An Authorization Scheme for Version Control Systems

Sitaram Chamarty
Tata Consultancy Services
Hyderabad, India
sitaram@atc.tcs.com

Hiren D. Patel
ECE Department
Univ. of Waterloo, Canada
hdpatel@uwaterloo.ca

Mahesh V. Tripunitara
ECE Department
Univ. of Waterloo, Canada
tripunit@uwaterloo.ca

ABSTRACT

We present `gitolite`, an authorization scheme for Version Control Systems (VCSes). We have implemented it for the Git VCS. A VCS enables versioning, distributed collaboration and several other features, and is an important context for authorization and access control. Our main consideration behind the design of `gitolite` is the balance between expressive power, correctness and usability in realistic settings. We discuss our design of `gitolite`, and in particular the four user-classes in its delegation model, and the administrative actions a user at each class performs. We discuss also our ongoing work on expressing `gitolite` precisely in first-order logic, to thereby give it a precise semantics and establish correctness properties. `gitolite` has been adopted in open-source software development, university and industry settings. We discuss our experience with these deployments, and present some performance results related to access enforcement from a real deployment.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*

General Terms

Security, Design

Keywords

Authorization, Version Control Systems, Git

1. INTRODUCTION

We present `gitolite` [3], a new authorization scheme for Version Control Systems (VCSes). Authorization deals with the specification and management of the rights users have to resources. (The terms "authorization" and "access control" mean the same in this paper.) It is an important aspect of the security of a system. A VCS is used when resources (e.g., software programs) are in flux, and the history of changes need versioning. Typically, a VCS needs to allow multiple users to access, modify and update the resources.

`gitolite` has been implemented for the Git VCS [2] and is sufficiently general to be applicable to other VCSes such as SVN [1].

Our particular focus in this paper is our design of controlled delegation that is part of the scheme. Traditional authorization schemes for VCSes have a single administrator for an instance of a VCS that specifies authorization rules for accessors. Our scheme is more flexible and scalable as it has a finer-grained delegation model. We have validated the scheme with real-world deployments in large-scale open-source software development projects, universities and industry (see Section 5).

In the context of designing an authorization scheme for VCSes, we have set a broader goal for ourselves: to design a scheme that simultaneously addresses expressive power, usability and correctness. The scope of this paper is more modest that our overall goal — we seek to introduce `gitolite`, convey its salient features, and argue that it is indeed different in its expressive power from prior schemes. In this context, our intent is only for our scheme to be applicable to VCSes; 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) [12], Role-Based Access Control (RBAC) [9, 25] and Trust Management [7, 18], are expressive enough to subsume our scheme. We discuss this issue in more detail in Section 6. 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 [31]. 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.

Novelty A number of authorization schemes have been proposed in the research literature. Also, there are several schemes that are deployed in various application domains such as file and database systems. The elements of our scheme are not new. However, the manner in which we put the elements together into a composite scheme is novel. For example, delegation is a well-studied notion in authorization. However, in the context of a new scheme such as ours, one still needs to make a number of choices regarding, for example, the depth of delegation that is allowed, and the sorts of power an individual at a particular depth has.

There are two mindsets that we see in past work on authorization schemes. One is work that is agnostic to the application domain. A lot of past work on RBAC, for example, appears to be based on this mindset. The other mindset is that the application domain influences the authorization scheme significantly. Our design is based on this latter mindset, and our design is validated by our experience from several deployments, as we discuss in Section 5.

Layout The remainder of this paper is organized as follows. In the next section, we introduce VCSes. In Section 2 we discuss how we perform access enforcement. In Section 3, we discuss our authorization scheme. In Section 4, we discuss our ongoing work on giving our scheme a precise encoding in first-order logic. In Section 5, we discuss our experience from deployments. We discuss related work in Section 6, and conclude in Section 7.

1.1 Version Control Systems

A VCS provides the ability to version resources such as data files. (We use the phrase "a VCS" to mean "an instance of a VCS.") Considerable functionality is associated with modern VCSes; some examples are distributed development, non-linear development and the maintenance of history (versions) [1, 2]. A VCS comprises objects such as repositories and branches that help realize such functionality. An authorization scheme for a VCS 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 1: A Version Control System in BNF. The notation ::= stands for "comprises," and $\{\cdot\}$ means 0 or more occurrences.

In Figure 1, we show the components of a VCS from the standpoint of authorization in Backus-Naur Form (BNF) [16], and give an example below. A VCS 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. AUTHENTICATION AND ACCESS ENFORCEMENT

Before we discuss our authorization scheme, we describe authentication and access enforcement with `gitolite`. Our 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." Furthermore, given our implementation as middleware, we expect `gitolite` to be compatible with future versions of Git.

In Figure 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 VCS client, which issues access requests to objects maintained by the VCS server. The communication is mediated by the SSH server, and access requests are mediated by the `gitolite` Reference Monitor. A VCS client communicates a command that is intended for `gitolite` and Git, which is opaque to the SSH server. The SSH server authenticates the client based on its public key (pubKey in Figure 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 3.)

A command from a VCS client comprises two components: the name of a repository (repoName), and what we call a VCS object (vcsObject). The vcsObject is opaque to the SSH server and the `gitolite` Reference Monitor, and must be "unwrapped" by the VCS server. From the standpoint of access enforcement, the vcsObject contains a branchName, the data within the branchName to which the access request pertains, and the manner in which 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 (see Section 3.3). As Figure 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.

As we discuss in Section 3.3, 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 3.4, 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 VCS server, if necessary. It is not necessary if the client wishes read access only. The manner in which 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 VCS 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 which the request pertains. The `gitolite` reference monitor consults its policy and issues a binary decision on the access request.

As we discuss in Section 3, the access policy comprises rules which are compiled into an Access Control List (ACL). In Section 5.4, we present performance results with our approach to access enforcement that we discuss above. Our results indicate that our approach is pragmatic. This has also been validated by our experience with deployments (see Section 5).

3. AUTHORIZATION SCHEME

Our authorization scheme is discretionary [12]. We discuss in Section 3.2 how users specify rules. These rules are compiled into an ACL that is used for access enforcement; we discuss this in Section 3.4. An ACL, in our context, is a set of triples, each of the form $\langle$Resource, Right, User$\rangle$. It indicates that User possesses the Right to the Resource. There are two kinds of negative (deny) rights; we discuss these in Section 3.3.

The authorization rules include administrative actions such as delegations. We being, in the following section, with a discussion

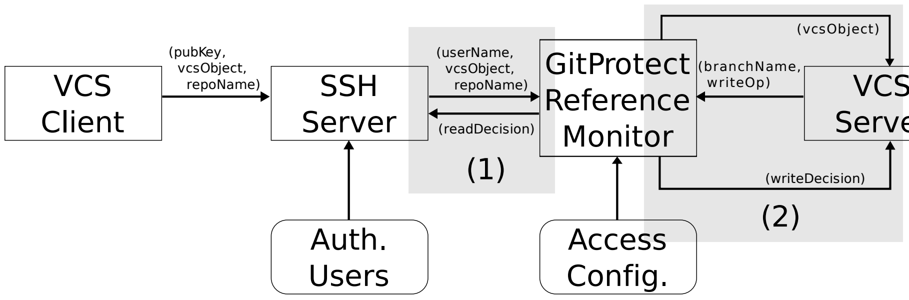


Figure 2: Access enforcement to VCS objects using the `gitolite` reference monitor.

of our classification of users into four user-classes, three of which comprise users that are allowed to specify authorization rules that pertain to administration.

3.1 Classes of Users

Traditional VCSes have only two classes of users: an administrator, and an accessor. As we discuss in Section 6, this is often the case in other application domains such as file systems as well. 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 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 [24] which has a delegation depth of two, and dissimilar to schemes such as RT [18] which allow 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 3 that a VCS Admin may delegate to a Repo Admin. However, he may delegate also to Repo Owner and Accessor.

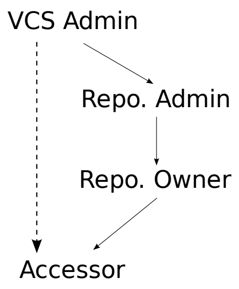


Figure 3: We have four classes of users. The dotted arrow shows what is in traditional VCSes – only two classes of users, VCS Admin and Accessor. We, in addition to those, have Repo Admin and Repo Owner. The transitive closure of the arrows represents "may delegate to."

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 VCS. 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 which may exist smaller projects. The term "within" has no meaning in the context of repositories in a VCS 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 paper 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 VCS objects such as repositories and branches.

A typical chronological flow of actions is as follows. A VCS is installed by the VCS Admin. Initially, and over time the VCS Admin adds users that may access parts of the VCS. The VCS Admin also broadly divides the VCS 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 our access rights to other users. We discuss more concrete uses of `gitolite` in Section 5.

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. Our user-classes are organized in a hierarchy (see Figure 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 VCS 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 VCS.

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

- The set of valid usernames for the VCS. This set is the domain to which the SSH Server maps public keys at the time of access (see Section 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. We discuss these differences in Section 6.

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 which a Repo Admin has purview. He does this by specifying regular expressions for the names of the repositories. (Repositories within a VCS 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 priority 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, which we discuss in Section 3.4.

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 which 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 which 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, which 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 which the user may acquire rights. This is a design choice that balances flexibility (expressive power) and security.

3.3 Objects and Rights

In this section, we specify the access rights that accessors may have to VCS objects. The different kinds of VCS objects and the relationship between them from the standpoint of authorization are shown in Figure 1. The objects are: repositories, branches, tags, folders and files. Rights are classified into read and write. There is only one read right. We designate `read` as the right that gives read rights to a repository. The kinds of write rights are the following.

- `write` – this is the right to write or commit an object. This right may be specified for any VCS object.
- `rewindBranch` – this is the right to be able to revert a branch to an original version. It may be specified for a repository or branch only. If it is specified for a repository, then it applies for every branch in the repository.
- `createBranch` – this is the right to be able to create a branch within a repository. It may be specified for a repository only.
- `deleteBranch` – this is the right to be able to delete a branch within a repository. It may be specified for a repository only.
- `createRepo` – this is the right to be able to create a repository.
- `deleteRepo` – this is the right to be able to delete a repository.

There are also two "negative" rights that explicitly deny the corresponding right to an accessor. These may be used, for example, by a Repo Admin to disallow a Repo Owner from giving the right to an accessor. In Section 3.4, we discuss how these negative rights are reconciled with their "positive" counterparts.

- `denyWrite` – this is the "negative" of `write`.
- `denyRewindBranch` – the "negative" of `rewindBranch`.

3.4 Compilation

As we mention in Section 3.2, in our system, rules are specified in a configuration. There is one configuration per VCS; the configuration is split across one or more files. The VCS Admin has a particular configuration file in which he specifies his rules, such as the legitimate usernames (the concrete values that correspond to the predicate definesUser) and the definitions of mnemonics (definesMnemonic). Similarly, each Repo Admin has his own configuration file.

In this section, we discuss how the configurations that are specified as we discuss in Sections 3.2 and 3.3 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, which comprises $\langle$Resource, Right, User$\rangle$ 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. We relate our discussions in this section to our discussion on access enforcement in Section 2. Also, in Section 5.4, we present data on the performance of access enforcement from a real deployment.

We use Git to administer Git. That is, our configuration files are maintained as part of the Git repository which is the VCS instance. As we mention in Section 3.2, an aspect of our scheme is our 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, which 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.

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 which 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 which each such assignment is associated.
9. For each right to which 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 $\langle$Resource, Right, User$\rangle$ for each Right and Resource to which 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 – see Section 5.4 for some data related to the performance of access enforcement.

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` 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` to that resource.
c. If a user has `createBranch` to a branch, then he has `rewindBranch` to that branch.
d. If a user has `deleteBranch` to a branch, then he has `createBranch` to that branch.

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` right to be at a finer granularity than to an entire repository.

4. TOWARDS A PRECISE ENCODING

It is our goal to establish a precise semantics and correctness properties for our scheme. A comprehensive description of those aspects is beyond the scope of this paper. Indeed, these aspects are ongoing work. In this section, we provide some discussion of how we go about achieving this goal.

We conjecture that our scheme can be expressed precisely in first-order logic [14]. To express a configuration, we specify predicates and functions. We then specify the manner in which we compute the effective rights (see Section 3.4) as inference rules in first-order logic. In the following section, we discuss the predicates and functions that we adopt to express a configuration. We discuss also an subset of the inference rules that we adopt to infer effective rights. We introduce "intermediate" predicates for ease of exposition.

What we seek is to define predicates that map directly to elements in a configuration. Then, we are able to make such a predicate concrete in a model [14] by populating it with values directly from the configuration. In Section 4.2 we discuss how we intend to specify a precise semantics along these lines.

4.1 Authorizations

In this section, we discuss the predicates and functions that we adopt to express a configuration. We discuss also a subset of the inference rules that are used to infer the effective rights. We follow the same flow as in Section 3.2.

VCS Admin We associate the following predicates with actions that a VCS Admin performs.

- definesUser(u, v) indicates that u is defined to be a valid user by the user v.
- definesMnemonic(m, u) indicates that m is a mnemonic that has been defined by user u.
- definesRepoAdmin(u, R, v) indicates that u is designated a Repo Admin to the repository R by the user v.

As we mention in Section 3.2, more than one Repo Admin may be associated with a set of repositories. A VCS Admin associates a priority with each Repo Admin for the authorization rules. We specify the following functions.

- $vcsAdmin$ – this is the constant function that identifies the VCS Admin.
- The function $adminPrio(u, r, v)$ is the strict priority that the user u has to administer the repository r according to v.

Repo Admin We associate the following predicates with the actions that pertain to a Repo Admin.

- mnemonicHasRight(m, ρ, o, u) indicates that the mnemonic m is assigned the right ρ to the VCS object o by the user u.
- userHasRight(u, ρ, o, v) indicates that the user u is assigned the right ρ to o by the user v. Only a Repo Admin or VCS Admin may directly assign rights to users. We discuss the manner in which a Repo Owner assigns rights in our discussions on Repo Owner below.
- isRepoOwner(u, R, v) indicates that the user u is specified to be a potential Repo Owner of all the repositories in the set R by the user v. We say "potential," as more than one user may be designed to be a Repo Owner, and the user that creates the repository with a particular name ends up being its only owner.

We define also the following functions.

- $priv(r)$ – a Repo Admin may designate a repository to be private. What this means is that the Repo Owner is disallowed from giving out any rights to that repository (even though he may create it). This somewhat coarse-grained restriction is useful in practice. The repository r is private if and only if $priv(r) = \mathsf{true}$.

$$\mathsf{auth}(u, \mathtt{read}, o) \longleftarrow \exists \rho \in \text{RIGHTS}^{+} \;\; \mathsf{auth}(u, \rho, o) \tag{1}$$

$$\mathsf{auth}(u, \mathtt{write}, o) \longleftarrow \exists \left(\rho \in \text{RIGHTS}^{+} - \{\mathtt{read}\}\right) \mathsf{auth}(u, \rho, o) \tag{2}$$

$$\begin{aligned}\mathsf{auth}(u, \rho, o) \longleftarrow\ & (\mathsf{authByOwner}(u, \rho, o) \wedge \neg\mathsf{deniedByRepoAdmin}(u, \rho, o) \wedge \\ & \neg\mathsf{deniedByVCSAdmin}(u, \rho, o)) \vee \\ & (\mathsf{authByRepoAdmin}\ (u, \rho, o) \wedge \neg\mathsf{deniedByVCSAdmin}\ (u, \rho, o)) \vee \\ & \mathsf{authByVCSAdmin}\ (u, \rho, o)\end{aligned} \tag{3}$$

$$\begin{aligned}\mathsf{authByOwner}(u, \rho, o) \longleftarrow\ & \neg \mathit{priv}\ (\mathit{repo}(o)) \wedge \\ & \exists m\ (\mathsf{validMnemonic}(m) \wedge \mathsf{isMember}\ (u, m, o, \mathit{creator}(\mathit{repo}(o))) \wedge \\ & \quad \mathsf{authMByRepoAdmin}(u, m, \rho, o))\end{aligned} \tag{4}$$

$$\mathsf{validMnemonic}(m) \longleftarrow \mathsf{definesMnemonic}(m, \mathit{vcsAdmin}) \tag{5}$$

$$\mathsf{isMember}(u, m, o, v) \longleftarrow \mathsf{isMember}(u, m, \mathit{repo}(o), v) \vee (\mathit{br}(o) \neq \epsilon \wedge \mathsf{isMember}(u, m, \mathit{br}(o), v)) \tag{6}$$

$$\begin{aligned}\mathsf{authMByRepoAdmin}(u, m, \rho, o) \longleftarrow\ & \exists v\ (\mathsf{validRepoAdmin}(v, \mathit{repo}(o)) \wedge \mathsf{highestRightPrio}(u, m, \rho, o, v) \wedge \\ & \quad (\forall w\ \ \mathsf{validRepoAdmin}(w, \mathit{repo}(p)) \wedge \mathsf{mnemonicHasRight}(m, \rho, o, w) \wedge \\ & \quad\quad (\mathit{adminPrio}(v, r, \mathit{vcsAdmin}) > \mathit{adminPrio}(w, r, \mathit{vcsAdmin}))))\end{aligned} \tag{7}$$

$$\begin{aligned}\mathsf{validRepoAdmin}(u, r) \longleftarrow\ & (\mathit{vcsAdmin} = u)\ \vee \\ & \mathsf{definesRepoAdmin}(u, r, \mathit{vcsAdmin})\end{aligned} \tag{8}$$

$$\begin{aligned}\mathsf{highestRightPrio}(u, m, \mathtt{createBranch}, o, v) \longleftarrow\ & \mathsf{mnemonicHasRight}(m, \mathtt{createBranch}, o, v) \wedge \\ & (\forall n\ \ \mathsf{isMember}(u, n, o, \mathit{creator}(\mathit{repo}(o))) \wedge \\ & (\mathit{rightPrio}(m, \mathtt{createBranch}, o, v) > \mathit{rightPrio}(n, \mathtt{createBranch}, o, v)))\end{aligned} \tag{9}$$

$$\begin{aligned}\mathsf{highestRightPrio}(u, m, \mathtt{write}, o, v) \longleftarrow\ & \mathsf{mnemonicHasRight}(m, \mathtt{write}, o, v) \wedge \\ & (\forall n\ \ \mathsf{isMember}(u, n, o, \mathit{creator}(\mathit{repo}(o))) \wedge \\ & \quad (\mathit{rightPrio}(m, \mathtt{write}, o, v) > \mathit{rightPrio}(n, \mathtt{write}, o, v))) \wedge \\ & (\forall n\ \ \mathsf{isMember}(u, n, o, \mathit{creator}(\mathit{repo}(o))) \wedge \\ & \quad (\mathit{rightPrio}(m, \mathtt{write}, o, v) > \mathit{rightPrio}(n, \mathtt{denyWrite}, o, v)))\end{aligned} \tag{10}$$

Figure 4: A portion of our inference rules for our authorization scheme. RIGHTS^{+} = {`read`, `write`, `rewindBranch`, `createBranch`, `deleteBranch`}.

- $rightPrio(mORu, \rho, o, v)$ is a function that specifies the strict priority that the rule that assigns the right ρ has been assigned by the user v over object o to the mnemonic or user $mORu$. Rights can conflict with one another. Within a particular user's grants, this is disambiguated by this priority. Rights can also conflict across granters. This is resolved via the inference rules.

Repo Owner As with the other user-classes, we associate the following predicate with actions relevant to a Repo Owner.

- $\mathsf{isMember}(u, m, o, v)$ indicates that user u is assigned to be a member of the mnemonic m for object o by a user v.

We define also the following function.

- $creator(r)$ – the Repo Owner of r. If the repository r does not exist, then $creator(r) = \epsilon$, where ϵ is a special symbol in the range of $creator$. A repository that exists has exactly one creator.

Inferences We show an example of inferring authorized access to an object in Figure 4. The predicate $\mathsf{auth}(u, \rho, o)$ is used to indicate whether the user u has the right ρ to the object o. Figure 4 has a partial list of inference rules that are relevant for ρ = `createBranch`, and ρ = `write`. We point out that the former does not have a corresponding negative right, while the latter does.

In Figure 4, Inference (1) asserts that possession of any positive right to o implies possession of `read` to o. Inference (2) asserts that possession of any positive right other than `read` to o implies possession of `write` to o. The rights `read` and `write` are the only ones for which such implication holds. The reason is that this is the most sensible design. For example, it does not make sense of a user to have the `createBranch` right to a repository unless he is able to read and write the repository.

Inference (3) presents the cases for user u to have right ρ to object o. The right ρ may be positive or negative. The three disjuncted clauses express the three possibilities. They are as follows.

- User u has ρ to o if he is granted it by the Repo Owner, and not denied it by either a Repo Admin or the VCS Admin. "Denied" in this context is somewhat non-straightforward to determine as we have priorities associated both with Repo Admins (as assigned by the VCS Admin), and to rules within a Repo Admin's ruleset.
- The second case is that u is authorized to ρ over o by the Repo Admin and not denied it by the VCS Admin. In this case, any rules of the Repo Owner are superceded by the rules of the Repo Admin.
- The third case is that u is authorized to ρ over o by the VCS Admin, which supercedes any denials by a Repo Admin.

Inference (4) clarifies when authByOwner is true. The only way for a Repo Owner to grant a right is by assigning a user to a mnemonic (see Section 3.2). For this, the repository in which o is cannot be private. Also, the right is assigned to the mnemonic by a Repo Admin; this is the reason we introduce the predicate authMByRepoAdmin.

Inference (5) clarifies that a valid mnemonic is one that is defined by the VCS Admin. Inference (6) asserts that if user v specifies that u is assigned to mnemonic m for an object (repository or branch) that contains o, then this implies that v assigns u to m for o.

Inference (7) specifies what we mean by a right being authorized to a mnemonic m by a Repo Admin. The Repo Admin must be valid, as specified by the VCS Admin. Also, within his set of rules for the object o, the assignment of ρ to the mnemonic m must supercede (by priority) any other assignments for o that pertain to ρ. We use the example of ρ = `createBranch`, which is somewhat simpler than, for example, `write`, as the latter has a corresponding negative right, `denyWrite`. Consequently, for `write`, we need to check also whether the Repo Admin has `denyWrite` of higher priority that any rules that grants `write` to the user via the mnemonic.

Inference (8) expresses what we mean by a "valid" Repo Admin. It must be someone to whom the vcsAdmin has delegated power. Finally, Inference (9) expresses how we use priorities within the authorization rules of a Repo Admin for the case that ρ = `createBranch`. Inference (10) expresses how the right write is assigned to a user by the Repo Owner. The highest priority Repo Admin must not have a higher priority rule that assigns the user the `denyWrite` right.

4.2 Semantics

For a semantics, we specify a model, $\mathcal{M}$, in which we instantiate sets of tuples for each predicate [14]. We consider only those environments in which variables are mapped meaningfully to concrete values that are meaningful. For example, the set of concrete values, A that we associate with $\mathcal{M}$ includes A_u, the set of users of the system, and A_r, the set of all rights that we specify in Section 3.3. We associate the variable u in Inference (1) of Figure 4, for example, only with elements from A_u, and not A_r. Similarly, we assign ρ only elements from A_r.

Our scheme has a well-founded semantics. We use negation in the inference rules. Examples are in Inference (4) in Figure 4 and in inference rules that involve the negative rights. However, the negation is stratified [11]. $\mathcal{M}$ is the least fix point from applying the inference rules to the model $\mathcal{M}_0$ with which we start. In $\mathcal{M}_0$, we populate the relations that make our predicates concrete with values directly from the authorization configuration files. For example, for every user u that the VCS Admin defines, we include the tuple $\langle u, vcsAdmin\rangle$ in $\mathsf{definesUser}^{\mathcal{M}_0}$. Similarly, for every user v that associates a mnemonic m with a right ρ for object o, we include the tuple $\langle m, \rho, o, u\rangle$ in $\mathsf{mnemonicHasRight}^{\mathcal{M}_0}$.

We are able to assert also that our algorithm to compute $\mathcal{M}$ (and $\mathcal{M}$ itself) is sound and complete. Soundness comprises two subproperties. One is that if there exists an environment in which $\mathsf{auth}^{\mathcal{M}} \ni \langle u, \rho, o\rangle$ is entailed by $\mathcal{M}$, then this can be only because u is indeed authorized to ρ on o by one of the conditions we discuss in the previous section. The second soundness subproperty is that we do not infer any contradictions with respect to negative rights. That is, we never infer that a user is authorized to both `write` and `denyWrite`, or `rewindBranch` and `denyRewindBranch` to the same object. Completeness is the property that for any instance of our scheme, our algorithm to compute $\mathcal{M}$ terminates.

5. EXPERIENCE FROM DEPLOYMENTS

In this section, we discuss our experience from deploying our authorization scheme. As we mention in Section 1, our scheme has been validated in two settings: university and open-source software development.

5.1 University

We have deployed `gitolite` within a university research team for managing access to repositories that contain versioning content for code, projects, papers, proposals, courses, and private repositories for users. At present, our deployment is one year old, and it has approximately fifty repositories and thirty users. Our users include professors, graduate students, undergraduate students, and collaborators. Amongst these users, we have one VCS Admin, one Repo Admin, and the rest of the users are either Repo Owners and/or Accessors. The VCS Admin is a professor, the Repo Admin is a graduate student, all collaborators are Accessors, and the remaining users are either Repo Owners and/or Accessors. The VCS Admin delegates the responsibility of managing access to the the projects, papers and code repositories to the Repo Admin. Only the VCS Admin administers control to proposals, courses and private repositories.

We describe a typical scenario where a student writes a paper for a conference. This involves a user creating a repository, and then delegating access to other users if there are co-authors. Note that every user possesses the ability to create repositories except for collaborators. This is because we characterize collaborators as external to the university, and we require that they are given access to the repository explicitly either by the Repo Owner or one of the administrators. A student who is the first author creates a repository for the paper. This makes the student a Repo Owner of that repository. At this point, only the Repo Owner can read and write to this repository. If there are Accessors that are co-authors of this paper, then Repo Owner must explicitly add them to the READERS or WRITERS mnemonic for that repository. Notice that even after the co-authors have access to the repository, they are only Accessors; only one user can be a Repo Owner.

We provide every professor and student with a set of repositories that are private, and only accessible to them. These repositories are used for version controlling confidential documents such as recommendations and financial summaries. While a professor or student can create a repository, they are not allowed to delegate access to other users. This means users can create repositories and thereby becoming Repo Owner but they cannot further delegate. Recall from Figure 3 that a Repo Owner may delegate to Accessors. For private repositories, we do not allow Repo Owners to do this.

We also give every professor and student a *scratch* space of repositories. These repositories are used for both personal and collaborative efforts on class projects, technical reports, and independent programming projects. In our entire deployment, we use the default set of mnemonics: WRITERS, READERS and CREATORS. Users associated with the WRITERS mnemonic have read and write permissions, READERS have only read permissions, and CREATORS have read, write and rewind permissions. This means that only the creator can issue commands that delete segments of the branches.

We heavily use the ability to let users create repositories and then delegate access to other users in our deployment. Students have the ability to create paper and project repositories, and professors can create proposal and course repositories. Once a student or a professor becomes an Repo Owner, he/she can determine which other users to allow access. None of our repositories are created by the VCS Admin or the Repo Admin. As a result, a minimal amount of involvement from either administrators to manage the repositories once the initial deployment was successful.

5.2 Open-source Software Development

We describe the deployment experience of KDE as an open-source software project. Currently, KDE's deployment has approximately $1,730$ users and 130 repositories as shown in Table 1. Approximately 60 of these repositories are created by either a VCS Admin or Repo Admin. These repositories are the projects that already exist in KDE's deployment of SVN, and they are directly transitioned and setup by the administrators. The remaining repositories are created by Repo Owners. These are mainly used for the beginnings of new projects, project notes, and configuration files. KDE's policy gives read and write permissions to repositories to developers, and read access to all accessors. There are some repositories that are private that have specific access control rules. Examples of private repositories are those that are exclusively used by the system administrators and board of directors.

KDE makes extensive use of the ability to allow users to create repositories, and delegate permissions using mnemonics. Their deployment uses four mnemonics: WRITERS, MANAGERS, DANGERS, and CREATORS. WRITERS associates users with the permissions to read, write and create branches, MANAGERS with read, write, create and delete branches, DANGERS with read, write, create, delete and rewind branches, and CREATORS with all the permissions of DANGERS including the ability to create repositories. This deployment ensures that CREATORS and DANGERS have full control over the repository with the exception that only CREATORS can delegate. A Repo Owner associating a user with DANGERS effectively allows the user to have the same permissions as the owner (aside from delegation). This mnemonic allows creators of the repository to provide equal permissions to the owner of the repository to other users. MANAGERS, on the other hand, are not allowed to rewind or force push onto the branches. Such repositories are commonly used for starting up new projects that are typically features of existing projects.

The current deployment heavily employs the mnemonics, creation and Repo Owner delegation. While only a small subset of KDE's software collection is currently managed using `gitolite`, the number of users and repositories will increase significantly once the transition to Git is complete. System administrators responsible for this transition and deployment mention that after the initial deployment, the administration using `gitolite` requires minimal effort. Their overall experience in using `gitolite` is positive.

5.3 Some Organizations that use `gitolite`

In Table 1, we list some of the organizations that use `gitolite`, and who were willing to share their deployment information to the public domain. The number of repositories and users in Table 1 are approximated and were taken at the time of collecting the survey data from the system administrators. We qualitatively describe the usability of `gitolite` in the last column. A good rating means that the administrators spent approximately twenty minutes a day to administer `gitolite`. Of this, they mentioned that most of it was spent in adding users and not any significant alterations to the access rules.

5.4 Performance of Access Enforcement

Figures 5a and 5b present the time for access checks when using `gitolite`. We use Fedora Linux's configuration, which contains approximately $11,600$ repositories with $1,000$ users as shown in Table 1. Our experiments are run on an Intel dual-core Corei7 1.2Ghz with 8GB of DRAM. We measure the elapsed time on the server hosting Git, and `gitolite`. Therefore, we do not incorporate the network latency that often dominates.

We describe the performance of `gitolite` under two scenarios. The first scenario in Figure 5a shows the read, write and combined access check times when using `gitolite`. We vary the number of repositories from 100 to $11,600$ repositories. The graph separates the access check time for readDecision and writeDecision as described in Figure 2, and it shows a line that combines the two labelled combined. The access time for readDecision is less than that of writeDecision.

We expect this result because the permission checks for readDecision are simple. However, the checks for writeDecision are more complex, which requires checks for a variety of permissions such as branches, files, and folders. Figure 5a shows that it takes approximately 0.16 seconds to perform a combined read and write access when a VCS has $11,600$ repositories. $11,600$ repositories is a large number of repositories, and yet the access check times are less than half a second. We recognize that measurements under a second may contain jitter such as operating-system context switches, but it allows us to claim that the access check times in `gitolite` are negligible for large numbers of repositories.

In the second scenario shown in Figure 5b, we evaluate the effect of increasing the number of access rules per repository on the access check times. We only perform this on one repository because an access is only done for a single repository. Our experiment varies the number of rules for the repository from 100 to $2,000$ rules, and the combined access check time for $2,000$ rules is less than 0.20 seconds. Note that the typical number of rules per repository in our experience is around 10 rules. This means that the typical access check time is also negligible.

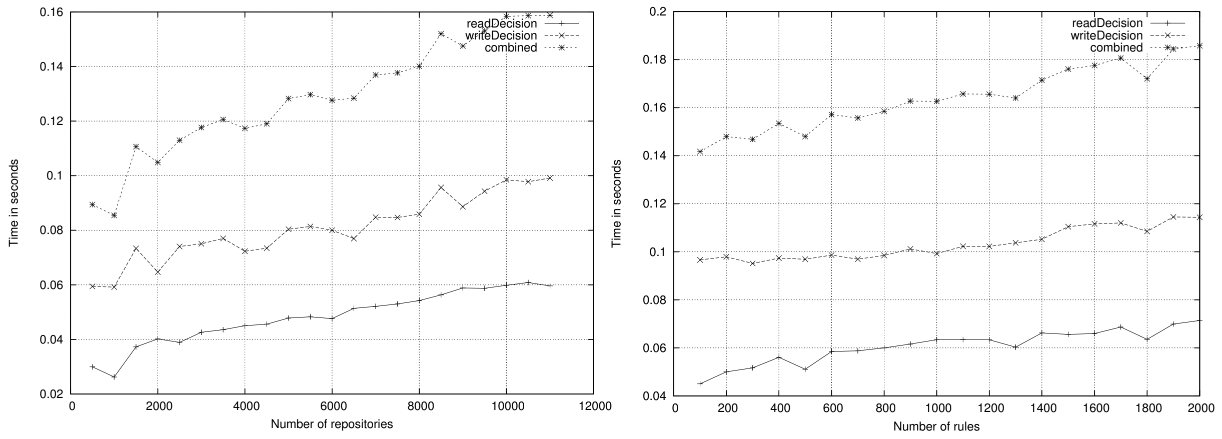
6. RELATED WORK

A number of authorization schemes have been proposed in the research literature, and a comprehensive discussion is beyond the scope of this paper. Our scheme pulls together several aspects that have been proposed before, such as the use of ACLs, delegation and resolving conflicts between rules using some sort of prioritization. In the context of VCSes, we are unaware of work on authorization that has been published in a research venue. The book by Pilato, Collins-Sussman and Fitzpatrick [22] on SVN includes a discussion on SVN's authorization scheme. It is a good example of traditional authorization schemes as used in VCSes in the past. In our work, we also use logic for modeling our scheme. The use of logic has a long history in access control, including the work of Abadi et al. [4], and more recent work [6, 10, 18, 26]. Our interest is not in developing a new logic for authorization, but rather in using first-order logic for precision, and for correctness properties.

We are primarily interested in the expressive power of our scheme relative to schemes that have been proposed in the past. Our objective in designing the scheme was to have a scheme that is sufficiently flexible (expressive) so administrators can configure it as they need to in practice, yet not make it so complex as for it to be unusable. Our deployment experiences (see Section 5) suggest

Setting	Name	# Repo.	# Users	Usability
University	MIT	20	40	Good
	University of Waterloo	50	30	Good
Open-source	KDE Project	130	1730	Good
	Fedora Project	11600	1000	Good
	Gentoo Linux	200	1000	(unknown)
	Racket	100	34	Good
Industry	Reaktor	100	40	Good

Table 1: Organizations that use `gitolite`.



(a) vs. # of repositories

(b) vs. # rules

Figure 5: Performance of `gitolite`— access check times.

that we have hit the right balance. Consequently, the question as to whether our scheme is more expressive that another scheme is not of interest in the context of this paper.

One may pose the question in the other direction: whether there exist schemes from the literature that can capture our scheme. To our knowledge, the answer to this question is no. At the minimum, we assert that there does not seem to be a natural encoding of our scheme in other schemes. For some example schemes from the literature, we discuss the basis for our above assertion.

Consider, for example, DAC schemes that are used to protect filesystems. There are two differences between such schemes and ours. One is that resources that are protected in filesystems are different, and have different relationships with one another than in VCSes. The second is that typically, such schemes, such as the ones in POSIX-compliant systems [27] and Windows ACLs [29, 30], have only two levels of administrators: a superuser ("root") and the owner of a file or directory. The owner has full discretion in handing out rights to resources that he owns. One may argue that the notion of a mnemonic in our scheme is the same as the notion of a group in such systems. However, there are some important differences. One is that in existing filesystems, the superuser not only defines a group (the mnemonic), but also assigns users to the group. We have adopted the approach wherein the VCS Admin only defines the mnemonic, but the assignment of users is discretionary to the Repo Owner. This has significant consequences to the administrative ease of the system.

There have also been administrative schemes for RBAC such as ARBAC97 [24], SARBAC [8] and UARBAC [17]. It may be argued that a mnemonic in our scheme is exactly a role as perceived in such schemes. However, there are some important differences between such schemes and our scheme. One is that we have three levels of administrators where these schemes have two only. It is unclear that our mid-level administrators (Repo Admin and Repo Owner) can be captured using the constructs provided within such schemes. Also, the rules by which a user (a Repo Owner) assigns users to mnemonics is discretionary, whereas in such schemes it is guarded by preconditions on users' memberships to other roles.

There are also schemes that are presented as logics, but may be perceived as authorization schemes in their own right; an example is DynPal [6]. It is quite possible that DynPal can capture our scheme. However, an issue with such a scheme is that it is too expressive. For example, we observe that DynPal permits delegations of unbounded depth. Furthermore, policy verification (safety analysis) is undecidable in general in DynPal. We do not delve into policy verification in the context of `gitolite` in this paper; however, we conjecture that it is decidable, and even tractable, even though our domain is infinite.

7. CONCLUSIONS AND FUTURE WORK

We have presented aspects of our design and implementation of `gitolite`, an authorization scheme for Version Control Systems (VCSes). Access enforcement relies on Access Control Lists (ACLs), and there is a new syntax that we have developed for specifying authorization rules that are compiled into an ACL. In ongoing work, we are using first-order logic to express our scheme precisely, and also give it a precise semantics, and to assert soundness and completeness properties. Our scheme has several deployments, and we have discussed our experiences from such deployments. We have also discussed how access enforcement works, and its performance relative to the number of repositories and ACL entries.

There is considerable scope for future work. One is to conduct a comprehensive usability test of our deployments to find out whether `gitolite` is indeed as usable as it can be. An aspect for us to assess in this context is whether we can fine-tune the balance between flexibility and usability. Another topic for future work regards an expression of our scheme precisely. It may be possible to use some automated proof technique and maintain the formal specification of our scheme in such a syntax even as it evolves. We plan also to explore the issue of policy verification in the context of our scheme.

8. REFERENCES

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

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

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

[4] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Oct. 1993.

[5] P. Ammann and R. S. Sandhu. Safety analysis for the extended schematic protection model. In *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pages 87–97, May 1991.

[6] Moritz Y. Becker. Specification and analysis of dynamic authorisation policies. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, July 2009.

[7] 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.

[8] Jason Crampton and George Loizou. SARBAC: A new model for role-based administration. Technical Report BBKCS-02-09, Birbeck College, University of London, UK, March 2002.

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

[10] D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. A linear logic of authorization and knowledge. In *ESORICS*, pages 297–312, 2006.

[11] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for logic programming. *Journal of the ACM*, 38(3):620–650, 1991.

[12] 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.

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

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

[15] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of Role-Based Access Control policies. *IEEE Transactions on Dependable and Secure Computing(TDSC)*, 5(4):242–255, Oct. 2008.

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

[17] Ninghui Li and Ziqing Mao. Administration in Role-Based Access Control. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, pages 127–138, New York, NY, USA, 2007.

[18] 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.

[19] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM*, 52(3):474–514, May 2005. Preliminary version appeared in *Proceedings of 2003 IEEE Symposium on Security and Privacy*.

[20] N. Li and M. V. Tripunitara. On Safety in Discretionary Access Control. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[21] 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.

[22] M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly, Sept. 2008.

[23] 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.

[24] 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.

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

[26] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (nal): Design rationale and applications. Cornell Computing and Information Science Technical Report, Sept. 2009. Available from `http://www.cs.cornell.edu/People/egs/papers/nal.pdf`.

[27] Security Working Group, IEEE Computer Society. IEEE 1003.1e and 1003.2c: Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) and Part 2: Shell and Utilities, draft 17. Available from `http://ece.uwaterloo.ca/~tripunit/Posix1003.1e990310.pdf`, October 1997.

[28] M. Soshi. Safety analysis of the dynamic-typed access matrix model. In *Proceedings of the Sixth European Symposium on Research in Computer Security (ESORICS 2000)*, pages 106–121. Springer, Oct. 2000.

[29] Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in windows nt. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, SACMAT '01, pages 87–96, New York, NY, USA, 2001.

[30] Michael M. Swift, Anne Hopkins, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control for windows 2000. *ACM Trans. Inf. Syst. Secur.*, 5:398–437, November 2002.

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