d_i \in \Delta_s$ and $d_j \in \Delta_o$, or, (2) $\{d_i, d_j\} \subseteq \Delta_s$. We point out that the case $\{d_i, d_j\} \subseteq \Delta_o$ in the enforcement state that corresponds to σ_n cannot happen. The reason is that if that is the case, then there exist information flows $o_1 \rightsquigarrow o$ (or $o_1 = o$) and $o_2 \rightsquigarrow o$ (or $o_2 = o$) in σ_n with $\delta(o_1) = d_i$ and $\delta(o_2) = d_j$. This is an object-violation of the policy, and by Theorem 8, this cannot happen.

  If case (1) is true, then by Lemma 6, there exists $o_1 \in O$ such that $d_i = \delta(o_1)$ and the information flow $o_1 \rightsquigarrow s$ exists in σ_n. Also, there exists o_2 such that $d_j = \delta(o_2)$ and either $o_2 = o$ or there exists information flow $o_2 \rightsquigarrow o$ in σ_n. The addition of the edge from s to o would result in information flows $o_1 \rightsquigarrow o$ and $o_2 \rightsquigarrow o$, which results in an object-violation.

  In case (2), we know from Lemma 6 that there exist $o_1, o_2 \in O$ such that there exist information flows $o_1 \rightsquigarrow s$ and $o_2 \rightsquigarrow s$ in σ_n. Thus, the addition of the edge from s to o to E results in information flows $o_1 \rightsquigarrow o$ and $o_2 \rightsquigarrow o$ which results in an object-violation.

□

Theorem 9 (Least-restriction) *A sequence of authorization states $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_n$ is allowed by our enforcement mechanism if and only if σ_n does not violate conflict of interest.*

PROOF. The "only if" part is exactly Theorem 8. The "if" part is exactly Lemma 7. □

3.4.3 Other Axes

In the previous sections, we show that our enforcement mechanism is sound and least-restrictive. In this section, we discuss how our enforcement mechanism does along the other axes from Section 3.3.1.

Time-efficiency Our enforcement mechanism is constant in the number of objects and subjects that exist. It is polynomial (at worst quadratic) in the number of company datasets, which we can think of as constant in the number of subjects and objects that exist.

Space-efficiency The size of the enforcement state is $O(|V_{\mathit{exists}}| \times |D|)$, where V_{exists} is the set of subjects and objects that exist, and D is the set of company datasets. If we assume that the number of company datasets is constant, then the space we need is linear in the number of subjects and objects that exist.

Independence of subjects In our enforcement mechanism, the actions of a subject can constrain the actions of another subject. This can be seen as a limitation of our enforcement mechanism. However, as we express in Theorem 10 below, this is unavoidable for any enforcement mechanism that is least-restrictive. It may be possible to characterize the "degree" of independence of subjects; this is beyond the scope of our work.

Theorem 10 *Let $\mathcal{A}$ be an enforcement mechanism. If $\mathcal{A}$ is least-restrictive, then subjects are not independent. That is, the actions of a subject can constrain the prospective actions of other subjects.*

PROOF. To establish the theorem, we provide a counterexample to the opposite assertion. In the context of Definition 22 from Section 3.3.1, the sequence Ψ_a is the creation of subjects s_1, s_2 and objects o_1, o_2, o_3 with $C(o_1, o_2) = 1$ and $C(o_1, o_3) = C(o_2, o_3) = 0$. This is followed by a read of o_1 by s_1, write of o_3 by s_1 and read of o_2 by s_2. We point out that the authorization state that results (starting at σ_0) does not violate conflict of interest, and therefore is reachable under $\mathcal{A}$. As Ψ_b, we pick the subsequence of Ψ_a without any of the read and write actions of s_1. The subsequent action we want to execute is a read of o_3 by s_2. $\mathcal{A}$ does not allow this action after Ψ_a as it causes a conflict of interest violation: $o_1 \rightsquigarrow s_2$ and $o_2 \rightsquigarrow s_2$. But it allows this action after Ψ_b, as the resulting state does not violate conflict of interest. □

Access to sanitized objects Our enforcement mechanism places no restrictions on access to sanitized objects so long as those objects are not used as conduits to a policy violation.

Transitivity of the conflict of interest relation The conflict of interest relation does not have to be transitive in our enforcement mechanism.

3.4.4 Uniqueness of Our Enforcement Mechanisms

One may ask whether ours is the only possible enforcement mechanism that is least-restrictive. While we are unable to prove such a strong result, we assert in the following theorem that any enforcement mechanism that is least-restrictive must deny *some* attempt by a subject to read an object to prevent subject-violations, and some attempt by a subject to write an object to prevent object-violations. This validates our design choice of mediating read attempts to prevent subject-violations, and mediating write attempts to prevent object-violations.

Theorem 11 *Let $\mathcal{A}$ be an enforcement mechanism for the Chinese Wall security policy that is least-restrictive. Then $\mathcal{A}$ must deny some attempt by a subject to read an object to prevent subject-violations, and must deny some attempt by a subject to write to an object to prevent object-violations.*

PROOF.

- Subject-violations: Assume otherwise, for the purpose of contradiction. That is, $\mathcal{A}$ does not deny any attempt to read. Let $\sigma_0 \to \ldots \to \sigma_n$ be a sequence of states each reachable from the previous by a state-change such that we have a subject-violation in σ_n, but not in any of the prior states. We make two observations: (1) in σ_n, all information flows because of which the policy is violated are into a particular subject s, and, (2) the state-change $\sigma_{n-1} \to \sigma_n$ must be a read by s that causes there to exist information flows of the form $o_1 \rightsquigarrow s$, $o_2 \rightsquigarrow s$, $\ldots$, $o_m \rightsquigarrow s$ such that $m \geq 2$ and $C(\delta(o_i), \delta(o_j)) = 1$ for all $i, j \in [1, m], i \neq j$. To prevent such a sequence of states from occurring without denying any read in the sequence, either (a) the creation of a subject or object, or, (b) the write by a subject s' on an object o' must be denied. Let the state-change $\sigma_{k-1} \to \sigma_k$ be such a state-change that is denied. Then, by (2) above, $k < n$, because the state-change $\sigma_{n-1} \to \sigma_n$ is a read. Therefore, by (1) above, the sequence $\sigma_0 \to \ldots \to \sigma_k$ contains only those states that do not contain a subject-violation. As that sequence is precluded by $\mathcal{A}$, this contradicts the assumption that $\mathcal{A}$ is least-restrictive.

- Object-violations: Assume otherwise, for the purpose of contradiction. That is, $\mathcal{A}$ does not deny any attempt to write. Let $\sigma_0 \to \ldots \to \sigma_n$ be a sequence of states each reachable from the previous by a state-change such that we have an object-violation in σ_n, but not

in any of the prior states. We make two observations: (1) in σ_n, all information flows because of which the policy is violated are into a particular object o, and, (2) the state-change $\sigma_{n-1} \rightarrow \sigma_n$ must be a write to o that causes there to exist information flows of the form $o_1 \rightsquigarrow o$, $o_2 \rightsquigarrow o$, $\ldots$, $o_m \rightsquigarrow o$ such that $m \geq 2$ and $C(\delta(o_i), \delta(o_j)) = 1$ for all $i, j \in [1, m], i \neq j$. To prevent such a sequence of states from occurring without denying any write in the sequence, either (a) the creation of a subject or object, or, (b) the read by a subject s' on an object o' must be denied. Let the state-change $\sigma_{k-1} \rightarrow \sigma_k$ be such a state-change that is denied. Then, by (2) above, $k < n$, because the state-change $\sigma_{n-1} \rightarrow \sigma_n$ is a write. Therefore, by (1) above, the sequence $\sigma_0 \rightarrow \ldots \rightarrow \sigma_k$ contains only those states that do not contain an object-violation. As that sequence is precluded by $\mathcal{A}$, this contradicts the assumption that $\mathcal{A}$ is least-restrictive.

□

3.5 The Brewer-Nash Approach

In this section, we present new results on the Brewer-Nash enforcement mechanism. We identify the enforcement state that their approach needs to maintain, and express the simple security and $*$-rules, and the corresponding rules for read and write as algorithms.

Enforcement state We can directly infer the enforcement state that needs to be maintained from the simple-security and $*$-rules. To assess whether the simple-security rule is met when a subject s requests access to an object o, we need to know whether an object from $\delta(o)$ has already been accessed, and whether given any $o' \in O$ already accessed by s, $C(\delta(o'), \delta(o)) = 0$. In addition, for the $*$-rule, we need to know whether the Company Dataset for any object that can be read by s is different from $\delta(o)$.

Consequently, these rules can be applied if we maintain: (a) for each subject that exists, the set of Company Datasets of objects it has already accessed (read or written), and, (b) the set of objects that exist. (For (a), we can alternately maintain the set of Company Datasets from which no object has been accessed by each subject.)

As for our enforcement mechanism, we represent the enforcement state as $\mathcal{V}$, which comprises sets $\Delta_s \subseteq D$ of Company Datasets, where the subscript s is the subject with whom Δ_s is associated.

Algorithm We now present the algorithm for the Brewer-Nash enforcement mechanism as sub-routines for read and write attempts, READ-BN and WRITE-BN, respectively. We omit the ones for creation and destruction — those involve straightforward updates to the enforcement state, similar to our enforcement mechanism. We present the simple-security and $*$-rules as sub-routines invoked by READ-BN and WRITE-BN. (We number the lines of SIMPLE-SECURITY and $*$ below as we need to refer to them in our proofs in the next section.)

SIMPLE-SECURITY(s, o)

if $\delta(o) \in \Delta_s$ **then return** true;
else if $\exists o'$ *with* $\delta(o') \in \Delta_s$ *and* $C(\delta(o), \delta(o')) = 1$ **then return** false;
else return true;

$*(s, o)$

if *SIMPLE-SECURITY*$(s, o) =$ *false* **then return** false;
else if $\exists o'$ *with SIMPLE-SECURITY*$(s, o') =$ *true,* $\delta(o) \neq \delta(o')$*, and* $\delta(o') \neq D_{[s]}$ **then return** false;
else return true;

READ-BN(s, o)

if *exists*(o) *and SIMPLE-SECURITY*(s, o) **then**
$\Delta_s \leftarrow \Delta_s \cup \{\delta(o)\}$; allow;
else deny;

WRITE-BN(s, o)

if *exists*(o) *and* $*(s, o)$ **then**
$\Delta_s \leftarrow \Delta_s \cup \{\delta(o)\}$; allow;
else deny;

Evaluation The Brewer-Nash enforcement mechanism is sound — if a request is allowed, we are guaranteed that the policy is not violated in the subsequent state.

The enforcement mechanism is not least-restrictive. For example, as we mention in Section 3.1, the authorization state that corresponds to Figure 3.1 is not reachable under Brewer-Nash even though it does not violate the Chinese Wall security policy. At time 2, the subject s_2 would have been denied the ability to write to o_4 having the permission of reading o_2. $*$-rule prevents s_2, because when s_2 wants to write to o_4, he has the ability of reading o_2 that is in a different conflict of interest class from o_4.

Although the original exposition of Brewer-Nash [17] assumes that the conflict of interest relation is transitive, there is nothing in their enforcement mechanism that requires this, as is demonstrated above by our exposition.

Under Brewer-Nash, subjects are independent; that is, the actions of a subject do not constrain the prospective actions of another subject. (By Theorem 10 in Section 3.4.3, this is another way to intuit that Brewer-Nash is not least-restrictive.) It is space-efficient — we maintain only $O(|S_{exists}| \cdot |D| + |O_{exists}|)$ amount of space, where S_{exists} and O_{exists} are the sets of subjects and objects that *exists*$(S) = 1$, and *exists*$(O) = 1$ respectively. It is less time-efficient than our enforcement mechanism, but still polynomial-time in the number of objects that exist. The algorithm SIMPLE-SECURITY, and therefore READ-BN, runs in time $O(|O_{exists}| \cdot |D|)$, and $*$, and therefore WRITE-BN, runs in time $O(|O_{exists}|^2 \cdot |D|)$.

3.5.1 New Properties

In this section, we present new properties of the Brewer-Nash enforcement mechanism. Theorem 12 expresses that the enforcement mechanism is even more restrictive than previous work suggests [20, 21, 22]. Theorem 13 expresses that the $*$-rule is overstated.

Theorem 12 *Suppose there exists a subject s and an unsanitized object o such that s is allowed to write to o under Brewer-Nash. Then, there is exactly one Conflict of Interest class to which all unsanitized objects belong.*

PROOF. We can restate the theorem as the following: suppose there exists $s \in S$ and $o_1 \in O$ such that WRITE-BN$(s, o_1) =$ allow. Then for any $o_2 \in O$ with *exists*$(o_2) = 1$, $\delta(o_2) \neq D_{[s]}$ and $\delta(o_2) \neq \delta(o_1)$, we have $C(\delta(o_1), \delta(o_2)) = 1$.

Suppose, as in the premise, WRITE-BN$(s, o_1) =$ allow. This means that $*(s, o_1) =$ true. Therefore, $*(s, o_1)$ returns in Line 3 of that routine. Now consider any other $o_2 \in O$ with *exists*$(o_2) = 1$, $\delta(o_2) \neq \delta(o_1)$ and $\delta(o_2) \neq D_{[s]}$. We need to show that $C(\delta(o_2), \delta(o_1)) = 1$.

Suppose, for the purpose of contradiction, that $C(\delta(o_2), \delta(o_1)) = 0$. We have two cases: (a) READ-BN(s, o_2) returns allow, or, (b) READ-BN(s, o_2) returns disallow. In case (a), we have the following contradiction. We know that $*(s, o_1) =$ true, yet, the condition in Line 2 of $*(s, o_1)$ evaluates to true for $o' = o_2$. The reason is that SIMPLE-SECURITY(s, o_2) has to return true for READ-BN(s, o_2) to be allow.

In case (b), we know that SIMPLE-SECURITY$(s, o_2) =$ false. Therefore, there is some o_3 that serves as the o' in Line 2 of
SIMPLE-SECURITY(s, o_2). We now consider the following sub-cases. (i) $o_3 = o_1$. This results in a contradiction because by Line 2 of SIMPLE-SECURITY(s, o_2), $C(\delta(o_3), \delta(o_2)) = 1$, but we have assumed that $C(\delta(o_1), \delta(o_2)) = 0$. (ii) $o_3 \neq o_1$, but $\delta(o_3) = \delta(o_1)$. This results in the same contradiction as (i). And finally, (iii) $o_3 \neq o_1$ and $\delta(o_3) \neq \delta(o_1)$. Now, because o_3 serves as the o' in Line 2 of SIMPLE-SECURITY(s, o_2), we know that $\delta(o_3) \in \Delta_s$. This means that READ-BN$(s, o_3) =$ allow. Now consider an invocation of $*(s, o_1)$. The "if" condition in Line 2 evaluates to true for $o' = o_3$, and the invocation returns false. But this contradicts the assumption that WRITE-BN(s, o_1) returns allow. □

Before we present the next theorem, we restate the $*$-rule. "Write access is only permitted if (a) access is permitted by the simple security rule, and, (b) no object can be read which is in a different Company Dataset to the one for which write access is requested and contains unsanitized information."

Theorem 13 *If a subject s and object o satisfy subrule (b) of the $*$-rule of Brewer-Nash, then they satisfy subrule (a) (the simple-security rule).*

PROOF.

We can restate the theorem as follows: let $s \in S$ and $o \in O$. Assume that $\nexists o'$ such that READ-BN$(s, o') =$ allow, $\delta(o) \neq \delta(o')$ and $\delta(o') \neq D_{[s]}$. Then, SIMPLE-SECURITY$(s, o) =$ true.

Given our assumption, we know that for all $o' \neq o$, at least one of the following is true. (a) READ-BN$(s, o') =$ deny, (b) $\delta(o') = \delta(o)$, or, (c) $\delta(o') = D_{[s]}$. For the purpose of contradiction, assume that SIMPLE-SECURITY(s, o) = false. This means that the "if" condition in Line 2 of SIMPLE-SECURITY(s, o) evaluates to true. That is, there exists some o'' that serves as the o' in Line 2. Specifically, $\delta(o'') \in \Delta_s$.

Such an o'' cannot serve as o' in (c) above, as then, $C(\delta(o''), \delta(o)) = 0$. Suppose it serves as the o' in (b) above. Then,
SIMPLE-SECURITY(s, o) = true from Line 1; a contradiction. Finally, suppose o'' serves as the o' in (a) above. But, in the last sentence of the previous paragraph, we asserted that $\delta(o'') \in \Delta_s$. Therefore, READ-BN$(s, o'') =$ allow, a contradiction. ☐

3.5.2 Bishop's Rendition [32]

Bishop [32] reproduces the work of Brewer and Nash in his popular book on computer security. However, perhaps for brevity of exposition, he treats sanitized objects differently. In Bishop's rendition, D, the set of Company Datasets does not contain the special Dataset $D_{[s]}$ for sanitized objects, but only a Dataset for each company, $D_1, \ldots, D_k$. Every object also has what one can think of as a one-bit labelling function, l: $O \longrightarrow \{u, \overline{u}\}$, which indicates whether it is unsanitized or sanitized. The simple security rule is amended to the following.

SIMPLE-SECURITY-BISHOP(s, o)

if $l(o) = \overline{u}$ **then return** true;
else return SIMPLE-SECURITY(s, o);

This somewhat subtle difference in expressing the simple-security rule has fairly significant consequences. A system that adopts
SIMPLE-SECURITY-BISHOP instead of SIMPLE-SECURITY does not behave as it would under the Brewer-Nash rules. Specifically, Theorem 13 is no longer necessarily true.

We can show this with a counterexample. Suppose we have a subject s and two objects o_1 and o_2. The objects belong to different Company Datasets, D_1 and D_2, respectively. The two Company Datasets are in conflict; i.e., $C(D_1, D_2) = 1$. The object o_1 is sanitized, and o_2 is unsanitized. The subject s has read the object o_1 at time t.

We observe that SIMPLE-SECURITY-BISHOP(s, o_2) is false. The reason is that the "if" condition in Line 2 of SIMPLE-SECURITY, which is invoked by SIMPLE-SECURITY-BISHOP, is met for $o' = o_1$, and therefore it returns false in Line 2. However, subrule (b) of the $*$-rule is trivially true.

Consequently, satisfaction of subrule (b) of the $*$-rule does not imply satisfaction of subrule (a) in Bishop's rendition, which is what Theorem 13 asserts about Brewer-Nash.

This suggests something good about Bishop's enforcement mechanism when compared to the Brewer-Nash enforcement mechanism: the $*$-rule is not overstated. However, a closer examination of our counterexample shows a somewhat peculiar behaviour by the system. Suppose s has not yet read o_1. Then, s is allowed to read o_2. And, s may later read o_1 as well, because $l(o_1) = \overline{u}$ and therefore SIMPLE-SECURITY-BISHOP is always true for it. In other words, the order in which s reads sanitized and unsanitized objects matters. This is not the case with the Brewer-Nash enforcement mechanism.

3.6 Related Work

There is a large body of work that pertains to the Chinese Wall security policy [18]. Our work pertains to enforcement of the policy. As such, it is most closely related to the original work of Brewer and Nash [17], and the subsequent work of Lin [34], Kessler [20] and Sandhu [21, 22]. We discuss those pieces of work first, and then sample other work. We have discussed the work of Brewer and Nash [17] extensively in this dissertation, and therefore do not discuss it any further.

The work of Lin [34] points out that Brewer-Nash assumes that the conflict of interest relation is transitive, but it does not have to be. We have adopted his generalization in this work.

The work of Kessler [20] makes observations about how restrictive the Brewer-Nash enforcement mechanism can be (see Section 3.1). That work then proposes an alternate enforcement mechanism that is purportedly less restrictive. It does not consider the notion of least-restrictive enforcement.

Furthermore, Kessler [20] precludes two subjects from being able to write to the same object. Specifically, once a subject writes to an object, no other subject can write to the same object. We point out that two subjects could read and write a single object only, and thereby not violate the Chinese Wall security policy.

This is an aspect with which Kessler's approach cannot be said to be less restrictive than Brewer-Nash. That is, there exist authorization states that satisfy the Chinese Wall security policy that are unreachable under Kessler's approach that are reachable under Brewer-Nash. And, there exist other authorization states that satisfy the policy that are unreachable under Brewer-Nash that are reachable under Kessler's approach.

Sandhu's work [21, 22] also observes that the Brewer-Nash enforcement mechanism is restrictive (see Section 3.1). It also points out some confusion in the work of Brewer and Nash [17] regarding subjects, as opposed to users and principals. We have incorporated Sandhu's mindset in our work. Sandhu's work [21, 22] also refutes an assertion of Brewer and Nash [17] that the Chinese Wall security policy cannot be encoded in a lattice. That work proposes a construction, which we can perceive as an enforcement mechanism for the policy.

In Sandhu's enforcement mechanism, every object and subject is associated with a label from a lattice. The enforcement mechanism then simply uses the "no read up" and "no write down" rules that is customary for lattice-based access control systems. In Sandhu's enforcement mechanism, subjects are independent, and this immediately tells us that the enforcement mechanism is not least-restrictive (see Theorem 10 in Section 3.4.3). Furthermore, as a subject is bound to a label in the lattice, if an object belongs to only one Company Dataset (as is the case in the original specification of Brewer and Nash [17]), a subject that can read objects from two Company Datasets cannot write to any object. The reason is that such a subject must be at least one level above all objects in the lattice, and by the "no write down" rule, it cannot write any object. (We discuss below, however, that Sandhu's work [21, 22] proposes the generalization that an object can be labelled with multiple Company Datasets from different Conflict of Interest classes.) We point out also that Sandhu's enforcement mechanism requires the Conflict of Interest relation to be transitive.

We now sample other work on the Chinese Wall security policy with which our work is not related closely. Our intent is to give a somewhat broad overview of the kinds of work related to the Chinese Wall security policy that exists.

There has been work on realizing the Chinese Wall security policy in specific contexts. Recent work [15, 16] proposes the application of the policy to cloud computing environments. Tsai et al. [15] propose to use the Chinese Wall security policy to resist particular kinds of attacks in the context of cloud-computing, such as what they call an inter-Virtual Machine (VM) attack. They give an algorithm based on graph-coloring to assign VMs that do not belong to the same Conflict of Interest class to the same physical machine, and then use the Brewer-Nash enforcement mechanism to prevent such attacks. Wu et al. [16] propose to use the Brewer-Nash enforcement mechanism as an Infrastructure-as-a-Service (IaaS) in cloud-computing environments, and provide an implementation. Similar earlier work on applying the Chinese Wall security policy to specific contexts are those of Loscocco and Smalley [41], Jajodia et al. [42], and Edjlali et al. [43].

Other work, such as that of Atluri et al. [14] and Sobel and Alves-Foss [44] generalizes the Brewer-Nash policy. The work of Sandhu [21, 22] also proposes the generalization that objects be allowed to be associated with multiple Company Datasets from different Conflict of Interest classes. Such generalizations are certainly interesting, but are beyond the scope of this work. We consider only the original setup of Brewer and Nash [17], and leave dealing with such generalizations for future work.

Finally, there is work that attempts to reconcile or model the simple-security and $*$-rules of Brewer-Nash within it. An example of such work is that of Fong [45]. Such work is relevant to enforcement. However, it does not consider the issues our work addresses — that of least-restrictive enforcement of the policy.

3.7 Conclusion

We have addressed enforcement of the Chinese Wall security policy. Specifically, we have proposed a notion of least-restrictive enforcement of the policy, devised an enforcement mechanism and shown that it is least-restrictive. Our enforcement mechanism is simple, sound and efficient. Our enforcement mechanism mediates read attempts only to prevent subject-violations, and write attempts only to prevent object-violations.

We have precisely identified the trade-off our enforcement mechanism incurs in achieving least-restriction. The actions of a subject may constraint the prospective actions of other subjects, a property that we call (lack of) independence of subjects. We have established the somewhat strong result that *any* enforcement mechanism to enforcing the Chinese Wall security policy that is least-restrictive must incur this trade-off.

We have then established two new results for the Brewer-Nash enforcement mechanism to enforcement [17]. We have shown that it is more restrictive than previous work points out. We have shown that if a subject is allowed to write to an object, then all objects must belong to the same Conflict of Interest class. We have shown also that the so-called $*$-rule that is part of the Brewer-Nash enforcement mechanism to enforcement is overstated in that one of its sub-rules implies the other. We have also investigated the rendition of the Brewer-Nash enforcement mechanism in Bishop's work [32], and shown that it is not equivalent to the Brewer-Nash enforcement mechanism.

In summary, our work establishes new results, and thereby sheds new light on what is generally considered to be important work in information security.

In our discussion on related work, we observe that subsequent enforcement mechanisms of Kessler [20] and Sandhu [21, 22] are also restrictive in their own ways, and not least-restrictive.

We see applications of our enforcement mechanism as the most promising future work. Specifically, we seek to investigate the problems in cloud-computing that prior work such as those of Tsai et al. [15] and Wu et al. [16] have proposed, and ask whether the least-restriction that our enforcement mechanism provides is useful and meaningful in those contexts.

Another direction of future work is to investigate extensions that have been proposed to the policy in work such as those of Atluri et al. [14] and ask how we can extend our enforcement mechanism to deal with such extensions. We seek also to rigorously establish that other enforcement mechanisms such as those of Kessler [20] and Sandhu [21, 22] are indeed incomparable to the Brewer-Nash enforcement mechanism from the standpoint of reachable authorization states.

Chapter 4

Property-Testing Real-World Authorization Systems

4.1 Introduction

Authorization deals with the specification and management of accesses principals have to resources. It is an important aspect of security. We address the problem of testing implementations of authorization systems for properties that are of interest.

Authorization systems can be complex; this is the underlying technical challenge in our work. The complexity arises from a feature that realistic authorization systems have: it is possible to not only directly authorize a principal to a resource, but also to do so indirectly. For example, a principal may acquire some rights via membership in groups or roles. Similarly, if the principal is authorized to execute a program, in the context of that program she may have some additional rights.

The reason such indirect ways are allowed is to balance the scalability needs of enterprise authorization systems, and security. Enterprise authorization systems must typically support large numbers of principals (e.g., users) and resources (e.g., files and documents). Providing indirect authorizations eases administrative burden. Granting a right to a group, for example, is less burdensome than granting it to each member of the group.

Given an implementation of an authorization system, one may want to test it. The objective is to ascertain whether the implementation has certain security and availability properties of interest. For example, in an authorization system that supports the notion of groups, we may want to ask, "if Alice is a member of the group G at the time G is authorized to read file f, she should be authorized to read file f at that time." The two direct authorizations in the example refer to Alice's membership in G, and G's authorization to f. (This is a special case of what we call the forward availability property — see Section 4.4.1).

Prior foundational work suggests that the problem of verifying that an authorization system has such a property may be intractable or even undecidable [24, 25, 26]. This is the case even

if we want to carry out such verification "on paper," i.e., considering only the design of the system, and not any implementation. Clearly, such limits on the tractability of verification apply to testing as well. Nonetheless, we argue that the problem we address is an important one. If an implementation does not have a property we seek, we either need to redesign it, or address some implementation bug that causes it to not have the property.

Furthermore, in testing, it is customary to make the *small-scopes assumption* [46], with which we bound the scope of the system that is tested. The manner in which we apply the small-scopes assumption is to limit the system under test to contain only a few entities (e.g., principals, resources, and groups). The mindset behind the small-scopes assumption is that a system has a property we seek if and only if it has the property under the small-scopes assumption.

We argue that customary software testing is not geared to the kinds of properties we address in this work, that are specialized to authorization systems. This is not surprising — a similar observation has been made in other contexts in security, for example, testing for bugs in cryptographic implementations [47].

However, we do borrow ideas from software testing, for example, the small-scopes assumption which we discuss above. Similarly, the use of model-checking for software testing [48] and the use of traces to verify models "on paper" [49, 50] have been explored in recent work. Our work also has similarities to penetration testing in the context of security, in particular, the flaw-hypothesis methodology [51]. We discuss these in Section 4.5 on related work.

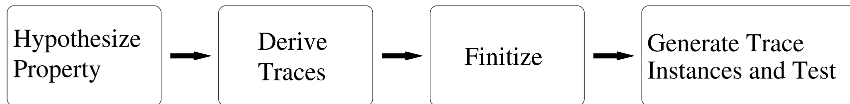

Figure 4.1: The four stages in our testing methodology.

Our work We first propose and adopt a 4-stage process for property-testing an implementation of an authorization system [23]. We show the stages in Figure 4.1.

1. In the first stage, we articulate properties of interest. A property typically balances generality and specificity. The specificity comes from the authorization features that the system under test supports. For example, one of the systems to which we have applied our methodology, the $\mathcal{I}$ System (see Section 4.4), does not have notions of executable programs or scripts via which a principal acquires rights. Consequently, some of the properties we articulate for the $\mathcal{D}$ System (see Section 4.3.1) are not meaningful for the $\mathcal{I}$ System.

   Particularly from a research-standpoint, properties that we anticipate are relevant not only to the two systems we discuss in this work, but to large classes of authorization systems.

   Another issue regards the syntax in which we express properties. We have adopted existential second-order temporal logic*[52] in this work. The reason is that it gives us an elegant

*We recognize that "temporal" is redundant once we specify "existential second-order." We include that term to emphasize that some of the properties we consider have temporal aspects.

syntax and an associated precise semantics that is sufficient for us to succinctly express the properties we consider. It is conceivable that a different syntax is more natural to express other properties.

2. A property is a declarative specification. A system under test can be more naturally seen as a procedural entity. To test for a property, therefore, we introduce the notion of a trace. A trace is a sequence of authorization states and the state-change from one to the next.

   We relate properties to traces: we identify traces that should exist, or should not exist in the system for a property to hold. That is, we establish assertions of the form, "property p holds if and only if no trace of type t is generated by the system."

3. We make finitizing (i.e., small-scopes) assumptions that are realistic. That is, we assume that certain parameters are bounded by constants. This assumption is crucial to the practicality of our approach. In Theorem 14, we establish that verifying a property under the kind of finitizing and other assumptions about predicates and their arity that we make, is in **PSPACE**.

   This implies that we can employ an approach similar to model-checking [53, 54]. We point out that for a particular property, the problem is not necessarily **PSPACE**-complete. For example, the verification problem for all the three properties we consider for the $\mathcal{D}$ System are in co-**NP**. **PSPACE** is an upper-bound only.

4. Based on the finitization, we identify instances of traces for which we need to check in the system under test. We then exercise an implementation and check whether the instances of traces are generated, or not generated by the implementation. We use a model-checker as the trace-instance-generator (see Section 4.2.2).

We have designed a testing system based on the four stages above, and built portions of it. We discuss it in Section 4.2.2. We have used it to apply the above methodology to two real-world systems. One, which we call the $\mathcal{D}$ System, comprises components of the authorization system built into a commercial database system. The other comprises portions of the authorization system that underlies a commercial system for controlled file sharing that we call the $\mathcal{I}$ System. We have been in communication with the vendors of both systems, and both have requested us to use pseudonyms for their systems.

In the $\mathcal{D}$ System, we have discovered flaws in the authorization system related to procedures that aggregate SQL statements, and make calls to other procedures. To our knowledge, these have not been presented before in the literature. We have discussed our findings with the vendor, and discuss their response in Section 4.3.5.

The kinds of issues we identify are very different from well-known SQL injection and buffer overflow vulnerabilities that have plagued database systems [55, 56]. We discuss the properties and our findings in Section 4.3.4.

For the $\mathcal{I}$ System, we have verified that it has several properties. The vendor has confirmed that the establishment of these properties is indeed of interest to them, and consistent with their intended design. The kinds of properties we discuss for the systems are qualitatively different.

Operator	Read as	Explanation
X	Next	(X p) means that formula p will hold in the next state.
F	Eventually	(F p) means that formula p will eventually hold in the future.

Table 4.1: Temporal operators that we use.

We choose to present these properties in this dissertation to demonstrate that a good range of properties of interest can be tested for.

4.2 More Details on the Testing Methodology

In this section, we discuss our methodology in more detail, and also discuss the testing system we have put together based on it. The first stage in our methodology is to articulate properties. Our choice of syntax is existential second-order temporal logic [52].

Authorization system To articulate properties of interest, we first need an abstraction for an authorization system. We discuss the basic elements of such an abstraction here. Particular systems may extend these, as we need in the $\mathcal{D}$ System (Section 4.3). As is customary in temporal logic, the system progresses in what are called *time steps*. A system is as follows.

- Time progresses only as a consequence of actions.
- In a time step, exactly one action occurs.
- There are two possible actions:
  - "x is directly authorized to y" and,
  - "x is directly unauthorized from y".
- With each action, we associate a predicate. We call such predicates *action-predicates*. We have the following two kinds of action-predicates that correspond to the above two kinds of actions, respectively.
  - Authorization action: $\mathsf{dAuth}(x, y)$
  - Unauthorization action: $\mathsf{dUnauth}(x, y)$

  The argument x is the entity being authorized or unauthorized, and y is the entity to which it is authorized or from which it is unauthorized. A direct authorization of x to y may be, for example, a group x being authorized to a file y, or a principal x being added to the group y.

It may seem surprising that we consider only an authorization action, and do not qualify it as, for example, an authorization to read or write. For the properties we consider, it turns out that this suffices. As we mention earlier, we can extend our basic abstraction if necessary.

- An action predicate is true in a particular time step if the particular action is effected in that time step. Consequently, we adopt the following rule with regards to the above predicates. We use "$\rightarrow$" for implication and "$\neg$" for negation.

$$\mathsf{dAuth}(x, y) \longleftrightarrow \neg\mathsf{dUnauth}(x, y)$$

- In addition to action predicates, we have the following *non-action predicate*: $\mathsf{auth}(x, y)$, which indicates that x is authorized to y.

Semantics We endow a semantics to auth, dAuth and dUnauth by specifying a model [52]. We omit the details here and use a somewhat informal notation. To indicate that an authorization of x to y is part of the authorization state, we simply write $\mathsf{auth}(x, y)$; we write $\neg\mathsf{auth}(x, y)$ to indicate that it is not. Whether x and y are abstract arguments or concrete values is clear from context (e.g. x is a group of users and y is a file). A state-change is associated with a predicate dAuth or dUnauth that becomes true as a consequence of the action. To denote a state-change on x to y, we simply write $\mathsf{dAuth}(x, y)$ or $\mathsf{dUnauth}(x, y)$ respectively.

Authorization state An authorization state is a family of sets, one for each predicate p. The set that corresponds to a particular predicate p contains those tuples $\langle x_1, \ldots, x_k \rangle$ for which $p(x_1, \ldots, x_k)$ is true, where we assume that p's arity is k. For example, in an authorization system that has the auth, dAuth and dUnauth predicates only, a state is a collection of $\mathsf{auth}(x, y)$, $\mathsf{dAuth}(x, y)$ and $\mathsf{dUnauth}(x, y)$ that are true.

Properties and Traces A property is a formula that is specified in existential second-order temporal logic using the predicates we mention above. The temporal operators we use for the properties we consider in this work are shown in Table 4.1. A property is typically an implication of the form "$F_1 \rightarrow F_2$," where F_1, F_2 are formulas, and F_2 is typically an assertion regarding the auth predicate. We defer a discussion on specific properties to Sections 4.3 and 4.4 in which we discuss our two applications.

As we discuss in Section 4.1, we need traces as procedural specifications that relate declarative properties to the system under test. A trace is a three tuple: a start authorization state, a sequence of state changes, and a final authorization state. A trace may be parameterized. For example, we may use a parameter for a user or group that is authorized to a resource in the specification of a trace. In our third stage, we concretize traces to what we call instances of traces. In theory, a parameter could take on one of infinitely many values, and therefore there may be an infinite number of trace instances that we may associate with a trace. We finitize the trace instances that we need to consider by making assumptions about the real system.

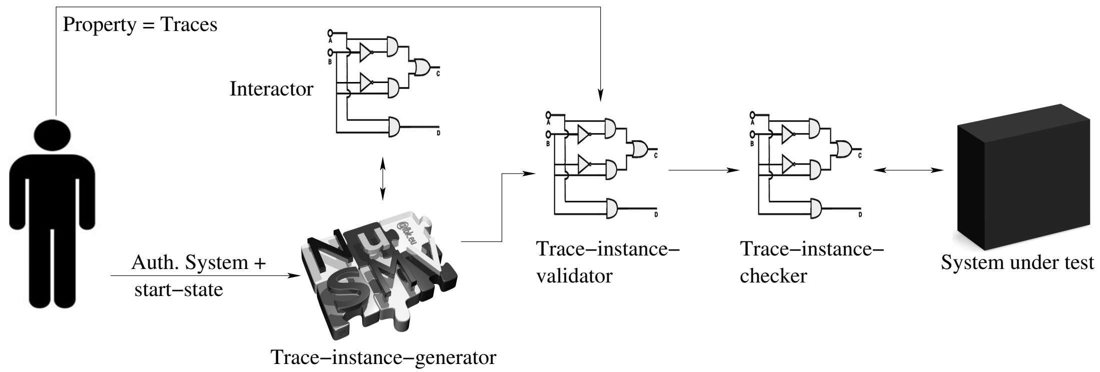

Figure 4.2: Our testing system. The "interactor," "trace-instance-validator," and "trace-instance-checker" are automata (programs or humans). In our current implementation, the interactor and the trace-instance-validator are human-driven. We discuss the components in Section 4.2.2.

4.2.1 Relationship to Model-Checking

We can perceive testing as a verification problem. The question then is whether we can place an upper-bound on the computational complexity of the problem. We point out that for the logic we adopt, verification of a property is undecidable in general. However, under assumptions that we argue are realistic for the kind of testing we do, this is not the case, as the following theorem asserts.

Theorem 14 *Let an authorization system and a property of interest be expressed using at most p predicates, each of at most constant arity, a. Also let n be the number of entities (user, groups, resources, etc.) that can exist in an authorization system. Then, whether the system has the property is decidable in **PSPACE**.*

All the assumptions in the statement of the above theorem can be thought of as finitizing assumptions. The proof of the theorem relies on the observation that to maintain the truth value of each predicate for each tuple of entities, we need $O(n^a)$ bits, because each predicate has arity at most a[†]. Therefore, each authorization state can be encoded with $O(pn^a)$ bits, which is polynomial in the size of the input, given the assumption that a is a constant.

The above theorem tells us that we can reduce our testing problem under the assumptions stated in the theorem to model-checking [53]. Indeed, our testing methodology can be seen as model-checking under those assumptions. We point out that we use a model-checker as part of our process, but in a somewhat non-standard way, as a trace-generator. We discuss this in the next section on our testing system.

[†]We use $O(\cdot)$ as is customary in computing: $f(x) = O(g(x))$ means that $g(x)$ is an asymptotic upper-bound for $f(x)$ [57].

4.2.2 A Testing System

We have designed a testing system based on the methodology we discuss in Section 4.1 and above. We show it in Figure 4.2 and discuss it in this section. Our implementation of it is not complete; however, we have implemented it sufficiently for us to carry out the applications we discuss in the next two sections.

As the figure indicates, a human user specifies the authorization system and a start-state for it. She specifies also the property of interest. The property of interest is specified as a set of traces that are sufficient and necessary for the property to hold. We derive these for particular properties for our applications in Sections 4.3 and 4.4. We use a model-checker, specifically NuSMV [58], as a generator of trace-instances. The input to NuSMV is a model (a "program," in NuSMV parlance), which encodes the authorization system and a start-state for it. The finitization assumptions are built into the model we input to NuSMV.

We then ask NuSMV to generate trace-instances for that model ("simulate," in NuSMV parlance). We do this in what NuSMV calls "interactive mode." The interactive mode guarantees that every possible trace-instance is generated. As its name suggests, its intent is for a human user to interact with NuSMV as the trace-instances are generated. This part is realized in what we call the Interactor in Figure 4.2.

One may wonder why we do not provide the property of interest also as input to NuSMV. The reason is that we want to exercise every trace-instance that pertains to the property against the system under test. NuSMV can only tell us whether the (abstract) model we provide it satisfies the property.

Each trace-instance generated by NuSMV is then validated against the traces that correspond to the property by the Trace-instance-validator in Figure 4.2. That is, we check whether a particular trace-instance that NuSMV generates is indeed an instance of a trace that pertains to the property. We then check whether each validated trace-instance is generated by the system under test. This is done by the Trace-instance-checker.

We have not yet fully automated the entire process in Figure 4.2. In particular, interaction with NuSMV and the validation done by the trace-instance-validator are performed by humans. We recognize that without fully automating the entire process, our testing system cannot scale. However, even with the current level of automation, we have been able to apply our system meaningfully (Sections 4.3 and 4.4).

The Trace-instance-checker exercises a trace on the system under test in one of two ways. For the $\mathcal{D}$ System, we can exercise a trace either programmatically, via an Application Programmer Interface (API), or by simulating the keystrokes and mouse-clicks of a human user. For the $\mathcal{I}$ System, the only interface we have been provided is the latter. To simulate a human user, we use the Robot class in Java.

4.3 Application 1: the $\mathcal{D}$ System

In this section, we discuss our application of our testing methodology to the system that we call the $\mathcal{D}$ System. The $\mathcal{D}$ System comprises components of the authorization system contained within a commercial database system. The components of the authorization system with which we deal are discretionary permissions to database tables, and permissions to procedures.

The Structured Query Language (SQL) is a command-line syntax for querying and manipulating relational database tables. An example of a SQL command is "select * from book_table where price $>$ 100." This is the equivalent of exercising a read permission on particular entries in the table called book_table.

A procedure may be used to aggregate multiple SQL statements. For example, a user Alice may create a procedure that takes as argument a new entry for a table, first checks some condition on existing entries in the table, and inserts the new entry only if the condition is met. A procedure may also invoke other procedures. In these aspects, procedures are like executable programs.

Authorization Every table is associated with an owner. Customarily, the owner is the user that creates the table. The owner may then, at her discretion, grant permissions over the table to other users. In Figure 4.3, for example, a user Alice grants Bob the right to insert entries into the table `t` that she owns.

Similarly, every procedure is associated with a definer, who is its creating user. That user may then give other users the permission to execute that procedure. In Figure 4.3, for example, Bob grants Carl the right to execute Bob's procedure `def`, and Carl grants Dan the right to execute Carl's procedure `inv`.

A procedure may be annotated to run with invoker's rights or definer's rights. As one may expect, when a procedure is invoked, an invoker's rights procedure runs with the privileges of the invoker of the procedure, and a definer's rights procedure runs with the privileges of the definer. For example, in Figure 4.3, Bob's procedure `def` is specified to be definer's rights. (It is annotated as neither definer's nor invoker's rights; the former is the default.) So, if Carl were to invoke Bob.`def`, the "`insert into alice.t...`" statement within it should run with the privileges of Bob.

The $\mathcal{D}$ authorization system We refer to the subsystem within the database system that comprises discretionary permissions to tables, and permissions to procedures as the $\mathcal{D}$ System. In particular, we focus on the co-existence of the two. We focus on nested procedures — procedures that invoke other procedures, that ultimately issue a SQL statement. Figure 4.3 is an example — `inv` invokes `def`, which in turn issues a SQL statement.

Specifically, we constrain the $\mathcal{D}$ System as follows. We have m procedures $p_1, \ldots, p_m$. Each p_i is owned by a user u_i where $u_i \neq u_j$ for any $i \neq j$. For $i > 1$, the procedure p_i invokes the procedure p_{i-1} only. Only one table exists in the $\mathcal{D}$ System, which we denote as p_0. We consider only one kind of privilege to the table p_0 in the $\mathcal{D}$ System, the privilege to insert into it. The procedure p_1 attempts to exercise the insert privilege to p_0. The table p_0 is owned by a user u_0.

```
Alice: grant insert on table t to Bob

Bob:   procedure def(x), definer's rights
         insert x into Alice's table t

       grant execute on procedure def to Carl

Carl:  procedure inv(x), invoker's rights
         invoke Bob's procedure def(x)

       grant execute on procedure inv to Dan

Dan:   invoke Carl's procedure inv(23)
```

Figure 4.3: An example with four users in the $\mathcal{D}$ System, Alice, Bob, Carl and Dan. Alice owns the table `t`, and grants insert privilege over it to Bob. Bob defines the procedure `def` to be definer's rights and grants Carl execute privilege to it. The procedure `def` inserts a row into Alice's table. Carl creates an invoker's rights procedure `inv` and grants Dan execute privilege to it. The procedure `inv` invokes Bob's procedure `def`. Finally, Dan attempts to execute Carl's procedure `inv`.

We also have a user u_{m+1} that owns no procedure. The properties we care about address whether the user u_{m+1} can exercise the insert privilege over the table p_0 via the procedures $p_m, \ldots, p_1$.

An example of the $\mathcal{D}$ System is the system shown in Figure 4.3. In the figure, u_0 is Alice, u_1 is Bob, u_2 is Carl and u_3 is Dan. The table `t` is p_0, the procedure `def` is p_1, and `inv` is p_2. We seek to know whether Dan can exercise the insert privilege over the table Alice.`t` by executing Carl.`inv`.

The $\mathcal{D}$ System in our syntax We adopt the predicates auth, dAuth and dUnauth as we discuss in Section 4.2. Each is 2-ary. We discuss their semantics in the $\mathcal{D}$ System below.

We adopt a predicate $\mathsf{def}(p)$ to indicate whether a procedure p is definer's rights. We adopt also a function $\mathsf{eff}(p)$ that outputs the effective user of p and is defined inductively using the def predicate as follows:

$$\begin{aligned}
&\mathsf{def}(p_i) \rightarrow \mathsf{eff}(p_i) = u_i \quad \text{for all } i \\
&\neg\mathsf{def}(p_i) \rightarrow \mathsf{eff}(p_i) = \mathsf{eff}(p_{i+1}) \quad \text{for all } i < m \\
&\neg\mathsf{def}(p_m) \rightarrow \mathsf{eff}(p_m) = u_{m+1}
\end{aligned}$$

Unix Our above rules for defining the eff function in the $\mathcal{D}$ System come from Unix systems. We observe the similarity between executing procedures in the $\mathcal{D}$ System, and executing programs in Unix systems.

In Unix systems, a program typically runs with invoker's rights. However, it is possible to

use the setuid bit [59] to cause a program to run with definer's rights. Given this similarity and the somewhat long history of the setuid bit, we adopt the way Unix systems handle indirect authorization via running programs as correct.

Our definition of the eff function above captures the manner in which Unix systems define what they call the effective user in a program that is running. If a program does not have the setuid bit set, then the effective user is the invoker; otherwise, it is the definer. (Of course, we do not consider programs that change the effective user within themselves, as those scenarios are not pertinent to the $\mathcal{D}$ System.)

Semantics of auth **and** dAuth The predicate $\mathsf{dAuth}(x, y)$ has to do with whether x is directly authorized to y. In the $\mathcal{D}$ System, $\mathsf{dAuth}(x, y)$ becomes true if we issue the SQL command to (directly) authorize x to y. For example, in Figure 4.3, after Alice issues the grant command, for $u_1 =$ Bob and p_0 the table, $\mathsf{dAuth}(u_1, p_0)$ is true. In the start-state, $\neg\mathsf{dAuth}(x, y)$ for all x, y.

As we mention above, our properties of interest deal with whether u_{m+1} can acquire the privilege to p_0 via his authorization to p_m. Consequently, the semantics of $\mathsf{auth}(u_i, p_j)$ is whether u_i has access to p_j via his right to execute p_{i-1}, in the case that $i > 1$. Of course, if $j \geq i$, then $\neg\mathsf{auth}(u_i, p_j)$ in all states. If $i = 1$, then $\mathsf{auth}(u_1, p_0) = \mathsf{dAuth}(u_1, p_0)$.

In the testing system that we discuss in Section 4.2.2 and show in Figure 4.2, these semantics translate to the following for the trace-instance-checker. The truth-value of $\mathsf{dAuth}(x, y)$ and $\mathsf{dUnauth}(x, y)$ is assumed based on the commands we issue. We ascertain the truth value of $\mathsf{auth}(u_i, p_j)$ in a state by checking whether u_i is able to successfully execute p_j via p_{i-1}.

4.3.1 Properties of interest

We now articulate the three properties we consider for the $\mathcal{D}$ System. We call them necessity, security and temporal consistency.

Necessity With the necessity property, we intuitively seek to answer the following question: is there a situation that a user is required to have more privileges than should be necessary to gain access? If yes, then we say that the system does not possess the necessity property; otherwise we say it does. The opposite of necessity is redundancy — that the system requires more privileges than should be necessary.

What we mean by "more" privileges in this context comes from the scenario that corresponds to the $\mathcal{D}$ System in Unix, as we discuss in the previous paragraph.

Definition 23 (Necessity) *The $\mathcal{D}$ System has the Necessity property if and only if the following is true for all u_i, p_j, $0 \leq j < i$ in all states of the system.*

$$(\mathsf{dAuth}(u_i, p_{i-1}) \wedge_{k=j+1}^{i-1} \mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})) \rightarrow \mathsf{auth}(u_i, p_j)$$

To explain the above property, it may be easier to consider its contrapositive. What it says is if u_i is not authorized to p_j, then u_i is not directly authorized to p_{i-1}, or for some k, the effective user of p_k is not authorized to p_{k-1}. As we mention above, this comes directly from the notion of effective user in the context of Unix programs. For u_i to be authorized to p_j via the "chain" of programs $p_{i-1}, \ldots, p_{j+1}$, the effective user for each of those programs must be authorized to the program at the next level. We recall from the previous section that the only thing p_k does is attempt to exercise the privilege over p_{k-1}.

Security The security property asks whether there is a situation in which a user is able to gain access with too few privileges.

Definition 24 (Security) *The $\mathcal{D}$ System has the security property if and only if the following is true for all u_i, p_j, $0 \leq j < i$ in all states of the system.*

$$\mathsf{auth}(u_i, p_j) \rightarrow (\mathsf{dAuth}(u_i, p_{i-1}) \wedge_{k=j+1}^{i-1} \mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1}))$$

The security property can be seen as the complement of the necessity property. It states that if u_i is authorized to p_j, then u_i must be authorized to p_{i-1}, and for every p_k, the effective user for p_k must be authorized to p_{k-1}.

Temporal Consistency Temporal consistency captures the following intuition. Suppose, at a given time, a certain set of privileges is necessary and sufficient to gain some access. Then, at any time in future, that set should be exactly the set of privileges needed to gain the same access. We expect the $\mathcal{D}$ System to have temporal consistency — the set of privileges to gain access should not change over time.

Definition 25 (Temporal Consistency) *The $\mathcal{D}$ System has the temporal consistency property if and only if the following is true for all u_i, p_j, p_l, $0 \leq j < l < i$ in all states of the system.*

$$(\mathsf{auth}(u_i, p_j) \wedge \mathsf{dAuth}(\mathsf{eff}(p_l), p_l)) \rightarrow \mathsf{X}(\mathsf{dUnauth}(\mathsf{eff}(p_l), p_l) \rightarrow \mathsf{X}(\mathsf{dAuth}(\mathsf{eff}(p_l), p_l) \rightarrow \mathsf{auth}(u_i, p_j)))$$

4.3.2 Traces

We now discuss the traces that are necessary and sufficient to verify the properties from the previous section. As we mention in Section 4.1, properties are declarative assertions and it is not possible to show that a system has or does not have a property directly. To relate properties and systems, we introduce the notion of a trace. A trace is an imperative (or procedural) specification of states that an authorization system can reach or cannot reach.

Definition 26 (Trace) *A trace is a tuple $\langle s_{init}, t, s_{fin} \rangle$, where s_{init} is an initial authorization state, t is a sequence of state-changes, and s_{fin} is a final authorization state.*

As notation, we write "$\mathsf{auth}(a,b)$ in the state s" to indicate an authorization, and "$\neg\mathsf{auth}(a,b)$ in ..." to indicate a lack of authorization. For the state-changes, we know that each causes a dAuth or dUnauth to become true. Therefore, we represent a sequence of state changes with those predicates. For example, the sequence $\mathsf{dAuth}(a,b), \mathsf{dUnauth}(b,c)$ indicates that a is first authorized to b, and then, b is unauthorized from c.

To express traces of interest to us, we make the following syntactic assumptions about any trace. These assumptions pertain to dAuth and dUnauth. As we mention in Section 4.2, those predicates have to do with whether we issue commands for authorization or unauthorization. Therefore, it is easy to ensure that these assumptions are indeed true — we do not cause the actions that correspond to dAuth and dUnauth to occur in violation of these assumptions.

One of our assumptions is that we cannot have two instances of $\mathsf{dAuth}(a,b)$ in a sequence of state-changes unless there is a $\mathsf{dUnauth}(a,b)$ somewhere between them. Similarly, we cannot have two instances of $\mathsf{dUnauth}(a,b)$ unless we have a $\mathsf{dAuth}(a,b)$ between them. Another assumption is that an instance $\mathsf{dUnauth}(a,b)$ cannot occur in a sequence when the number of $\mathsf{dAuth}(a,b)$ and $\mathsf{dUnauth}(a,b)$ that occur before is even.

Definition 27 (Redundancy trace) *A trace in the $\mathcal{D}$ System is redundant if and only if the entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init}, for every x, y, $\neg\mathsf{auth}(x,y)$ and $\mathsf{dUnauth}(x,y)$.*
- *Let $t = \langle t_1, \ldots, t_n \rangle$ for $n \geq 1$ and $i, j \geq 0$:*
  - $\mathsf{dUnauth}(\mathsf{eff}(p_k), p_{k-1}) \rightarrow \mathsf{F}(\mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1}))$ *for $0 \leq j < k < i \leq n$ in t, and,*
  - $\mathsf{dUnauth}(u_i, p_{i-1}) \rightarrow \mathsf{F}(\mathsf{dAuth}(u_i, p_{i-1}))$ *in t.*
- *In the final state s_{fin}, $\neg\mathsf{auth}(u_i, p_j)$.*

The initial state, s_{init} has no authorizations. For the trace t between s_{init} and the final state s_{fin} above, the first condition states that the effective user of p_k is eventually authorized directly to p_{k-1}. The second condition states that u_i is eventually authorized directly to p_{i-1}. The point behind these conditions are clarified in the next section, in which we relate traces to properties.

Definition 28 (Insecure trace) *A trace in the $\mathcal{D}$ System is insecure if and only if The entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init}, for every x, y, $\neg\mathsf{auth}(x,y)$ and $\mathsf{dUnauth}(x,y)$.*
- *Let $t = \langle t_1, \ldots, t_{n-1}, t_n \rangle$ and $t' = \langle t_1, \ldots, t_{n-1} \rangle$ for $n \geq 1$ and $i, j \geq 0$:*
  - $\mathsf{dUnauth}(\mathsf{eff}(p_k), p_{k-1}) \rightarrow \mathsf{F}(\mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1}))$ *for $0 \leq j < k < i \leq n$ in t'.*
  - $\mathsf{dUnauth}(u_i, p_{i-1}) \rightarrow \mathsf{F}(\mathsf{dAuth}(u_i, p_{i-1}))$ *in t'.*

– *Either* $t_n = \mathsf{dUnauth}(\mathsf{eff}(p_{k'}), p_{k'-1})$ *for some* k', $0 \leq j < k' < i$, *or* $t_n = \mathsf{dUnauth}(u_i, p_j)$.

- *In the final state* s_{fin}, $\mathsf{auth}(u_i, p_i)$.

Definition 29 (Temporally inconsistent trace) *A trace in the* $\mathcal{D}$ *System is temporally inconsistent if and only if the entries* s_{init}, t *and* s_{fin} *are as follows.*

- *In the initial state* s_{init}, *for every* x, y, $\neg\mathsf{auth}(x, y)$ *and* $\mathsf{dUnauth}(x, y)$.
- *Let* $t = \langle t_1, \ldots, t_{n-2}, t_{n-1}, t_n \rangle$ *and* $t' = \langle t_1, \ldots, t_{n-2} \rangle$ *for* $n \geq 1$ *and* $i, j \geq 0$*:*
  – $\mathsf{dUnauth}(\mathsf{eff}(p_k), p_{k-1}) \rightarrow \mathsf{F}(dauth(\mathsf{eff}(p_k), p_{k-1}))$ *for* $0 \leq j < k < i \leq n$ *in* t'.
  – $\mathsf{dUnauth}(u_i, p_{i-1}) \rightarrow \mathsf{F}(dauth(u_i, p_{i-1}))$ *in* t'.
  – $t_{n-1} = \mathsf{dUnauth}(actv(p_l), p_{l-1})$ *for some* l, $0 \leq j < l < i$.
  – $t_n = \mathsf{dAuth}(actv(p_l), p_{l-1})$ *for the same* l *as the previous state change.*
- *In the final state* s_{fin}, $\neg\mathsf{auth}(u_i, p_j)$.

4.3.3 Relating Properties and Traces

We now relate the properties from Section 4.3.1 with the traces from the previous section in Theorems 15–17.

Theorem 15 (Necessity) *The* $\mathcal{D}$ *System has the necessity property if and only if it produces no redundancy trace.*

PROOF.

Only if: By contradiction. Assume that the system has the necessity property, but produces a redundancy trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state s_{fin}, $\neg\mathsf{auth}(u_i, p_j)$ from Definition 27. We immediately have a contradiction because we assume that the system has the necessity property, and therefore, the formula from Definition 23 should be true in all states, but it is false in the state s_{fin}.

If: Assume that the system does not have the necessity property. This means that there exists a state in which the formula from Definition 23 is not true. The only way for it to be not true is with $\neg\mathsf{auth}(u_i, p_j)$ and $(\mathsf{dAuth}(u_i, p_{i-1}) \wedge_{k=j+1}^{i-1} \mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})$ true. So, the system can produce the following redundancy trace: starting from s_{init}, in which there are no authorizations, we start performing state changes of the form $0 \leq j < k < i$, $\mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})$ and $\mathsf{dAuth}(u_i, p_{i-1})$ for a specific i and j. Then we enter the state in which all the predicates related to the exercised state changes are true. From the state in our initial assumption, we know that there exists such a state in the system. So we have a state in which $\mathsf{auth}(u_i, p_j)$ is not true. The produced trace is a redundancy trace. □

Theorem 16 (Security) *The $\mathcal{D}$ System has the security property if and only if it produces no insecure trace.*

PROOF.

Only if: By contradiction. Assume that the system has the security property, but produces an unsecure trace $\langle s_{init}, t, s_{fin}\rangle$. Then we know that, in the final state s_{fin}, $\mathsf{auth}(u_i, p_j)$ is true from Definition 28. We immediately have a contradiction because we assume that the system has the security property, and therefore, the formula from Definition 24 should be true in all states, but it is false in the state s_{fin}.

If: Assume that the system does not have security property. It means that there exists a state in which the formula from Definition 24 is not true. The only way for it to be not true is when the left side formula $\mathsf{auth}(u_i, p_j)$ is true and right side subformula $(\mathsf{dAuth}(u_i, p_{i-1}) \wedge_{k=j+1}^{i-1} \mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})$ is false. So the system can produce the following insecure trace: starting from s_{init}, in which there are no authorizations, we start performing state changes of the form $0 \leq j < k < i$, $\mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})$ and $\mathsf{dAuth}(u_i, p_{i-1})$ for a specific i and j. Then we do $\mathsf{dUnauth}$ (u_i, p_{i-1}) as a state change to reach a state where $\mathsf{auth}(u_i, p_j)$ is true and $(\mathsf{dAuth}(u_i, p_{i-1}) \wedge_{k=j+1}^{i-1} \mathsf{dAuth}(\mathsf{eff}(p_k), p_{k-1})$ is false. $\square$

Theorem 17 (Temporal consistency) *The $\mathcal{D}$ System has the temporal consistency property if and only if it produces no temporally inconsistent trace.*

PROOF.

Only if: By contradiction. Assume that the system has the temporal consistency property, but produces a temporally inconsistent trace $\langle s_{init}, t, s_{fin}\rangle$. Then we know that, in the final state s_{fin}, $\neg\mathsf{auth}(u_i, p_j)$ from Definition 29. We immediately have a contradiction.

If: Assume that the system is not temporally consistent. It means that there exists a state in which the formula from Definition 25 is not true. The only way for it to be false is when $(\mathsf{auth}(u_i, p_j) \wedge \mathsf{dAuth}(\mathsf{eff}(p_l), p_l))$, in the next state $\mathsf{dUnauth}(\mathsf{eff}(p_l), p_l)$ and in the following next state $\neg\mathsf{auth}(u_i, p_j) \wedge \mathsf{dAuth}(\mathsf{eff}(p_l), p_l)$ are true. So there is a trace that can be produced by the system as follows: from the empty state we do state changes to enter the state in which we have $\mathsf{auth}(u_i, p_j) \wedge \mathsf{dAuth}(\mathsf{eff}(p_l), p_l)$. To enter such a state we do the direct authorizations that the necessity property proposes. If the system does not have the necessity property we do more state changes to give the objects inside the sequence of access of user u_i to procedure p_j to grant access of p_j to u_i indirectly via other procedures inside the sequence. Then we do a sequence of state changes $\mathsf{dUnauth}(u_k, p_l)$, and $\mathsf{dAuth}(u_k, p_l)$. In the following state s_{fin}, $\mathsf{auth}(u_i, p_j)$ is false. The produced trace is a temporally inconsistent trace. $\square$

4.3.4 Results for the $\mathcal{D}$ system

We have used our testing-system with different values for m (number of procedures and corresponding users) to exercise the $\mathcal{D}$ System with the traces we discuss in the previous section to determine whether it has the necessity, security and temporal consistency properties. We have discovered that the $\mathcal{D}$ System does not have the necessity, security or temporal consistency properties. To our knowledge, these flaws in the $\mathcal{D}$ System have not been presented before in the literature.

The lack of the necessity property means that some extra privileges than what we would consider meaningful are needed before a user has access. The lack of the security property means that with only seemingly insufficient privileges, a user is able to gain access. And the lack of temporal consistency means that the set of privileges that is needed changes over time.

We now present the results for the smallest m for which the $\mathcal{D}$ System displays these problems. This was for $m = 3$. That is, we have five users, $u_0, \ldots, u_4$. We have a table p_0, and three procedures $p_1, \ldots, p_3$. The user u_i owns the procedure/table p_i.

Correctness is specified in our necessity, security and temporal consistency properties in Section 4.3.1. We have verified that our Linux system has all of these properties. In the Linux system p_0 is a data file that the user u_4 attempts to write to, by invoking the program p_3. The program p_3, in turn, attempts to execute the program p_2 and so on.

We explain one of the cases in detail, and present all the results in Table 4.2. We summarize our findings after our discussions of the case that we present in detail.

Consider that case that $\mathsf{def}(p_1), \neg\mathsf{def}(p_2), \neg\mathsf{def}(p_3)$. That is, the case that p_1 is definer's rights, and p_2 and p_3 are invoker's rights. Correct authorizations are shown in the following table.

Correct				
	p_0	p_1	p_2	p_3
u_0	0	0	0	0
u_1	1	0	0	0
u_2	0	0	0	0
u_3	0	0	0	0
u_4	0	1	1	1

The above table expresses the authorizations for the necessity and security properties to hold. What we require for this case is that u_4 is authorized to p_1, p_2 and p_3, and u_1 is authorized to p_0. To intuit why this makes sense, consider the following.

The procedure p_3 requires u_4 to have execute privilege over it. As it is invoker's rights, when p_3 invokes p_2 within it, the effective user is u_4. The procedure p_2 is also invoker's rights. Therefore, the effective user is still u_4, and u_4 needs have the execute privilege to p_1. Finally, p_1 is definer's rights. Therefore, the effective user is its definer, u_1, who needs to have the insert privilege to p_0.

The $\mathcal{D}$ System's table differs from the above. At the time that the procedures are defined, the following is the table for the $\mathcal{D}$ System. We associate it with the subscript 1 to indicate that this is at the time that the procedures are defined.

$\mathcal{D}_1$				
	p_0	p_1	p_2	p_3
u_0	0	0	0	0
u_1	1	0	0	0
u_2	0	1	0	0
u_3	0	0	1	0
u_4	0	0	0	1

The above table demonstrates that for this case, the $\mathcal{D}$ System does not satisfy the security or necessity properties. For the security property to be satisfied, we require a 1 to appear in the same cells as the Correct table above. The fact that u_4 is not required to have the execute privilege over p_2 is an example of something that causes the security property to be violated.

For the necessity property to be satisfied, we require no 1's to appear where there is a 0 in the corresponding cell in the Correct table. The fact that u_3 is required to have execute privilege to p_2 is an example of something that causes the necessity property to be violated.

Once the procedures have been defined, it turns out that u_4 is authorized to p_0 (via $p_3, \ldots, p_1$) even if some rights are revoked, as indicated by the following table. This demonstrates its lack of the security property as we have only three 1's, as opposed to four in the Correct table above. We associate this table with the subscript 2 to indicate that this is some time-period after the procedures are defined.

$\mathcal{D}_2$				
	p_0	p_1	p_2	p_3
u_0	0	0	0	0
u_1	1	0	0	0
u_2	0	0	0	0
u_3	0	0	1	0
u_4	0	0	0	1

4.3.5 Summary of results

Table 4.2 demonstrates that co-existence of procedure authorizations and discretionary authorizations to tables are flawed in the $\mathcal{D}$ System. In only one case does $\mathcal{D}_1$ (the $\mathcal{D}$ System at the time of defining the procedures) match the Correct table — this is when all three of p_1, p_2 and p_3 are definer's rights. However, even for this case, we have a temporal consistency problem, as we

$\mathsf{def}(p_1), \neg\mathsf{def}(p_2), \neg\mathsf{def}(p_3)$			$\mathsf{def}(p_1), \neg\mathsf{def}(p_2), \mathsf{def}(p_3)$		
Correct	$\mathcal{D}_1$	$\mathcal{D}_2$	Correct	$\mathcal{D}_1$	$\mathcal{D}_2$
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0
0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0
0 0 0 0	0 0 1 0	0 0 1 0	0 1 1 0	0 0 1 0	0 0 1 0
0 1 1 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
$\mathsf{def}(p_1), \mathsf{def}(p_2), \neg\mathsf{def}(p_3)$			$\mathsf{def}(p_1), \mathsf{def}(p_2), \mathsf{def}(p_3)$		
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0	1 0 0 0
0 1 0 0	0 1 0 0	0 0 0 0	0 1 0 0	0 1 0 0	0 0 0 0
0 0 0 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
0 0 1 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
$\neg\mathsf{def}(p_1), \neg\mathsf{def}(p_2), \mathsf{def}(p_3)$			$\neg\mathsf{def}(p_1), \neg\mathsf{def}(p_2), \neg\mathsf{def}(p_3)$		
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0
0 0 0 0	0 1 0 0	0 0 0 0	0 0 0 0	0 1 0 0	0 0 0 0
1 1 1 0	1 0 1 0	1 0 1 0	0 0 0 0	0 0 1 0	0 0 1 0
0 0 0 1	0 0 0 1	0 0 0 1	1 1 1 1	1 0 0 1	1 0 0 1
$\neg\mathsf{def}(p_1), \mathsf{def}(p_2), \neg\mathsf{def}(p_3)$			$\neg\mathsf{def}(p_1), \mathsf{def}(p_2), \mathsf{def}(p_3)$		
0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0	1 0 0 0	0 0 0 0
1 1 0 0	1 1 0 0	1 0 0 0	1 1 0 0	1 1 0 0	1 0 0 0
0 0 0 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0	0 0 1 0
0 0 1 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1

Table 4.2: Our complete results for our tests for the $\mathcal{D}$ System with $m = 3$. For each case (e.g., $\mathsf{def}(p_1), \mathsf{def}(p_2), \neg\mathsf{def}(p_3)$), we have five rows that correspond to the rights of the users $u_0, \ldots, u_4$, respectively. We have four columns, that correspond to the table p_0, and procedures $p_1, \ldots, p_3$. A "1" indicates that that user needs to have the privilege to that procedure/table so the necessity and security properties are satisfied. $\mathcal{D}_1$ corresponds to the system immediately after we create all the procedures, and $\mathcal{D}_2$ corresponds to the system at a later time, after we perform some actions on it.

are able to revoke u_2's authorization to execute p_1 once the procedures have been defined, and still, u_4 is able to exercise the insert privilege over the table p_0 via the procedure p_3.

In one other case, specifically $\neg\mathsf{def}(p_1), \mathsf{def}(p_2), \mathsf{def}(p_3)$, $\mathcal{D}_1$ does not have a security problem. However, it has a necessity problem: the user u_1 is redundantly required to have the insert privilege to p_0. In this case as well, we have a temporal consistency problem. Furthermore, $\mathcal{D}_2$ in this case has a security issue — two of the privileges can be revoked after defining the procedures, and u_4 still has access.

Vendor's response We have communicated our findings to the vendor of the $\mathcal{D}$ System, and helped them reproduce these issues. Their response is the following. They acknowledge all three of the security, redundancy and temporal-inconsistency issues. For the first two issues (security and redundancy), they clarified to us that these exist by design.

That is, the default privilege mode when one invokes a procedure p_2 from within a procedure p_1 is for p_2 to run with definer's rights, immaterial of whether p_1 is defined to run with invoker's or definer's rights. They have a different (and somewhat less natural) syntax by which a programmer can cause p_2 to run with invoker's rights instead. To us, this default choice suggests poor design; however, that is a subjective assessment on our part.

As for temporal-inconsistency, the vendor has acknowledged that this is a bug in their implementation. They are working on what they call a critical patch, and requested that we not publish this dissertation till their customers have had a chance to apply that patch. They like to give their customers up to 1 year to apply the patch. Instead, we offered to use a pseudonym to refer to their system and abstract some syntactic details so we could go ahead and publish now, but still be sensitive to any security implications to their customers. They tell us that we will be credited when their patch is released.

4.4 Application 2: the $\mathcal{I}$ System

Our other application has been to a commercial file-sharing system that we call the $\mathcal{I}$ System. The properties that we discuss for the $\mathcal{I}$ System are somewhat different than for the $\mathcal{D}$ System. The reason is that the $\mathcal{I}$ System's features that we seek to exercise are different from those of the $\mathcal{D}$ System. For example, the $\mathcal{I}$ System does not have the equivalent of procedures.

The $\mathcal{I}$ System In the $\mathcal{I}$ System, a resource is a file. Authorization is discretionary — there is the notion of users, who own files. The owner of a file may grant access to the resource to other users. Such users must first be designated as what are called contacts of the owner. A user may create groups and add contacts to groups.

There is only one kind of access, which is called sharing. A user may authorize either a contact or a group access to a particular file he owns. The software provides two interfaces to a user for authorization management.

If an owner authorizes a user u to a group or file x and the interface says that that is successful, then we assume that u is authorized to x. For indirect authorizations, we can check whether a user in question indeed has an access, or does not have access with a file-access test outside of the $\mathcal{I}$ system.

We are able to model the $\mathcal{I}$ System in our syntax using only the 2-ary predicates auth, dAuth and dUnauth. There is only one kind of authorization in the $\mathcal{I}$ System, which is to share a file.

4.4.1 Properties of interest

We defined six properties for the $\mathcal{I}$ System: necessity, security, forward availability, backward availability, forward safety and backward safety. We define these below. The first two, necessity and security, are somewhat simpler versions compared to those for the $\mathcal{D}$ System.

Definition 30 (Necessity) *$\mathcal{I}$ System has the necessity property if and only if the following is true for all a, b in all states of the system.*

$$\mathsf{dAuth}(a, b) \longrightarrow \mathsf{auth}(a, b)$$

The property defined in Definition 30 expresses that if a is directly authorized to b, then it is authorized to b.

Definition 31 (Security) *$\mathcal{I}$ System has the security property if and only if the following is true for all a, b in all states of the system.*

$$(\mathsf{dUnauth}(a, b) \wedge \neg(\exists l \geq 1\,, x_1, \ldots, x_l.\, \mathsf{dAuth}(a, x_1)$$
$$\wedge\, \mathsf{dAuth}(x_1, x_2) \wedge \ldots \mathsf{dAuth}(x_l, b))) \longrightarrow \neg\mathsf{auth}(a, b)$$

The property defined in Definition 31 expresses that if a is directly unauthorized from b, and there is no "chain" of direct authorizations from a to b, then a is not authorized to b.

With the following forward-availability property, we seek to capture what is customarily seen as a natural semantics for authorization via groups and roles. When a has access to b, and in future b has access to c, then a has access to c.

Definition 32 (Forward Availability) *$\mathcal{I}$ System is forward available if and only if, for all a, b, c.*

$$(\mathsf{auth}(a, b) \wedge \neg\mathsf{auth}(b, c)) \rightarrow \mathsf{X}(\mathsf{auth}(b, c) \rightarrow \mathsf{auth}(a, c))$$

An intuition behind forward availability above is provided by the example of a user u being authorized to a resource r via membership in a group g. We require that u must be a member of g before g is authorized to r for u to be authorized to r. With backward availability, we change the order of the two direct authorizations from that example. That is, for u to be authorized to r, we require that g be authorized to r before u is authorized to g.

Definition 33 (Backward Availability) *$\mathcal{I}$ System has the backward availability property if and only if the following is true for all a, b, c.*

$$(\mathsf{auth}(b,c) \wedge \neg\mathsf{auth}(a,b)) \rightarrow \mathsf{X}(\mathsf{auth}(a,b) \rightarrow \mathsf{auth}(a,c))$$

A system may have one or both of the forward and backward availability properties, or neither. In our example from above of u being authorized to f via g, a system has both properties when the ordering of the two direct authorizations does not matter. This is the case, for example, for groups in Unix systems. In other systems such as secure multicast [60], forward availability is desirable, but not backward availability.

The above availability properties are two example properties where we want authorization to hold. With forward and backward safety, we have two example properties where we want authorization to not hold.

With forward safety below we express that unless a is already authorized to c, authorizing b to c after a is authorized to b does not grant a authorization to c. From the standpoint of intuitive appeal, we see forward safety as the natural counterpart of forward availability. That is, simply because b is now authorized to c should not give a authorization to c via b. To us, this captures the notion of the "leakage" of a privilege that was originally proposed by Harrison et al. [25].

Definition 34 (Forward Safety) *$\mathcal{I}$ System has the forward safety property if and only if the following is true for all a, b, c.*

$$(\mathsf{auth}(a,b) \wedge \neg\mathsf{auth}(b,c) \wedge \neg\mathsf{auth}(a,c)) \rightarrow \mathsf{X}(\mathsf{auth}(b,c) \rightarrow \neg\mathsf{auth}(a,c))$$

The following backward-safety property is the safety counterpart of backward availability. The difference from forward safety is the temporal ordering of the authorizations of a to b, and b to c.

Definition 35 (Backward Safety) *$\mathcal{I}$ System has the backward safety property if and only if the following is true for all a, b, c.*

$$(\mathsf{auth}(b,c) \wedge \neg\mathsf{auth}(a,b) \wedge \neg\mathsf{auth}(a,c)) \rightarrow \mathsf{X}(\mathsf{auth}(a,b) \rightarrow \neg\mathsf{auth}(a,c))$$

Relationship between properties A natural question that arises is whether one of the recent four properties is related to another even before we consider them in the context of a given authorization system. And indeed, we have the following relationships between availability and safety that we fully expect. In the following, a "non-empty system" means a system in which we can create some entity that can take the place of a, b and c in Definitions 32–35.

Theorem 18 *If a non-empty system is forward available, then it is not forward safe. If a non-empty system is backward available, then it is not backward safe.*

It is easy to prove the above theorem. To show the first assertion, for example, we pick an a, b, c and a state such that in that state the left-hand side of the formula in Definition 32 is true and the formula is satisfied. This choice now serves as a counterexample to Definition 34. The contrapositives of the two assertions in the theorem may also be of interest. That is, a forward (backward) safe system is not forward (backward) available. We point out also that these assertions are not "if and only if" – a system may be neither forward (backward) available nor forward (backward) safe.

4.4.2 Traces

We now define traces that correspond to properties in the previous section. Recall the definition for a trace from Section 4.3.2.

Definition 36 (Redundancy trace) *A trace in the $\mathcal{I}$ System is redundant if and only if the entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init}, for every x, y,* $\neg\mathsf{auth}(x, y)$ *and* $\mathsf{dUnauth}(x, y)$.
- *Let $t = \langle t_1, \ldots, t_n \rangle$ for $n \geq 1$:*
  - $\mathsf{dAuth}(a, b) \rightarrow \mathsf{F}(dunauth(a, b))$
  - $t_n = \mathsf{dAuth}(a, b)$.
- *In the final state s_{fin},* $\neg\mathsf{auth}(a, b)$.

Definition 37 (Insecure trace) *A trace in $\mathcal{I}$ System is insecure if and only if the entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init}, for every x, y,* $\neg\mathsf{auth}(x, y)$ *and* $\mathsf{dUnauth}(x, y)$.
- *Let $t = \langle t_1, \ldots, t_n \rangle$ for $n \geq 1$:*
  - *Pick a set $\{x_1, \ldots, x_k\}$ for some $k > 1$.*
  - $\mathsf{dUnauth}(x_i, x_{i+1}) \rightarrow \mathsf{F}(dauth(x_i, x_{i+1}))$ *for $1 \leq i < k$ in t*
  - $t_n = \mathsf{dUnauth}(x_j, x_{j+1})$ *for a j, $1 \leq j < k$.*
- *In the final state s_{fin},* $\mathsf{auth}(x_1, x_k)$.

We use the forward availability property from definition 32 for the following definition of a forward available trace. We denote that except necessity and security properties, other properties only concern the non action predicate. To check these properties using traces we need to assume that the system has necessity property. fortunately $\mathcal{I}$ System has the necessity property.

Definition 38 (Forward available trace) *A trace in $\mathcal{I}$ System is forward available if and only if the entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init},* $\mathsf{auth}(a, b)$, $\mathsf{dAuth}(a, b)$ *and for every x, y except a, b ,* $\neg\mathsf{auth}(x, y)$ *and* $\mathsf{dUnauth}(x, y)$.
- *Let $t = \langle t_1, \ldots, t_n \rangle$ for $n \geq 1$:*
  - $\mathsf{dUnauth}(a, b) \rightarrow \mathsf{F}(\mathsf{dAuth}(a, b))$ *in t.*
  - $\mathsf{dAuth}(a, c) \rightarrow \mathsf{F}(\mathsf{dUnauth}(a, c))$ *in t.*
  - $t_n = \mathsf{dAuth}(b, c)$.
- *In the final state s_{fin},* $\mathsf{auth}(a, c)$.

We define the following three kinds of traces similarly, relying on the corresponding properties from Section 4.4.1.

Definition 39 (Backward available trace) *A trace in $\mathcal{I}$ System is backward available if and only if the entries s_{init}, t and s_{fin} are as follows.*

- *In the initial state s_{init},* $\mathsf{auth}(b, c)$, $\mathsf{dAuth}(b, c)$ *and for every x, y except b, c ,* $\neg\mathsf{auth}(x, y)$ *and* $\mathsf{dUnauth}(x, y)$.
- *Let $t = \langle t_1, \ldots, t_n \rangle$ for $n \geq 1$:*
  - $\mathsf{dUnauth}(b, c) \rightarrow \mathsf{F}(\mathsf{dAuth}(b, c))$ *in t.*
  - $\mathsf{dAuth}(a, c) \rightarrow \mathsf{F}(\mathsf{dUnauth}(a, c))$ *in t.*
  - $t_n = \mathsf{dAuth}(a, b)$.
- *In the final state s_{fin},* $\mathsf{auth}(a, c)$.

Definition 40 (Forward safe trace) *A forward safe trace is the same as a forward available trace (see Definition 38), with the only exception that in the final state s_{fin},* $\neg\mathsf{auth}(a, c)$ *is true.*

Definition 41 (Backward safe trace) *A backward safe trace is the same as a backward available trace (see Definition 39), with the only exception that in the final state s_{fin},* $\neg\mathsf{auth}(a, c)$ *is true.*

4.4.3 Relating Properties and Traces

We now relate properties and traces via Theorems 21–24.

Theorem 19 (Necessity) *The $\mathcal{I}$ System has the necessity property if and only if it produces no redundancy trace.*

PROOF. Only if: By contradiction. Assume that the system has the necessity property, but produces a redundancy trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state s_{fin}, $\neg\mathsf{auth}(a, b)$, from Definition 36. We immediately have a contradiction because we assume that the system has the necessity property, and therefore, the formula from Definition 30 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system does not have necessity property. It means that there exists a state in which the formula from Definition 30 is not true. The only way for it to be not true is with $\mathsf{dAuth}(a, b)$ and $\neg\mathsf{auth}(a, b)$. Now we show that the system can enter such a state starting doing state changes from an empty state. In empty state s_0 we have all $\neg\mathsf{auth}(x, y)$ and $\mathsf{dUnauth}(x, y)$ for all objects x and y. So we do a state change $\mathsf{dAuth}(a, b)$ and enter the state s_1. So the system produces the following trace: $T = \{s_0, \langle \mathsf{dAuth}(a, b) \rangle, s_1\}$, which is a redundancy trace. □

Theorem 20 (Security) *The $\mathcal{I}$ System has the security property if and only if it produces no insecure trace.*

PROOF. Only if: By contradiction. Assume that the system has the security property, but produces an unsecure trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state s_{fin}, $\mathsf{auth}(x_1, x_k)$, from Definition 37. We immediately have a contradiction because we assume that the system has the security property, and therefore, the formula from Definition 31 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system does not have security property. It means that there exists a state in which the formula from Definition 31 is not true. The only way for it to be not true is with $\wedge_{i=1}^{k-1}\mathsf{dAuth}(x_i, x_{i+1})$ and $\mathsf{auth}(x_1, x_k)$. Now we show that the system can enter such a state starting doing state changes from an empty state. We do a sequence of state changes $t_i = \mathsf{dAuth}(x_i, x - i + 1)$ for $1 \leq i < k$ and enter the state s_k. So the system produces the following trace: $T = \{s_0, \langle t_1, \ldots, t_{k-1} \rangle, s_k\}$, which is an unsecure trace. □

Theorem 21 (Forward availability) *The $\mathcal{I}$ System has the forward availability property if and only if it produces no forward safe trace.*

PROOF. Only if: By contradiction. Assume that the system has the forward availability property, but produces a forward safe trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state

s_{fin}, $\neg\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and $\mathsf{auth}(a, b)$ from Definition 40. We immediately have a contradiction because we assume that the system has the forward availability property, and therefore, the formula from Definition 32 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system is not forward available. It means that there exists a state in which the formula from Definition 32 is not true. The only way for it to be not true is when in the next state we have $\neg\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and in the current state we have $(\mathsf{auth}(a, b) \wedge \neg\mathsf{auth}(b, c))$. So there are two consecutive states s_1 and s_2 with the following properties. In s_1 we have only $\mathsf{auth}(a, b)$ (and its related direct authorization predicate) true and in s_2, $\mathsf{auth}(b, c) \wedge \neg\mathsf{auth}(a, c)$ is true. So the system produces the following trace: $T = \{s_1, \langle \mathsf{dAuth}(b, c) \rangle, s_2\}$, which is a forward safe trace. □

Theorem 22 (Backward availability) *The $\mathcal{I}$ System has the backward availability property if and only if it produces no backward safe trace.*

PROOF. Only if: By contradiction. Assume that the system has the backward availability property, but produces a backward safe trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state s_{fin}, $\neg\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and $\mathsf{auth}(a, b)$ from Definition 41. We immediately have a contradiction because we assume that the system has the backward availability property, and therefore, the formula from Definition 33 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system is not backward available. It means that there exists a state in which the formula from Definition 33 is not true. The only way for it to be not true is when in the next state we have $\neg\mathsf{auth}(a, c)$, $\mathsf{auth}(a, b)$ and in the current state we have $(\mathsf{auth}(b, c) \wedge \neg\mathsf{auth}(a, b))$. So there are two consecutive states s_1 and s_2 with the following properties. In s_1 we have only $\mathsf{auth}(b, c)$ (and its related direct authorization predicate) true and in s_2, $\mathsf{auth}(a, b) \wedge \neg\mathsf{auth}(a, c)$ is true. So the system produces the following trace: $T = \{s_1, \langle \mathsf{dAuth}(a, b) \rangle, s_2\}$, which is a backward safe trace. □

Theorem 23 (Forward safety) *The $\mathcal{I}$ System has the forward safety property if and only if it produces no forward available trace.*

PROOF. Only if: By contradiction. Assume that the system has the forward safety property, but produces a forward available trace $\langle s_{init}, t, s_{fin} \rangle$. Then we know that, in the final state s_{fin}, $\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and $\mathsf{auth}(a, b)$ from Definition 38. We immediately have a contradiction because we assume that the system has the forward safety property, and therefore, the formula from Definition 34 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system is not forward safe. It means that there exists a state in which the formula from Definition 34 is not true. The only way for it to be not true is when in the next

state we have $\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and in the current state we have $(\mathsf{auth}(a, b) \wedge \neg\mathsf{auth}(b, c))$. So there are two consecutive states s_1 and s_2 with the following properties. In s_1 we have only $\mathsf{auth}(a, b)$ (and its related direct authorization predicate) true and in s_2, $\mathsf{auth}(b, c) \wedge \mathsf{auth}(a, c)$ is true. So the system produces the following trace: $T = \{s_1, \langle\mathsf{dAuth}(b, c)\rangle, s_2\}$, which is a forward available trace. □

Theorem 24 (Backward safety) *The $\mathcal{I}$ System has the backward safety property if and only if it produces no backward available trace.*

PROOF. Only if: By contradiction. Assume that the system has the backward safety property, but produces a backward available trace $\langle s_{init}, t, s_{fin}\rangle$. Then we know that, in the final state s_{fin}, $\mathsf{auth}(a, c)$, $\mathsf{auth}(b, c)$ and $\mathsf{auth}(a, b)$ from Definition 39. We immediately have a contradiction because we assume that the system has the backward safety property, and therefore, the formula from Definition 35 should be true in all states. But the formula is false in the state s_{fin}.

If: Assume that the system is not backward safe. It means that there exists a state in which the formula from Definition 35 is not true. The only way for it to be not true is when in the next state we have $\mathsf{auth}(a, c)$, $\mathsf{auth}(a, b)$ and in the current state we have $(\mathsf{auth}(b, c) \wedge \neg\mathsf{auth}(a, b))$. So there are two consecutive states s_1 and s_2 with the following properties. In s_1 we have only $\mathsf{auth}(b, c)$ (and its related direct authorization predicate) true and in s_2, $\mathsf{auth}(a, b) \wedge \mathsf{auth}(a, c)$ is true. So the system produces the following trace: $T = \{s_1, \langle\mathsf{dAuth}(a, b)\rangle, s_2\}$, which is a backward available trace. □

4.4.4 Results for the $\mathcal{I}$ System

We established the following results for the $\mathcal{I}$ System using our testing methodology and system. The $\mathcal{I}$ System satisfies the necessity, security, and forward and backward availability properties. It does not possess the forward and backward safety properties.

Vendor's response The vendor of the $\mathcal{I}$ System has confirmed that our results are consistent with their intended design.

4.5 Related Work

We are aware of only three uses of state-space exploration in the context of access control. In safety and security analysis [24, 25, 26], one provides as input a start-state and a specification of how states can change. One then asks whether a state with certain properties is reachable, or whether all reachable states have a certain property. In more recent work [61], state-space exploration has been used to verify that the design of particular authorization schemes is sound. What

this means is that a particular syntax is specified for the specification of authorization schemes. A new authorization scheme may be specified with the syntax, but the question remains as to whether any scheme that is syntactically valid is also semantically valid. State-space exploration may be used for this purpose. The third use is in information flow, for example, the work of D'Souza et al. [62]. Our work is different from such existing work related to state-space exploration in access control. Our goal is to determine whether an implemented access control system has properties that were determined to be desirable in its design. In other words, does the "real world" access control system have a property of interest?

There is also a large amount of research in checking for vulnerabilities in software. A comprehensive discussion is well beyond this dissertation. However, there is relatively little work for real-world access control systems that targets concrete implementations. The work of Bauer et al. [63] is relevant in this context. That work proposes a framework to apply misconfiguration-prediction to access control policies. They predict misconfiguration might happen in the system, based on the visible accesses in the system so far. Our work is on property-testing an implementation of an authorization system rather than the detection of misconfigurations in policies.

Our work borrows ideas from software testing, in particular the small-scopes hypothesis [46]. It also has similarities to recent work that proposes the use of model-checking for software testing [48]. However, such work focuses on aspects such as white-box testing of software (i.e., with access to the source-code) and achieving coverage. Our focus is in testing for properties related to authorization, which to our knowledge, conventional software testing does not cover.

There has also been recent work in the use of traces to verify access control models that are specified in UML [49, 50]. However, such work does not consider testing actual systems, and considers only "on paper" verification. Furthermore, the models tested do not consider administrative changes, which, for example, the $\mathcal{I}$ System that is one of our application contexts incorporates.

Finally, our work also has similarities to flaw-hypothesis or penetration testing [51]. A major difference, however, is that we hypothesize somewhat sophisticated properties that are specific to authorization systems.

4.6 Conclusion

We have addressed the problem of testing an implementation of an authorization system for properties of interest. We have proposed a 4-stage approach: articulation of properties, construction of traces, finitization and a model-checking based approach to testing a system. We have considered several properties in the context of two commercial systems: the $\mathcal{D}$ System and the $\mathcal{I}$ System. We have discovered flaws in the $\mathcal{D}$ System related to the co-existence of authorizations to procedures and discretionary rights to tables that, to our knowledge, have not appeared before in the literature.

There is considerable scope for future work. One avenue is the articulation of a larger set of properties of interest, and abstracting those properties into a "meta-property." This would be

similar to the notion of a query in prior work [24]. Completing the implementation of our testing system is also interesting future work.

Chapter 5

Conclusion

We have addressed three important problems in access control systems design and analysis.

Version Control System Design We have presented aspects of the design and semantics of `gitolite`, an authorization scheme for Version Control Systems. We have used term rewriting rules to present the semantics of `gitolite`. Also we have presented the algorithm which provides access enforcement relied on Access Control Lists (ACLs). The proposed algorithm is sound and complete and we have shown that in real world experiment it is faster than previous enforcement of `gitolite` in many situations.

Chinese Wall Security Policy We have addressed enforcement of the Chinese Wall security policy. Specifically, we have proposed a notion of least-restrictive enforcement of the policy, devised an enforcement mechanism and shown that it is least-restrictive. Our enforcement mechanism is simple, sound and efficient. Our enforcement mechanism mediates read attempts only to prevent subject-violations, and write attempts only to prevent object-violations.

We have precisely identified the trade-off our enforcement mechanism incurs in achieving least-restriction. The actions of a subject may constraint the prospective actions of other subjects, a property that we call (lack of) independence of subjects. We have established the somewhat strong result that *any* enforcement mechanism to enforcing the Chinese Wall security policy that is least-restrictive must incur this trade-off.

We have then established two new results for the Brewer-Nash enforcement mechanism to enforcement [17]. We have shown that it is more restrictive than previous work points out. We have shown that if a subject is allowed to write to an object, then all objects must belong to the same Conflict of Interest class. We have shown also that the so-called $*$-rule that is part of the Brewer-Nash enforcement mechanism to enforcement is overstated in that one of its sub-rules implies the other. We have also investigated the rendition of the Brewer-Nash enforcement mechanism in Bishop's work [32], and shown that it is not equivalent to the Brewer-Nash enforcement mechanism.

In summary, our work establishes new results, and thereby sheds new light on what is generally considered to be important work in information security.

In our discussion on related work, we observe that subsequent enforcement mechanisms of Kessler [20] and Sandhu [21, 22] are also restrictive in their own ways, and not least-restrictive.

We see applications of our enforcement mechanism as the most promising future work. Specifically, we seek to investigate the problems in cloud-computing that prior work such as those of Tsai et al. [15] and Wu et al. [16] have proposed, and ask whether the least-restriction that our enforcement mechanism provides is useful and meaningful in those contexts.

Another direction of future work is to investigate extensions that have been proposed to the policy in work such as those of Atluri et al. [14] and ask how we can extend our enforcement mechanism to deal with such extensions. We seek also to rigorously establish that other enforcement mechanisms such as those of Kessler [20] and Sandhu [21, 22] are indeed incomparable to the Brewer-Nash enforcement mechanism from the standpoint of reachable authorization states.

Property Testing We have addressed the problem of testing an implementation of an authorization system for properties of interest. We have proposed a 4-stage approach: articulation of properties, construction of traces, finitization and a model-checking based approach to testing a system. We have considered several properties in the context of two commercial systems: the $\mathcal{D}$ System and the $\mathcal{I}$ System. We have discovered flaws in the $\mathcal{D}$ System related to the coexistence of authorizations to procedures and discretionary rights to tables that, to our knowledge, have not appeared before in the literature.

There is considerable scope for future work. One avenue is the articulation of a larger set of properties of interest, and abstracting those properties into a "meta-property." This would be similar to the notion of a query in prior work [24]. Completing the implementation of our testing system is also interesting future work.

References

[1] Ross Anderson, “Security Engineering”. WILEY 2001.

[2] R. Chandramouli, “Specification and Validation of Enterprise Access Control Data for Conformance to Model and Policy Constraints”. *7th World Multi-conference on Systemics, Cybernetics and Informatics*, 2003.

[3] S. Chamarty. Gitolite. https://github.com/sitaramc/gitolite/ (Accessed Mar. 2011).

[4] Sitaram Chamarty, Hiren Patel and Mahesh Tripunitara, “An Authorization Scheme for Version Control Systems.” In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2011.

[5] Git – the fast version control system. http://git-scm.com/ (Accessed Dec. 2010).

[6] Apache subversion. http://subversion.apache.org/ (Accessed Dec. 2010).

[7] G. S. Graham and P. J. Denning. Protection — principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429. May 16–18 1972.

[8] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, Apr. 2003.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2), pages 38–47, February 1996.

[10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, May 1996.

[11] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[12] M. Tripunitara and N. Li. A theory for comparing the expressive power of access control models. *Journal of Computer Security*, 15(2), pages 231–272, 2007.

[13] Michael Davis and Andrew Stark. *Conflict of Interest in the Professions*. Oxford University Press, 2001.

[14] Vijayalakshmi Atluri and Soon Ae Chun and Pietro Mazzoleni. A Chinese wall security model for decentralized workflow systems. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 48–57, 2001.

[15] Tien-Hao Tsai and Yen-Chung Chen and Hsiu-Chuan Huang and Pei-Ming Huang and Kuo-Sen Chou and Kuo-Sen Chou. A practical Chinese wall security model in cloud computing. *Network Operations and Management Symposium (APNOMS)*, pages 1–4, 2011.

[16] Ruoyu Wu and Gail-Joon Ahn and Hongxin Hu and M. Singhal. Information flow control in cloud computing. In *Proceedings of the 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 1–7, 2010.

[17] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[18] The Chinese Wall Security Policy – Brewer and Nash, Citation Count. http://scholar.google.ca/scholar?hl=en&q=the+chinese+wall+security+policy&btnG=&as_sdt=1%2C5&as_sdtp= (Accessed Sep. 2012).

[19] Alireza Sharifi and Mahesh V. Tripunitara. Least-restrictive enforcement of the Chinese wall security policy. In *Proceedings of the ACM symposium on Access control models and technologies (SACMAT)*, pages 61–72, 2013.

[20] Volker Kessler. On the Chinese Wall Model. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS '92, pages 41–54, New York, NY, USA, 1992.

[21] Ravi S. Sandhu. Lattice-based enforcement of Chinese Walls. *Computers & Security*, 11(8):753–763, 1992.

[22] Ravi S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proceedings of the 15th National Computer Security Conference*, pages 221–235, 1992.

[23] Alireza Sharifi, Paul Bottinelli, and Mahesh V. Tripunitara. Property-testing real-world authorization systems. In *Proceedings of the ACM symposium on Access control models and technologies (SACMAT)*, pages 225–236, 2013.

[24] N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and Systems Security (TISSEC)*, 9(4):391–420, Nov. 2006.

[25] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, Aug. 1976.

[26] R. S. Sandhu. Undecidability of the safety problem for the schematic protection model with cyclic creates. *Journal of Computer and System Sciences*, 44(1):141–159, Feb. 1992.

[27] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736, 1964.

[28] R. S. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based aministration of roles. *ACM Transactions on Information and Systems Security*, 2(1):105–135, Feb. 1999.

[29] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos and Ronald L. Rivest. Certificate Chain Discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4), pages 285–322, 2001.

[30] Takahito Aoto and Junichi Yoshida and Yoshihito Toyama. Proving Confluence of Term Rewriting Systems Automatically. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, pages 93–102, 2009.

[31] Wouter Gelade and Frank Neven, Succinctness of the Complement and Intersection of Regular Expressions. In *In CoRR abs/0802.2869*, 2008.

[32] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2003.

[33] Dieter Gollmann. *Computer Security*. Wiley 3rd ed., 2011.

[34] T.Y. Lin. Chinese wall security policy-an aggressive model. In *Proceedings of the Fifth Annual Computer Security Applications Conference*, pages 282–289, 1989.

[35] Jay Ligatti and Lujo Bauer and David Walker. Run-Time Enforcement of Nonsafety Policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, Jan. 2009.

[36] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[37] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, Jun. 1962.

[38] Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, Jan. 1962.

[39] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[40] J. A. Goguen and J. Meseguer. Unwinding and Inference Control. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[41] Peter Loscocco and Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, pages 29–42, 2001.

[42] Sushil Jajodia and Pierangela Samarati and V. S. Subrahmanian and Eliza Bertino. A unified framework for enforcing multiple access control policies. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 474–485, 1997.

[43] Guy Edjlali and Anurag Acharya and Vipin Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM conference on Computer and communications security*, pages 38–48, 1998.

[44] Ann E. Kelley Sobel and Jim Alves-Foss. A trace-based model of the chinese wall security policy. In *Proceedings of the 22nd National Information Systems Security Conference*, 1999.

[45] P.W.L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, 2004.

[46] Jackson, D. and Damon, C.A., "Elements of style: analyzing a software design feature with a counterexample detector". *Software Engineering, IEEE Transactions on*, 22(7):484–495, 1997.

[47] Mihhail Aizatulin and Andrew D. Gordon and Jan Jürjens, "Extracting and verifying cryptographic models from C protocol code by symbolic execution". In *Proceedings of the 18th ACM conference on Computer and communications security,(CCS 2011)*, 2011.

[48] Corina S. Pasareanu. Combining Model Checking and Symbolic Execution for Software Testing. In *Tests and Proofs (TAP), Lecture Notes in Computer Science Volume 7305*, pages 2, 2012.

[49] Wuliang Sun and Robert B. France and Indrakshi Ray. Rigorous Analysis of UML Access Control Policy Models. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 9–16, 2011.

[50] Lijun Yu and Robert B. France and Indrakshi Ray and Wuliang Sun. Systematic Scenario-Based Analysis of UML Design Class Models. In *Proceedings of the IEEE 17th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 86–95, 2012.

[51] Richard R. Linde. Operating system penetration. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, AFIPS '75, pages 361–368, New York, NY, USA, 1975.

[52] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, UK, 2nd edition, 2004.

[53] Edmund M. Clarke, Orna Grumberg and Doron Peled. *Model checking*. MIT Press, 2001.

[54] Hao Chen, Drew Dean, David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, 2004.

[55] Chris Anley. Advanced SQL injection in SQL server applications. *White paper, Next Generation Security Software Ltd*, 2002.

[56] E. M. Fayo. Advanced SQL Injection in Oracle Databases. *Technical report, Argeniss Information Security, Black Hat Briefings*, 2005.

[57] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[58] NuSMV. http://nusmv.fbk.eu/ (Accessed Jan. 2013).

[59] Hao Chen and David Wagner and Drew Dean. Setuid Demystified. In *Proceedings of USENIX Security Symposium*, pages 171–190, 2002.

[60] Yongdae Kim, Adrian Perrig and Gene Tsudik. Simple and fault-tolerant key agreement for dynamic collaborative groups. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS 2000)*, pages 235–244, 2000.

[61] Ram Krishnan, Jianwei Niu, Ravi Sandhu and William H. Winsborough. Group-Centric Secure Information Sharing Models for Isolated Groups. *ACM Transactions on Information and System Security*, 14(3):1–29, 2011.

[62] Deepak D'Souza, Raveendra Holla, Raghavendra K. R. and Barbara Sprick. Model-Checking Trace-based Information Flow Properties. *Journal of Computer Security*, 19(1):101–138, 2011.

[63] Lujo Bauer, Yuan Liang, Michael K. Reiter and Chad Spensky. Discovering access-control misconfigurations: New approaches and evaluation methodologies. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, pages 95–104, 2012.