# Administrative Role-Based Access Control on Top of Security Enhanced Linux

**Ayla Solomon**

Submitted in Partial Fulfillment

of the

Prerequisite for Honors

in Computer Science

June 2009

**Abstract**

In a time when the Internet is increasingly dangerous and software is increasingly complex, properly securing one's system has become a widespread concern, especially for system administrators. There are a number of ways to secure a Linux system; these methods vary in effectiveness and usability. Regular Linux discretionary access control is suitable for most mundane tasks and is easy to use, but it fails to confine insecure programs that run with administrative powers and is difficult to scale. Security Enhanced Linux (SELinux) is an extension to the Linux kernel that provides fine-grained controls for preventing malicious agents from exploiting software vulnerabilities. It can be very effective, but its policy language is extremely low-level and its paradigm difficult to understand and use. This makes it difficult to map a high-level idea of a security policy to what needs to be done in SELinux, which rules out its use to a vast majority of people with Linux systems, including system administrators, though many attempts have been made to make it more usable.

I have designed and partially implemented a system on top of SELinux to implement administrative role-based access control (ARBAC), which is not available on any Linux system. ARBAC is an intuitive, flexible, and scalable security paradigm well suited for user confinement, especially for very large systems with many users. ARBAC's main advantages are not only in its intuitive structure but also its built-in administrator confinement mechanism: its very structure promotes separation of privilege amongst administrators. The system I have designed consists of a high-level language for specifying ARBAC policies that will compile mainly to low-level SELinux policy language, using its existing framework to enforce the security policies. However, because of the limitations of what SELinux can express and enforce, the system also includes Linux utilities to enforce administrative constraints otherwise relegated to enforcement by convention. The result is a language and system that can easily express high-level security policies and have them enforced in a fine-grained way, helping prevent Linux security vulnerabilities from becoming exploits.

# Contents

# Chapter 1

# Introduction

Linux users and system administrators are most familiar with Linux's native discretionary access control (DAC) mechanism, in which object owners (and the superuser `root`) may define all of the object's security aspects, of which there are only a few: read, write, and execute for the owner, group, and world. Though these permissions are not always the most intuitive ways to define security on an object (what does it mean to execute a directory?), it is very simple and easily configurable for each object. However, it can be tedious to administer a large system based on this because user and group permissions must be set per-object. The worst part of this system is not its tediousness, but its well-documented security weaknesses, most especially centered on the superuser `root`. Every program and daemon that needs some form of administrative access (and some that do not) run as `root`, giving them unrestricted access to the entire system, no matter what they actually do. Unless a program is very well secured, it can be possible to exploit it to gain unrestricted access to (or 'own') a system with something as common as a buffer overflow exploit [One96]. This is called *privilege escalation* and it is enabled by the all-or-nothing model of DAC administrative privileges.

Because of this, there are other available security modules that implement mandatory access control (MAC), in which a system-wide policy defines who may modify what and how, without regard to owner (unless defined in the policy). In a way it is discretionary on the part of the policy writer, but owners no longer have special status for their own files and `root` can be restricted. This kind of system policy can be very effective at eliminating or limiting damage done by security weaknesses such as privilege escalation, but it is historically very inflexible, as the policies are under sole control of one or a few administrators and extremely low-level. Security Enhanced Linux (SELinux) is the most well-known and widely available security module for many versions of Linux. It exemplifies the strengths and traditional problems with MAC while trying, though rarely succeeding, to make MAC tractable for the average system.

Administrative role-based access control (ARBAC) is a high-level, abstract security paradigm, focusing on users and their permissions. It organizes permissions into roles and

roles into an inheritance hierarchy. Users are assigned to roles and get all of permissions of the roles to which they are directly assigned and to the roles of which they are implicitly members due to the role hierarchy. Administration takes the form of controlling which roles users are assigned to and promotes separation of privilege; administrative privileges are associated with administrative roles, creating the same general structures as regular hierarchies and not necessarily separate from them. This paradigm, if implemented correctly to be able to define MAC policies[1], could remedy Linux MAC's usability weaknesses, bringing together an intuitive, high-level design, the specificity needed to properly secure a system, and a dynamic and well-defined administration model.

I have designed and partially implemented a system, called Tolypeutes[2], that creates a limited form of ARBAC for Linux machines equipped with SELinux. It starts with a high-level language in which to specify ARBAC policies. The language is parsed and converted to SELinux policy modules and to configuration files used by other parts of the system. The SELinux modules enforce most of the policy and the other parts utilize the configuration files to enforce administrative constraints. I have also written a utility to expedite the process of adding Linux users with unique SELinux identifiers, which is required for the generated SELinux modules to adequately confine them. The result is a security system that can utilize SELinux's strong enforcement, should be radically more usable than SELinux, and can be dynamically administered. All of these would ease the lives of Linux system administrators that want strong security. Administering SELinux alone comes with a lot of overhead in addition to knowing enough to write and compile a module.

The rest of this document is divided into the following chapters:

**Chapter 2: Background** This chapter provides background information about ARBAC and SELinux that is essential to understand both what Tolypeutes does and why I feel that it is important. It goes into detail about standard role-based access control (RBAC) and why ARBAC is advantageous. It includes a detailed description of SELinux's architecture, history, and design, and why it is both a wonderful and terrible thing.

**Chapter 3: Related Work** I am not the first person to think that SELinux needs serious help to become usable. Others have written languages on top of it that compile down to policy code and have written tools to support easier policy management. I also include notes on some security packages that are not related to SELinux.

**Chapter 4: Tolypeutes: ARBAC on Top of SELinux** This chapter describes the design and partial implementation of the Tolypeutes system in great detail. It describes

[1] ARBAC is flexible enough in theory to express both MAC and DAC policies [OS00].

[2] *Tolypeutes*, pronounced taw-lih-**pyoo**-teez in Latin (though it appears to have a Greek rather than Latin root despite its Latinization), is the genus name of the three-banded armadillos. I chose it because they are small but very secure creatures.

the purpose, design, strengths, and weaknesses of all six modules that make up the Tolypeutes system.

**Chapter 5: Conclusions and Future Work** This chapter summarizes what I've done and why. It also describes possible extensions and fixes to the Tolypeutes system.

# Chapter 2

# Background

In order to understand the finer details of and motivations for Tolypeutes, it is necessary to understand both the model I am reaching towards (ARBAC) and the tools with which I am working (SELinux). ARBAC has two distinct, though closely related, components: the basic role-based access control (RBAC) core and the administrative addition that makes it ARBAC. SELinux is a very large and complex system, implementing a fairly simple paradigm in a very complex and multifaceted way. It is both an extremely effective security mechanism when used and managed correctly and incredibly difficult to use and manage. This difficulty usually cancels out its benefits. If only there were a way to combine it with a highly usable upper layer!

## 2.1 RBAC

Role-based access control (RBAC) is based upon abstract roles arranged in a hierarchy. Users are associated with a set of roles and each role has its own set of specific permissions. The role hierarchy affects how users may access the system; in addition to the permissions from roles to which users are directly assigned, users inherit permissions from roles dominated by those to which they are assigned. This simple fact is the essence of RBAC.

One advantage of RBAC is in its extra level of indirection: assigning permissions directly to users can result in a large policy or access control list (as seen in DAC). Assigning permissions to roles and roles to users is simpler because roles tend to have many fewer permissions than users, and assigning roles to users is simpler still as users tend to have only a few roles and roles may be assigned to a number of users, so there are fewer total assignments. Roles may also be arranged in a hierarchical structure, making it even easier to assign users' permissions. The rule set for RBAC is very small:

- User-role assignment
- Role-permission assignment
- Role-role assignment (inheritance)

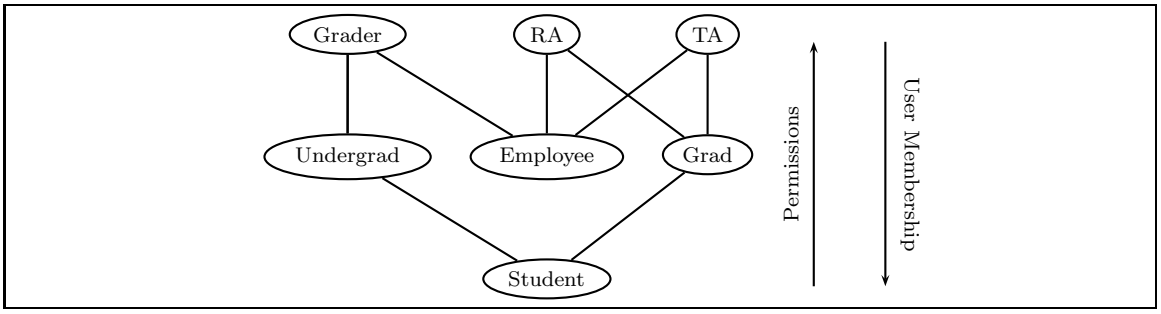


Figure 2.1: An example of a small RBAC policy role hierarchy, modeled after the student body of a university.

These three rule types are all that's required to create an RBAC policy.

In an RBAC diagram, as shown in Fig. 2.1 and with permissions in Fig. 2.2, permissions flow up and user membership flows down. Unique sets of permissions are associated with each role, and users assigned to roles get those roles' permissions and in addition, all of the permissions associated with roles below them in the hierarchy. Concretely, a user assigned to the role Grader gets all of the Grader permissions and also those of Undergrad, Employee, and Student. A Grader's permissions, then, would be write the Gradebook from the Grader role, use the Timesheet and Register programs as a Student and Employee, and read the Gradebook as a Student. The user would not get the permissions associated with any role in a disjoint part of the hierarchy or any that is upstream of a downstream role. This hierarchy is part of the running University policy example, derived from [GLS+09], that will illustrate concepts throughout the document.

These inheritance graphs may have any shape (cycles are meaningless, but possible) and any number of hierarchical levels; it is not restricted to trees, and forests are a common shape. They are essentially just directed acyclic graphs imbued with policy-specific meaning.

## 2.2 ARBAC

Administrative RBAC (ARBAC) is simply the addition of constrained administration of RBAC using RBAC. In addition to the standard user role set, there is a set of administrative roles that have permission to change how users interact with the policy, whereas regular users do not. The ability to change more integral parts of the policy, such as the role hierarchy and permissions associated with roles, is not specified in ARBAC and depends upon implementation. Nonetheless, the ability to change user/role relations is a powerful one. Because of this, a central idea of ARBAC is administrative separation of privileges. To accomplish this, the specification of the administrative part of an ARBAC policy consists of two kinds of rules:

- User assignment rules
- User revocation rules

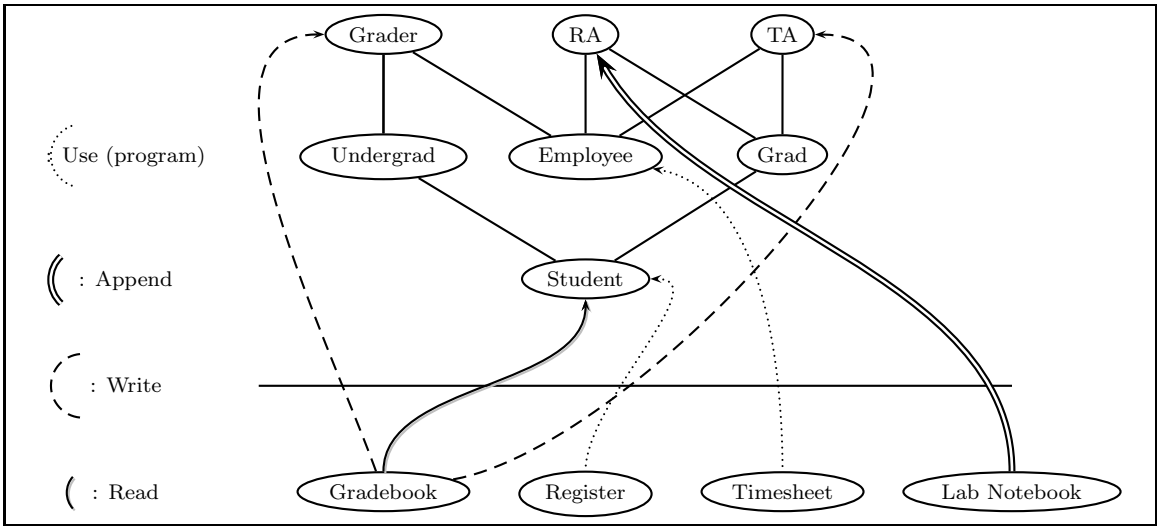


Figure 2.2: Permissions for the roles in Fig. 2.1 in a simple, high-level syntax. These abstract permissions are purely for demonstrative purposes and do not necessarily represent concrete permissions on a computer system.

These two rules encompass everything an RBAC administrator needs to be able to do — assign users to roles and unassign them. Assignment/revocation powers that are not mentioned are not allowed. The fields of these rules include the administrative role allowed to execute the change, the role it will be granting access to, and the roles that a user must have for the administrator to execute the change. The administrator may modify the roles of any user that satisfies the prerequisites field. It is important to note that administrative permissions do not only pertain to regular user roles; administrators administer each other in the exact same way.

ARBAC uses RBAC to administer RBAC in that administrative roles are arranged in the same sort of hierarchy structure as regular roles. Administrative role hierarchies have the same permission inheritance mechanism as regular role hierarchies, so senior administrative roles inherit user assignment/revocation privileges from their juniors in addition to any other normal permissions. Indeed, there is no need to separate administrative and regular hierarchies; they can be mixed with no loss of security and no violations of the ARBAC model, but such policies are much more difficult to reason about, and certainly to analyze [SYRG07]. For reasons of simplicity, the running university policy example, and all policies implemented by the Tolypeutes system, will enforce Separate Administration, a constraint in which administrative and regular role hierarchies must be entirely separate. A very simple administrative branch of the university policy is shown in Fig. 2.3, and with special administrative permissions added in Fig. 2.4.

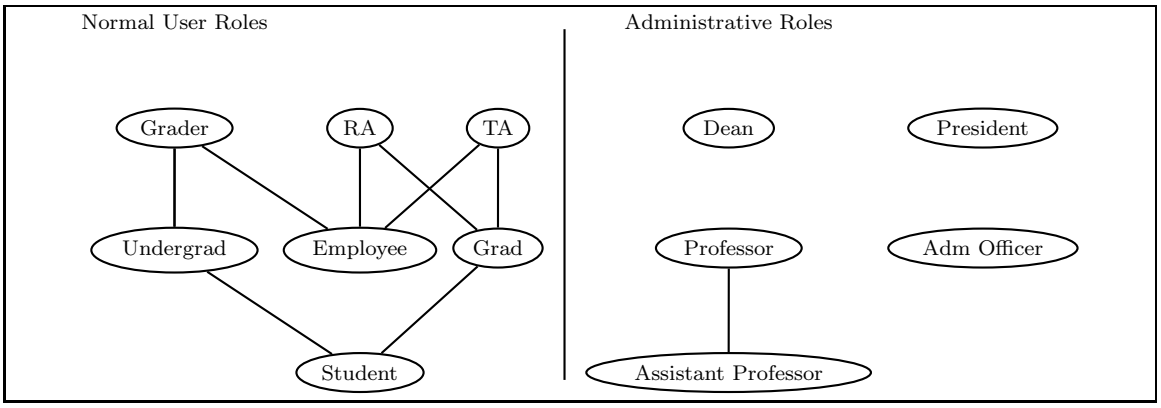


Figure 2.3: An ARBAC example building off that in Fig. 2.1. The administrative roles are presented here as completely separate from the regular user roles, but this needn't be the case. Administrative roles may dominate regular roles (and so get their permissions), may have their own set of regular permissions, and may have administrative privileges on any role, including other administrative roles.

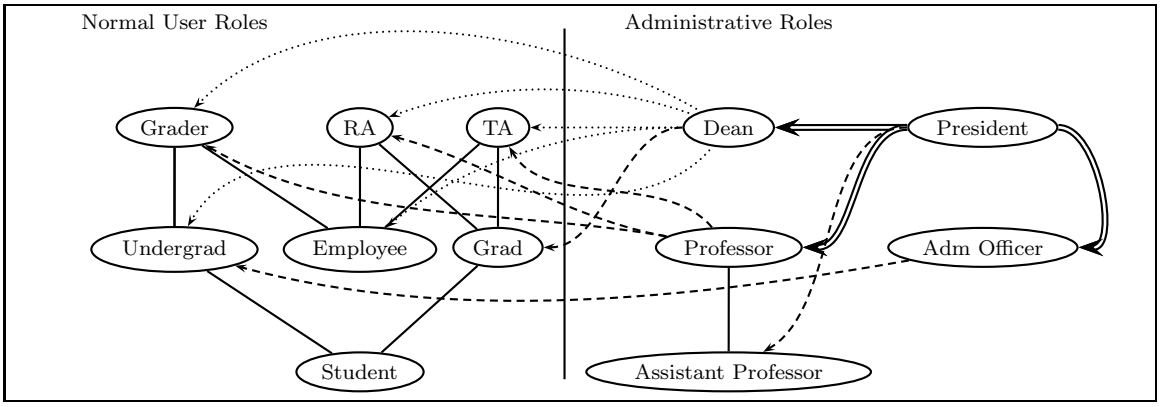


Figure 2.4: The ARBAC administrative branch from Fig. 2.3 with illustrated administrative permissions. The administrative roles have special permissions pertaining to how they administer other roles. Doubled lines indicate that the administrator may assign users to that role, but not revoke users from it. Dotted lines indicate that the administrator may revoke users from the role, but not assign them. Dashed lines mean that the administrator may do both. One extremely important aspect missing from this picture is preconditions: sometimes, administrators have permission to assign/revoke a user's role only when the user has or doesn't have certain other roles. An example is the Professor assigning and revoking from Grader: the Professor may only assign a user to Grader if the user is already an Undergrad.

## 2.3 SELinux Motivation

Security Enhanced Linux (SELinux) is a MAC implementation that utilizes the Linux Security Module, which provides kernel support for MAC on Linux, the main goal of which is to fix MAC's inflexibility problem. Type enforcement (TE) is the main security model chosen for SELinux because of its supposed flexibility, in addition to extreme specificity. Type enforcement works on the principle of type interaction — everything on the system has a type; they are arbitrary labels that all objects must have. Access decisions are needed when a user or process type wishes to interact in some way with another object, be it another process or some sort of file. Somewhere in the policy, there could be a TE rule governing what sort of permissions the acting type has on the object type. If there is, the permissions for that interaction are examined and the access is denied or allowed. If there is no such rule, the access is denied by default. It is important to emphasize that *everything* on the system has *a* type, though not necessarily a *unique* type.

A simple example of TE and type interaction is that of the `passwd` program accessing the shadow file. The `passwd` program's process has a unique type, say `passwd_t`, and the shadow file, being an important file with necessarily tight security, also does, say `shadow_t`. The `passwd_t` process should be able to read and modify the `shadow_t` file, though not delete it entirely or access any other, non-`shadow_t` file (this is for the sake of simplicity; the `passwd_t` process would need access to a great number of other file system objects not mentioned here in order to access the terminal, shell library files, etc.). This sets up the makings of some TE rules: the `passwd_t` type should have read, write, append, get attributes, and set attributes permissions on the `shadow_t` type. In addition to these obvious permissions is an SELinux construct that is both more complicated and guarantees better security than traditional DAC: the process type transition. The user shell process forks the `passwd` process when it executes the binary file (distinct from the process), which has type `passwd_exec_t`. The new process has a different set of permissions from the shell process, so it can't inherit them from the shell. That is why the spawned process must have its own type (`passwd_t`) with the permissions stated above. When the `passwd` binary is executed and the process started, it starts and dies with its own type and permissions, never affecting the shell process that created it. Without this set of permissions, the type interaction would fail and users would be unable to change their passwords. Fig. 2.5, from [MMC07], is the SELinux translation of this limited `passwd` security.

In the policy that surrounds the example in Fig. 2.5, it is assumed that the type `user_t`, the type for a user's shell process, is allowed to do many more things than executing the `passwd` program (such as logging in, creating files, etc.), but the types `passwd_exec_t`, `passwd_t`, and `shadow_t` are, as far as this example goes, fairly complete. The difference between the `passwd_t` type and the `passwd_exec_t` type is that the latter is the binary file in which the `passwd` program is stored, whereas the former is the `passwd` process. Should other processes need access to the `shadow_t` or `passwd_t` types, the access could be specified.

```
type user_t; #Type of a user's shell
type passwd_t; #Type of the passwd process
type passwd_exec_t; #Type of the passwd program file
type shadow_t; #Type of the shadow file

#Allow the passwd_t process to deal with the shadow file
#ioctl stands for I/O control--it usually has to do with device drivers.
allow passwd_t shadow_t : file {read write getattr setattr ioctl lock};

#Allow the user to execute the passwd program file
#getattr is needed here to stat the passwd binary file,
#which is needed to determine all sorts of important
#file attributes.
allow user_t passwd_exec_t : file {getattr execute};

#Allow the passwd program entrance the passwd process
#This feels backwards to the other allow rule formats because
#the thing being entered is the first attribute of the allow
#rule and not the second, but it makes sense because the first
#is still the domain and the second the object.
allow passwd_t passwd_exec_t : file entrypoint;

#Allow the user process to transition to the passwd process
allow user_t passwd_t : process transition;

#Makes it so that the type transition between the user and passwd
#process is made the default when the passwd program file is executed
#and does not need to be explicitly requested by the user.
type_transition user_t passwd_exec_t : process passwd_t;
```

Figure 2.5: SELinux security for the `passwd` program, an annotated example from [MMC07].

But why would this SELinux policy be needed in addition to the DAC security already found for the `passwd` program? The reason is privilege escalation. While using the `passwd` program, a DAC user is `root`, the unconfined superuser on the system. If the `passwd` program file were compromised or replaced with a malicious version, malicious things would be allowed to happen in regular DAC Linux because the user, while in the `passwd` process, is root. However, in SELinux, the type is just `passwd_t`, which can only read and write the shadow file and do nothing else. If other things are attempted, they will fail. This is why SELinux is, despite all of its complexity, a very good thing.

## 2.4 SELinux Architecture

SELinux is split into three parts: the security server, object managers, and the access vector cache (AVC). The security server is the repository of all of the SELinux rules specified in the policy. Object managers are what sit between system objects and processes that want to access them in any way. When an access is requested, the manager of the object to which a process wants access asks the security server if their interaction is allowed by any rule. If so, access is granted. If it is specifically denied or unspecified, it is disallowed. The AVC is like any cache — it stores these decisions in case the same question is asked again, speeding up the look-up time. If each instance of some request must be answered directly by the security server, which could be the case, the AVC is never consulted and cannot confer a

speed benefit. Fig. 2.6 illustrates the SELinux architecture.

The way that SELinux, and subsequently the kernel, knows what objects are what types is by what's called *extended attributes*. For files, attributes are found in the inode and include, amongst other things, file name, permissions bit string, inode number, size, creation date, and last access. Processes also have a finite number of attributes. Extending the attributes means adding another field; the extra SELinux field is the security context, a combination of an object's user (for non-processes and most daemons, the ruser is the generic default `system_u`), role, and type. For the most part only the type is used in access decisions. For files, these are called file contexts and are specified in special file context files for each SELinux policy module. Each object may only have a single security context at a time, but the context may change based on policy rules. Extended attributes have some consequences: it increases the size of the inode, which makes the inode larger and take up more file system space, but it also puts control entirely into the hands of the kernel, which is a security boon. Having the kernel own the extra security field for all objects makes it less vulnerable.

One important thing to emphasize is that SELinux can only restrict access further — nothing denied by standard Linux DAC can be allowed by SELinux. Indeed, when Linux bars an action, SELinux is never consulted. If a system administrator wishes to audit all access denials of a certain type, he or she will be unable to do so if Linux denies accesses and not SELinux. In addition, SELinux has three modes that affect how it works: enforcing, permissive, and disabled. Enforcing mode means that the current policy is enforced on the system and access denials are not only logged, but also actually denied. Permissive mode means that access denials, based on the currently installed policy, are not denied, just logged. Disabled mode means that no policy is installed, object attributes are not extended and labeling does not take place, and SELinux is not active at all. It is never consulted in access decisions and logs nothing.

## 2.5 SELinux History

The original example policy[1] version of SELinux required that every rule for every program, process, file, etc. be written in one single file to be compiled and loaded into the kernel. Every type, role, user, macro, and attribute name was global. This has obvious modularity issues and required that sysadmins modifying a policy have global knowledge of the entire policy, even kernel security. Obviously not ideal, this paradigm of the monolithic policy was abandoned and a modular policy paradigm was adopted to offset the difficulties of policy modification and creation. The second example policy version allowed modules to be later concatenated into a single monolithic policy, easing writing, but names were still global and it still required global knowledge. Revised again, the new reference and targeted

[1]The example policy, created by the NSA, was the first system policy shipped with SELinux.

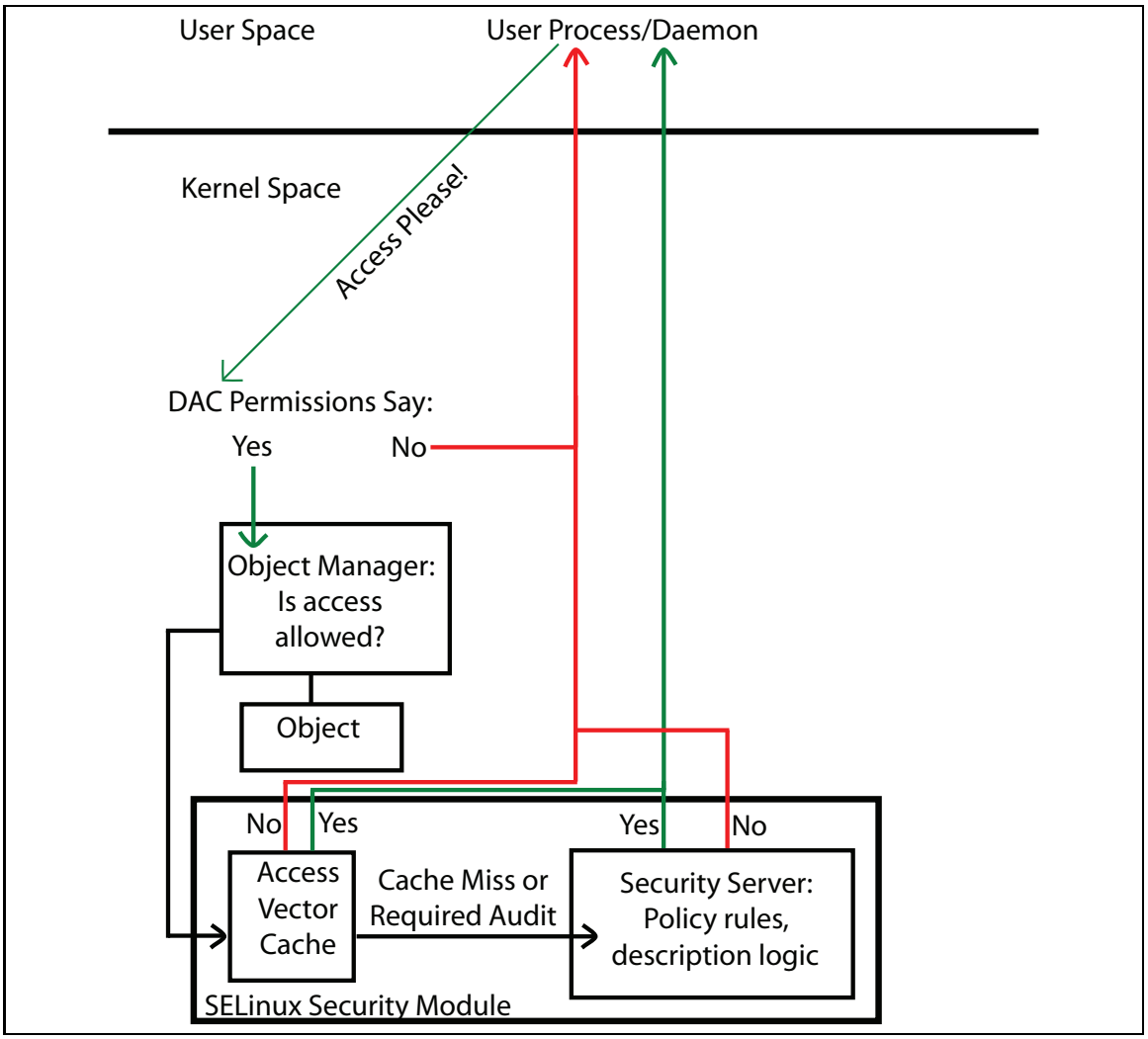


Figure 2.6: Overview of SELinux architecture. When a process in userspace wants access to any object, the access is first checked by regular Linux DAC. If it is allowed, only then is SELinux consulted. The object manager sitting on top of whatever object is requested asks the security server/AVC and from there the access decision is handed back up to the process.

policy version of SELinux allows modular design with a class-like interface structure and a non-global name-space. This was done by Tresys Technologies through a complex series of macros [MMC07]. While it does improve the SELinux landscape a great deal in terms of usability, it does not quite reach the previously stated design principles due to implementation details that are hidden but nonetheless need in some cases to be known. One example is the `unpriv_user_template()`, which creates type names that may collide with type names already defined in the module. It is therefore advantageous to look into the module in which the macro is defined and know exactly how it works and what names it creates.

While the macro superstructure of reference policy SELinux hides the implementation of kernel and other services policy, it does not make it so a policy writer may be any less intimately knowledgeable about each module. To write any policy module using the predefined macros, one has to both know what they are, exactly what they do, and where to find them. An example is that the `user_u` and `user_r` standard user and role are very important to the policy, allowing users on the Linux system but unspecified to SELinux to log into a system and do standard things. A suite of permissions are bound to the `user_r` role that allow all of these standard actions. When specifying special permissions for a user, it is very useful to be able to assign the same standard permissions in addition to nonstandard ones, which is a useful aspect of macros. One advantage (perhaps the only advantage) to the older global name-space of all of the modules was that `user_r` and `user_u` could be referred to directly and from anywhere, meaning that the following was an easy way to create a custom user with standard permissions:

```
user jdoe roles { custom_r user_r };
```

This way, `jdoe` has custom permissions specified as part of the `custom_r` role as well as standard permissions granted by `user_r`. In the reference policy[2], this is impossible and standard permissions are granted through macro templates. These templates define their own types that may collide with types defined by the policy writer, and without looking at the implementation details of the templates, it can be difficult to predict how its types will interact with writer-defined types. Therefore, even though they attempt to hide implementation details, they fail because it is either required or advantageous to be extremely familiar with them.

Today, Red Hat and other Linux systems still ship with SELinux Reference/Targeted policy installed. As far as I can tell, however, most people with those systems do not try to use it, or if they do, they are not, for the most part, developing new modules so much as using the predefined policy with as few modifications as possible. This becomes a problem when its standard configuration interferes with how one program or another works (which can be

[2]The reference policy was developed by Tresys Technologies; it is equivalent to the Targeted Policy. It seeks to confine only a sample of critical/vulnerable daemons, as opposed to the example policy, which was more far-reaching.

common on larger or more heavily modified systems), making necessary either modification of the policy or disabling SELinux. Because of the difficulty of modifying the policy, most people choose to disable SELinux. It is a shame that such a powerful system has fallen to the wayside when a little extra usability may be able to revive it, and effectively secure so many systems.

## 2.6 SELinux and ARBAC

SELinux does support a form of RBAC (not ARBAC) based on type enforcement, as seen in Fig. 2.7 and Fig. 2.8, except for static mutually exclusive roles (SMER) constraints, which mean that no one user may be assigned to both rules in the constraint at the same time. It is impossible to express SMER constraints in SELinux in such a way as to have SELinux enforce them using the role or type constructs that SELinux has. SELinux RBAC has an additional level of indirection in its implementation: roles are associated with a set of types rather than a set of permissions and the types are associated with permissions. For example, the user `user_u` has, possibly amongst other roles, the role `user_r`, which has, possibly amongst other types, type `user_t`, which is then associated with permissions. So the flow is `user_u`–`>user_r`–`>user_t`–>permissions. This means that in SELinux, RBAC is not the simple, intuitive construct it can be; a policy writer cannot envision only an RBAC policy if that is what he or she wants to create. There must be an entire underlying TE policy, with all of the possible type interactions specified and possibly interactions with the rest of a predefined reference or targeted policy. This is not to say that TE is a bad security paradigm, or inherently confusing. It certainly isn't; it is the implementation choices made by SELinux creators that make it difficult.

Roles as an extra level of indirection in SELinux does not, as far as I can tell, give an added benefit. If there were no roles, types could be associated directly with users with no apparent losses of expressiveness since special role allow and transition rules have been deprecated. As it stands, roles serve to group types before being assigned to users and security contexts can only have one role at a time and change roles only by explicitly requesting it via the `newrole` SELinux program. This limits the utility of using roles as an extra grouping mechanism.

Fig. 2.7 and Fig. 2.8 show a very small RBAC policy that I wrote, part of the University policy, using SELinux roles to represent roles. This code compiles, but it is as yet untested due to problems noted in the Intricacies and Errata section at the end of this chapter. Special considerations are noted in comments (lines that start with `#`). Overall, this extremely simple module was relatively difficult to create. It took me a long time to find out that user declaration statements are no longer valid within SELinux policies; there were problems with `require` statements, which are SELinux constructs to import types, roles, and classes used in a module; figuring out correct syntax was a challenge at first, especially for the

`dominance` statements; there were issues with built-in policy macros that didn't compile correctly and produced types and roles that collided with ones that I defined later in the module.

ARBAC is another issue entirely for SELinux. Because of the static nature of SELinux policies, it is impossible to express any role-checking ARBAC needs done to make it effective, such as checking precondition roles for role assignment and revocation. Fig. 2.9 and Fig. 2.10 show an attempt at creating an administrative branch for the previously discussed RBAC policy. The problem is that this code is enforced only by convention. To separate the privileges of the administrators, I have given them each access to separate SELinux policy modules that may only contain their administrative code. However, nothing in SELinux can prevent them from writing whatever policy code they want in those modules. I have given them the ability to read all of the policies in order to enable administrators to check up on each other, but those are the only checks available. `can_assign` and `can_revoke` rules are implicitly defined in the administrators' separate modules, but there is no precondition role checking available; administrators must assign users to roles and design the roles in such a way as to enforce the RBAC policy they imagine. In addition, this construct makes policy very difficult to change, and very static: administrators, to change a user's role, must edit an SELinux policy module, compile it, and install it, and even then, if the affected user has a long-lived shell, the change in access rights will not affect that shell session, creating possible security weaknesses.

[Sha07] defined and implemented a way by which a group of administrators may democratically administer a system. That is, root powers, such as maintaining backups, configuring services, creating and modifying user accounts, etc., are not allowed to be utilized at an administrator's discretion: the whole group must vote to allow the action. This makes sure that no administrator abuses root powers, intentionally or not.

Her idea is a perfect fit for ARBAC — the administrators have certain roles pertaining to what they may administer and, depending on the system, these roles could be arranged into a hierarchy for ease of use. And indeed she did use SELinux's RBAC facilities to create this administration scheme; she used a combination of SELinux policy and shell code (for the elections themselves). When she implemented it, SELinux's `dominate` statements had not yet been fully implemented, which made it much more difficult to create. She made heavy use of its macro facilities, however, and, after a great amount of difficulty, was able to get a working system. If there had been RBAC facilities already in place, in SELinux or otherwise, the development of this project could have gone much more smoothly. If ARBAC facilities has existed, perhaps it would have been even easier.

```
policy_module(rbac_module, 1.0)

###############
#
# The purpose of this file is to create a basic hierarchical RBAC structure,
# part of the example university policy. It contains only students, grads,
# undergrads, TAs, RAs, Graders, and Employees. This is the structure:
#          Grader    RA __   TA
#             |  \    |   \  /|
#             |   \   |    \/ |
#             |    \  |    /\ |
#         Undergrad Employee Grad
#                 \          /
#                  \        /
#                   \      /
#                     Student
# Roles above other roles inherit their juniors' permissions and users assigned
# to senior roles are implicitly members of the junior roles.
#
# The permissions have only to do with the following objects:
# gradebook (file), lab_notebook (file), class_list (file), signup_class_grad
# (process), signup_class_undergrad (process), input_hours (process), and
# class_curriculum (file).
# The permissions on the file objects are only read permissions and append
# permissions. The processes have only execute permissions.
#
# The set of users is {joe burg lisa gwen bob alice sarah bendy}

# Declare the roles and bind them to types
role ra types {ra_t};
role grader types { grader_t };
role ta types { ta_t};
role employee types {employee_t };
role grad types { grad_t };
role undergrad types { ugrad_t};
role student types { student_t };

# Dominance statements to create the hierarchy
dominance { role grader { role undergrad; role employee; } };
dominance { role ra { role grad; role employee; } };
dominance { role ta { role employee; } };
dominance { role grad { role student; } };
dominance { role undergrad { role student; } };

# An alternative to the set of dominance statements above is:
# dominance { role grader { role undergrad { role student}; role employee; } };
# dominance { role ra { role grad {role student}; role employee; } };
# dominance { role ta { role employee; } };
# It demonstrates the fact that dominance statements may be
# nested in SELinux.

# Declare the types; allow rules later.
#type ra_t; #The reason these are commented out is because they
#type grader_t; #collide with types defined later by templates used
#type grad_t;   #to allow all of the users standard permissions.
#type ugrad_t;
#type employee_t;
```

Figure 2.7: Sample role policy, Part 1. It is a hand-implemented version of part of the university policy, using roles and types as roles.

```
# Valid type declarations
type ta_t;
type student_t;
type gradebook;
type lab_notebook;
type class_list;
type curriculum;
type register_exec_t;
type register_t;
type input_hours_t;
type input_hours_exec_t;

# Allow rules for student_t:
allow student_t class_list : file { read_file_perms };
#Allow students to transition to the registration type for class sign-up.
allow student_t register_t : process transition;
type_transition student_t register_exec_t : process register_t;

# Allow rules for employee_t:
allow employee_t input_hours_t : process transition;
type_transition employee_t input_hours_exec_t : process input_hours_t;

# Allow rules for ugrad_t
allow ugrad_t gradebook : file { read_file_perms };

# Allow rules for grad_t
allow grad_t gradebook : file { read_file_perms };

# Allow rules for ra_t
allow ra_t lab_notebook : file { read_file_perms append };

# Allow rules for grader_t
allow grader_t gradebook : file { read_file_perms write append };

# Allow rules for the process types.
# These macros set up standard rules for making an executable
# file related to a process type, called a domain.
domain_type(register_t)
domain_entry_file(register_t, register_exec_t)
domain_type(input_hours_t)
domain_entry_file(input_hours_t, input_hours_exec_t)

# The userdom_unpriv_user_template gives standard permissions,
# trusting that the user has not already defined user types and
# roles. Folly! Also, this is the only template to grant these
# permissions, as far as I can tell.
userdom_unpriv_user_template(ra)
userdom_unpriv_user_template(grader)
userdom_unpriv_user_template(ugrad)
userdom_unpriv_user_template(grad)
userdom_inpriv_user_template(employee)

# Declare the users to SELinux and bind them to roles. If only this
# worked in Reference policy SELinux; as of now there is only a much
# more complex way to define these.
#user joe roles {ra};
#user burg roles {grader};
#user lisa roles {undergrad};
#user gwen roles {undergrad};
#user bob roles {undergrad};
#user alice roles {grad};
#user sarah roles {grad};
#user bendy roles {ra ta};
```

Figure 2.8: Sample role policy, Part 2

```
policy_module(arbac_module, 1.0)

##################################
#
# A policy for an administrative branch of an RBAC policy. Uses the same
# paradigms as regular RBAC.
#

# Define the roles
role president types { president_t };
role dean { dean_t };
role professor { professor_t };
role admission_officer { admission_officer_t };
role asst_proffesor_r { asst_professor_t };

# Define types for the roles
type president_t;
type dean_t;
type professor_t;
type admission_officer_t;
type asst_professor_t;

# Define the hierarchy.
dominance { professor_r {asst_professor_r; } };

# Define types for each administrative file dealing with the other roles.
# This is to enable specific administration; the defined files are the only ones
# in which that particular branch of administration may be found. However,
# there is little enough to enforce that each file contain only those elements.
# This is an imperfect system, and the only way I could think of to create
# this kind of administration.
type employee_t;
type undergrad_policy_t;
type grad_policy_t;
type undergrad_policy_t;
type ra_ta_grader_policy_t;
type admin_policy_t;

# Allow rules for the admin types and the types they can modify.
allow president_t admin_policy_t : file { read_file_perms write_file_perms };
allow admission_officer_t grad_policy_t : file { read_file_perms write_file_perms };
allow admission_officer_t udergrad_policy_t : file {read_file_perms \
      write_file_perms};
allow professor_t ra_ta_grader_policy_t : file { read_file_perms write_file_perms };
allow dean_t employee_t : file { read_file_perms write_file_perms };
# The dean may only revoke ra, ta, and grader roles from users, but that is here
# enforced only by convention.
allow dean_t ra_ta_grader_t : file { read_file_perms write_file_perms };

# All types should be able to read the files. This is to encourage
# administrators to check on each each other due to the loose nature
# of the administration.
# Note that the bracket syntax for the object types in these statements in
# personal shorthand and not valid SELinux policy language.
allow president_t { undergrad_policy_t grad_policy_t employee_t \
      ra_ta_grader_policy_t } : file { read_file_perms };
allow dean_t { admin_policy_t grad_policy_t undergrad_policy_t} : \
      file { read_file_perms };
allow professor_t { admin_policy_t undergrad_policy_t employee_policy_t \
      grad_policy_t } : file { read_file_perms };
allow admission_officer_t {admin_policy_t ra_ta_grader_policy_t \
      employee_policy_t} : file { read_file_perms };
```

Figure 2.9: A simple ARBAC policy in SELinux policy language. Note that this includes only administrative permissions and roles and not those of regular users. This, too, is based on the running University example.

```
# All of the other permissions created related administering the users are
# found in the files labeled <x>_policy_t. This would include user
# declarations and role declarations and assignments, role-type bindings,
# and type-permission bindings. Such things as allow and revoke rules are
# merely understood by convention and not enforced by SELinux, as there is
# no way to do so in only SELinux.

# Because there are specific files we're dealing with, we need a file contexts
# file (FC) to label the files specifically. It would contain:
<filepath to employee admin file> : system_u:system_r:employee_policy_t:s0
<filepath to undergrad admin file> : system_u:system_r:undergrad_policy_t:s0
<filepath to grad admin file> : system_u:system_r:grad_policy_t:s0
<filepath to ra, ta, and grader admin file> : system_u:system_r: \
     ra_ta_grader_policy_t:s0
<filepath to administrative admin file> : system_u:system_r:admin_policy_t:s0
```

Figure 2.10: A simple ARBAC policy in SELinux policy language, part 2: File contexts file (FC file). Because the administrative constraints aren't strictly enforceable, the convention is that each administrator puts administrative policy relevant to their roles in their dedicated policy files.

## 2.7 Stress and Hardship for the SELinux System Administrator

In [MMC07], the authors state that SELinux policy language merely exposes the complexities of Linux and adds no complexity of its own. While this may be true, there is little advantage to the user in being required to work with this complexity. SELinux policy language is roughly comparable to assembly code in both its usability and readability. This is obviously a problem — while it's possible to get used to writing and reading assembly, it is by no measure ideal. It allows the incredibly fine-grained control, but at the expense of usability. Besides, there is no guarantee that a higher-level look at the system will sacrifice fine-grained control.

There are many parts of correctly configuring and administering an enforcing SELinux system. I will, as best I can, enumerate what they are and why they are too difficult to be considered reasonable.

**Custom policy module creation/modification** The standard SELinux configuration does not confine every process. It is only logical that system administrators would want to write their own policies to confine specific processes or users that are unconfined in the standard policy.

The first hardship a system administrator faces is learning the policy language. It is not too difficult; there are resources, though many of them outdated, that very adequately explain what all of the rules are for and where to put them. In the grand SELinux scheme of things, this is one of the easier challenges to overcome. However, it does take considerable time to fully understand the semantics of each rule and how they are enforced and interact with each other.

The second hardship is figuring out all of the interactions that the confined process

needs to be able to work at all, what it claims it needs but doesn't, and how it interacts with currently confined objects on the system. None of these are at all trivial. Even the simplest processes often need access to various library files and devices. When I was writing modules, I realized there was a huge number of interactions going on that I previously had no idea were occurring, such as the library file and device accesses. The system administrator must not only know all of these interactions, but understand them well enough to securely confine the process to accesses it truly needs. If the process interacts with aspects of the system confined by SELinux, largely there are interfaces to the required types that allow generalized accesses. This makes the system administrator's life marginally easier.

**Debugging custom policy modules** Part of the process of developing custom policy modules is debugging them. It is an integral part of the process, not something to be done after a complete module is written but while it is being written. It is to make sure the system administrator catches all of the interactions and to make sure there are no bad rules in the module. In addition, the consequences of SELinux module bugs can be more disastrous than an insecure module; if a buggy module is installed and the system set to enforcement mode, it is possible that no user, not even root, will be able to log into the system. Thus module debugging is made even more important as part of the process of module development. Debugging is done via `setroubleshoot`, a policy utility that parses audit log files into a readable form and gives feedback to help resolve problems. It is a wonderful tool and greatly eases debugging, both by aiding understanding of the policy and by decreasing the total time spent debugging. This does not mean that the time spent debugging is shrunk to such a great extent that debugging becomes reasonable. As an SELinux beginner, I spent hours and hours with `setroubleshoot` and was grateful for its help. I'm sure more experienced policy writers have a much easier time of it, but the learning curve is very steep and very long.

**Debugging standard SELinux modules** It is inevitable in a MAC system that the standard configuration will not work for every system. When this happens, most people turn SELinux off, but for the system administrators that don't, they have to debug the standard policy if they want SELinux enforced. I have not had to do this, but I imagine that it would be extremely arduous, more so than debugging modules system administrators write themselves because they do not already have an inherent understanding of what the modules are supposed to be doing. Unless system administrators are very familiar with what the problem module(s) confine, it could be a very long process of learning that, figuring out how the policy code there expresses that, how it interacts with the rest of the policy to avoid breaking parts that may depend on it, and modifying it in such a way that the desired changes take place and undesired

ones do not. To me this seems like an impossible task, but somehow people do it. There are tools to ease the pain, such as `audit2allow`, which takes a denied access audit and generates a small policy module to allow it. The problem with these tools, though, is that oftentimes they do not *just* allow the desired interaction but a whole bunch that are unexpected. This is based on reading lists and forums where people have complained that this utility gave the problem process far-reaching permissions; it seemed to be a common problem. I would not trust these utilities to keep the system adequately secure.

**Managing the SELinux policy** This includes compiling and installing modules, adding users, checking and changing the status of SELinux, and relabeling the file system. Compiling and installing modules is easy once one knows what to do, but my path to figuring this out was unusually bumpy and documented in the Intricacies and Errata section at the end of this chapter. Adding users, also documented in the Intricacies and Errata section, is not something usually done to SELinux policies because they center on confining processes rather than users. Because of this, it is considerably more difficult than adding other policy components like types, roles, and rules. It took me three months to figure out how to do this on current SELinux versions, which is completely unreasonable. Checking and changing the status of SELinux, which can be either enforcing, permissive, or disabled, is the easiest of the management sections. There is a simple command to check and change the status, but changing status can have unforseen consequences. Disabled mode is related to file system relabeling: if the file system changes in a major way or is otherwise made inconsistent with the file groups and labels specified in the `file_contexts` file, turning SELinux back on, whether in permissive or enforcing mode, will break the system and could bar anyone from logging in due to file system inconsistencies. The way to fix this is to relabel the entire system, which is the most heavyweight operation in SELinux. Depending on the size of the file system and the complexity of the policy, relabeling can take hours. If at all possible relabeling should be avoided (and is usually only required in dire situations like the above, except on a fresh install), but is still a part of SELinux policy administration.

There may be parts of administration that I missed and of course it becomes easier with time, but that is the case with anything. Having a higher-level system on top of SELinux could ease a lot of the especially painful points, such as the module creation process. This is another reason I have designed Tolypeutes: fixing what parts of SELinux I can, because it's a worthwhile system.

## 2.8 SELinux Intricacies and Errata

One of the main difficulties for me in working with SELinux was finding out how to do many seemingly basic things, including compile and install modules, add SELinux users and map them to Linux users, and how to utilize dynamic policy components, amongst others. I imagine it is the same for many other SELinux users. The main barriers I faced are enumerated here.

**Module compilation and installation** Modularity, including administrators' ability to write new modules to confine different aspects of the system, is one of SELinux's central ideas. Despite this, the process of module compilation and installation was not obvious and not well-documented. When I learned that I had to use the SELinux development makefile rather than the `checkmodule` utility (the actual compiler), through searching the Internet rather than Tresys's Reference Policy documentation, I found that the makefile had errors in it that prevented compilation of anything. I successfully edited the makefile to allow compilation, but it was something of a process and not something an administrator should have to deal with — things like module compilation should just work, not be prohibitively difficult.

**Adding unique SELinux users and mapping them to Linux users** There is a reason I include in my implementation a way to add users in batches such that they are successfully tied to unique SELinux contexts: it is because I found this process to be opaque and not well documented. Because SELinux has traditionally been aimed at background daemons rather than users, there exists very little documentation on the process of confining individual users. I encountered what I perceived to be differing accounts of whether it was possible to confine users in this way, and on which version of SELinux, but eventually learned from asking on the SELinux list that it is possible, on any version of SELinux, and how to do it.

**Generating policy configuration files** This is a separate problem of policy compilation, but one I found to be very frustrating and never solved. It is supposed to be possible to generate configuration files, as a target of the policy makefile, that govern which modules are included in the policy and which are not, without using `semodule`, the module installer and modifier, to remove modules. It also lists whether each module is part of what is called the base module, the module that makes up the core of SELinux's system confinement. This would have given me valuable information about the makeup and structure of my SELinux install, and allowed me to more easily manage which modules I wanted enforced on my system (the better to test my modules with). However, I found that my makefile did not have the `conf` target that would allow it to generate such a file. When I searched about this and inquired on the list, I found no answers other than that it should be there. No matter the install I used,

Fedora 7, 8, or 10, my makefile did not have that target.

There were other problems I faced while working on this project, but these were the main ones that barred my progress for the longest time.

## 2.9 System Design Principles and SELinux

SELinux is very complex, but is it complex without gaining any advantages? I will evaluate it along eight well-known dimensions ([LM07], [SS75], [SM08]) that are especially relevant to security. Doing so exposes SELinux's particular strengths, but also makes it very obvious where SELinux is weak and needs to be fixed.

**Scalability and flexibility** For a security mechanism to be scalable means that it is just as easy, or only marginally more difficult, to use it on large systems as it is to use it on small systems. For one to be flexible means that it must not be too difficult to change. SELinux is not scalable or flexible, but it has the beginnings of those capabilities. It *can* be modified and rewritten, but that process is extremely complex and can require deep knowledge of the system. It can scale to larger systems because it already implements the lower-level and core network functionality (the Reference and Targeted policies), but even then writing policy for more applications, users, etc. can require deep knowledge of of other parts of the policy.

**Psychological acceptability** [SM08] put the importance of psychological acceptability succinctly: "If users can't understand the system, they won't use it correctly." In the case of SELinux, using it incorrectly is usually turning it off, which is a course of action often taken by users frustrated with the difficulty of configuring and using SELinux. Obviously having SELinux turned off could be a serious security flaw, as regular Linux is susceptible to privilege escalation and all sorts of other attacks. This demonstrates the importance of psychological acceptability, a principle SELinux simply does not adhere to.

**Fail-safe defaults and least privilege** I lump these two principles together here because in SELinux, they are very much related. SELinux was designed to implement least-privilege as closely as possible. This led to its default action — deny any access unless specifically allowed, which is the idea of fail-safe defaults. However, sometimes its fail-safe defaults are so fail-safe that *no one* can use the system, intruder or lawful user, so SELinux both fails and succeeds to adhere to this principle.

**Economy of mechanism** SELinux follows economy of mechanism in much the same way as assembly language programming does: there is a small mechanism (that is, few rules types), but specifying anything in it is very lengthy and complex. Like fail-safe defaults and least privilege, SELinux both fails and succeeds at this.

**Complete mediation** Every access request to any object by any process or user must be checked, even if access was granted previously. This is not necessarily the case in SELinux — the AVC can give decisions without consulting the security server, a potential security flaw if the policy can be dynamically changed in a serious way. It may not be the case that the policy can be changed at run-time in such a way as to invalidate the AVC, so this is potentially not a problem at all like it would be in regular Linux DAC.

**Open design** This is something SELinux does very well; it is an open source project and has an active, if small, development community. Security through obscurity is rarely if ever a good idea.

**Separation of privileges** One person should not have absolute power over a whole system. In standard Linux it is not possible to have separation of privileges because of the `root` account, from which permissions cannot be withheld. In SELinux, it is only convention that can prevent the existence of a superuser; the person or people with access to the policy have access to the whole system because of their ability to rewrite the policy. However, when there are multiple administrators on a system, access to security modules may be restricted so that an administrator has access to only half the policy or certain parts. This does not negate the problem, but helps build in a system of checks and balances among administrators that would not be possible with only the `root` account. So in this, SELinux does as good a job as it can, but the necessity of policy modification makes it impossible to remove the possibility of a superuser.

Out of the eight design principles discussed, SELinux fails to adhere, in part or in whole, to four of them: economy of mechanism, scalability and flexibility, psychological acceptability, and fail-safe defaults. Psychological acceptability is especially important here because SELinux fails so utterly at it, in part because it fails to adhere to the other two. Because SELinux mostly succeeds at the other four, the system I have designed uses its strengths and attempts to overcome those usability issues that have plagued it since its inception. My system is described in Chapter 4.

An important addition to the discussion of SELinux is whether it would be possible for ARBAC to express any security idea and fine-grained confinement expressible in SELinux. From [OS00], we know that ARBAC is flexible enough to express at least lattice based MAC policies, which are similar in more straightforward ways to RBAC. However, SELinux policies are not generally lattice based and TE is significantly lower-level. Despite this, I think that it is possible to express most all aspects of SELinux TE in ARBAC. As previously stated ARBAC is not intrinsically limited to confining users, especially when users are just glorified processes. I may not know all of what SELinux is able to express, but I am nonetheless confident that ARBAC is able to conceptually replace SELinux TE. A

discussion of this issue in relation to Tolypeutes is presented in Chapter 5, future work, and the correlation between Tolypeutes ARBAC and SELinux TE is presented in Chapter 4.

# Chapter 3

# Related Work

Linux, being open-source and relatively popular, is a great platform for developing new security solutions. But security is a very difficult problem and there are relatively few popular, supported, and widely available security packages outside of SELinux. The relatively few attempts to remedy SELinux's near-unusability, as well as the biggest non-SELinux security packages, are listed here.

## 3.1 Policy Generation and Analysis Tools

- `semanage` is a program a system administrator may run on the command line to affect the policy. It allows administrators to add, modify, delete, and list users, roles, and a variety of types in the current SELinux system. This is the simplest way to add and remove users and roles from an SELinux system, but it provides no more advanced functionality in that realm. It affects only what may be changed without recompilation and reloading of the policy. However, it does point to some interesting possibilities in the realm of dynamic policy modification.

- `setools` is a suite of troubleshooting and analysis tools by Tresys Technologies, the same company that created the Reference Policy version of SELinux. As such, the setools suite comes with some very nice tools, chiefly `apol`, a graphical policy analysis tool. It provides statistics about a policy and can perform a few types of analysis, including information flow, type relationships, and domain transitions. The setools suite also includes an SELinux log message analyzer, designed to make troubleshooting a policy much easier, and a difference engine specifically for SELinux policies. Altogether it is a very useful suite, though I have had trouble getting it to recognize modular policies as such. Nonetheless, it does not ease the burden of policy modification or generation.

- The Policy Generation Druid is a program that comes with the Fedora distribution of SELinux. It is, as the name suggests, for generating SELinux policy modules quickly,

easily, and with very little input from the user. It can only generate application and user policies, each adhering to set standards. This makes it unreasonable for creating very specific policies and it has no way to generate RBAC modules.

- SEEdit [NS08] is both a graphical policy generation tool and a higher-level language, working off of what is called the Simplified Policy. It was created by Hitachi Software to simplify the policy editing and creation process, though mostly the policy creation process. It was developed for consumer electronics (that is, small embedded computers) and so is heavily reduced in both size and scope; it lacks many of the features of regular SELinux, thus making it probably unusable for both regular computer systems and most certainly for servers. However, it does have good features, such as hiding type transitions and collapsing very specific permissions into larger groups. It does include RBAC, but it remains essentially unmodified from its state in SELinux.

## 3.2 Higher-Level Languages On Top of SELinux

- PLEASE [Qui07] is a higher-level language built on top of SELinux, created for a master's thesis. It does not include RBAC policy creation capabilities (it is in fact noted in the author's 'future work' section as something else to be done). Its main goals are to be easy to understand and to be modular. It is object-oriented and focuses on SELinux's TE.

- Lobster and Shrimp [Whi08] are higher-level languages on top of the Reference Policy. Shrimp exists as an actual implementation (though not circulated for general use) and implements a sort of type (in the programming languages sense) system for SELinux. The author also wishes to include an information flow constraint for Shrimp. Lobster, currently not implemented, is on top of Shrimp and provides further abstractions; it takes the object-oriented approach, much like PLEASE. RBAC is never mentioned in the source I have for Lobster and Shrimp.

What all of these tools and languages have in common is that they do not fully support RBAC. I find this problematic because RBAC is much easier to understand, use, and create than TE policies (though TE is wonderful and I appreciate its fine-grained beauty). In addition, none of them are in widespread usage. This leads the topic of my thesis: creating a higher-level language on top of SELinux that supports ARBAC policy creation and analysis.

## 3.3 Non-SELinux Security Enhancements

- Novell AppArmor is another way to implement a MAC policy on various Linux systems (SUSE Linux, Ubuntu, and Mandriva) [Bau06]. It uses the Linux Security Module

just like SELinux does, but is more limited in scope than SELinux. I chose not to use AppArmor because it is not available on my Fedora test system and because I need the user confinement capabilities of SELinux.

- RSBAC is a security framework formerly shipped with Mandriva Linux but available for a variety of Linux flavors. It is not a specific access control model, but provides implementations for MAC (in addition to other models, but not including ARBAC). RSBAC does not extend the attributes of everything on the system as SELinux does, but instead keeps extended attributes in special directories and modifies relevant system calls to consult these stored attributes [Ott07].

  I do not find this model as secure as SELinux because it depends on what is essentially hacked system calls and supposedly inaccessible directories of sensitive security data. Having all of the security stored, checked, and enforced via the kernel is a reassuring aspect of SELinux.

# Chapter 4

# Tolypeutes: ARBAC on Top of SELinux

In this section I describe the design and partial implementation of Tolypeutes, but first I would like to emphasize that SELinux is a very powerful security system, which is why I have chosen it for the base of the Tolypeutes ARBAC system; its effectiveness against privilege escalation is particularly alluring. It provides all of the hardest things to implement, making it so that the hardest thing I had to worry about when designing Tolypeutes was generating well-defined, reasonable SELinux code. The complexities involved with that task are certainly not trivial, but it allowed me to concentrate more on my main goals: building a system that is as close to the abstract ARBAC model as possible and making it as usable as I could. Merely following the ARBAC model leads to adhering to the design principles that SELinux misses: flexibility and scalability, economy of mechanism, and psychological acceptability.

Currently the Tolypeutes partial implementation only supports user confinement rather than more general process confinement. I made this decision to help me finish as much as I could in a short amount of time, but the extension to processes should be simple and is discussed in Chapter 5.

An additional goal for the Tolypeutes system is reducing module (or entire policy) compilation and installation and file system relabeling on an active SELinux-enabled system where a lot of administration takes place. Module compilation and installation is not an especially expensive operation, but should be avoided on a system concentrating on user confinement for reasons discussed later in this chapter. File system relabeling, however, can be an extremely expensive and lengthy operation and should be avoided if at all possible. If humans are not interfacing as much directly with SELinux, compilation and installation becomes less obvious and relabeling, which is usually prompted by human mistakes, will hopefully be even rarer.

## 4.1 Tolypeutes Features

The main feature of Tolypeutes is the ability to enforce ARBAC rules, such as SMER constraint and preconditions in assign and revocation rule checks, which, as previously stated, SELinux is unable to do. This is the core of the administrative aspect of ARBAC. Dynamic administration is another main feature of Tolypeutes, though it is not possible to modify the more integral policy components such as hierarchy and permission-role association without recompiling and reloading the underlying SELinux policy. It is expected that an ARBAC policy is in constant flux, especially at larger sizes, so it is important for administrators to be able to easily and quickly make the changes they need. For security, the dynamic administration capabilities are fortuitous: the same mechanism that allows for dynamic administration forces user changes in permissions to be active immediately, neutralizing some security weaknesses otherwise present.

Other language features include permission and policy shrinkage. By permission shrinkage I mean that what would normally be many convoluted `allow` rules in SELinux could become single permission assignment statements in Tolypeutes. These include built-in permission constructs such as file reading and program execution, but also more abstracted permissions that collect many low-level SELinux permissions together. It is possible for administrators to name collections of permissions on a set of objects that they have defined, making it more customizable. This functionality is similar to SELinux macros, except that the permissions may be tied to the set of objects; in SELinux this is not possible. Policy shrinkage occurs only in terms of the policy that an administrator writes: the Tolypeutes policy will be much, much smaller than the SELinux policy it produces. Currently it is unknown by what factor exactly is the policy smaller, though I believe it is significant.

## 4.2 Tolypeutes Architecture

The Tolypeutes system has six separate, self-contained components:

- The Tolypeutes ARBAC policy specification language,
- the language transformer,
- the SELinux code generator,
- the configuration file generator,
- the administrative constraints enforcer,
- the user batch-adder.

I have chosen to decompose the system into these components because it makes them easier to reason about, both for me and hopefully for any administrator using the system.

It is especially useful for me because as long as the files produced by each module remain constant, I can change individual parts of the Tolypeutes system without breaking it, a good feature for development.

All of these components work together to make it possible to have an enforced ARBAC policy. ARBAC policies are specified in the Tolypeutes language. The transformer reads the human-generated policy and generates more computer-readable, but significantly less human-readable, policy files that the SELinux code generator and configuration file generator use. The SELinux code generator generates SELinux policy modules corresponding to the ARBAC policy, though only RBAC rules are expressible in SELinux. The configuration file generator creates role hierarchy- and assignment/revocation prerequisite-specific configuration files that the administrative constraint enforcer utilizes to make sure administrators don't move outside their assignment/revocation constraints and that role assignment properly obeys the hierarchy. The user batch-adder is a tool to add users with unique SELinux security contexts, which is a prerequisite for the Tolypeutes system to work. A graphical representation of this structure is presented in Fig. 4.1.

Each component will be discussed in depth below. I will explain why each part is necessary and exactly what it accomplishes, both in terms of security achieved and security design principles. To better illustrate all of these things, the running University example will be augmented with a specific application: Beauregard's journey through the University and what system administrators do to effectively administer his account.

## 4.3 Tolypeutes Language Design

The language in which administrators specify ARBAC policies is the most important feature of the user interface, as it represents both the most complex part of the system administrators use and the one with which they're likely to spend the most time. Therefore, the principles of abstraction, small core language, psychological acceptability, and modularity are the most important design principles to consider for the language.

**Abstraction** SELinux exposes far too many details of the both the operating system and the file system, which is one reason why it's so difficult to use. It requires administrators to be far more familiar with every interaction a constrained process has with other system objects than many are, or would like to be, for basic security. The Tolypeutes language abstracts over as much of the system as possible, though it is still possible to specify SELinux-level permissions. Generally speaking, though, ARBAC permission assignment is far simpler. Permissions such as 'use program' and 'read and write a file' are atomic permissions. This eliminates the need to specify four-part type transitions and permissions like `getattr`, `setattr`, `lock`, and `ioctl` for otherwise intuitive 'read' accesses. Permission specification is certainly the most complex part of the language, though easier than in SELinux, because roles, users, user assignment, and

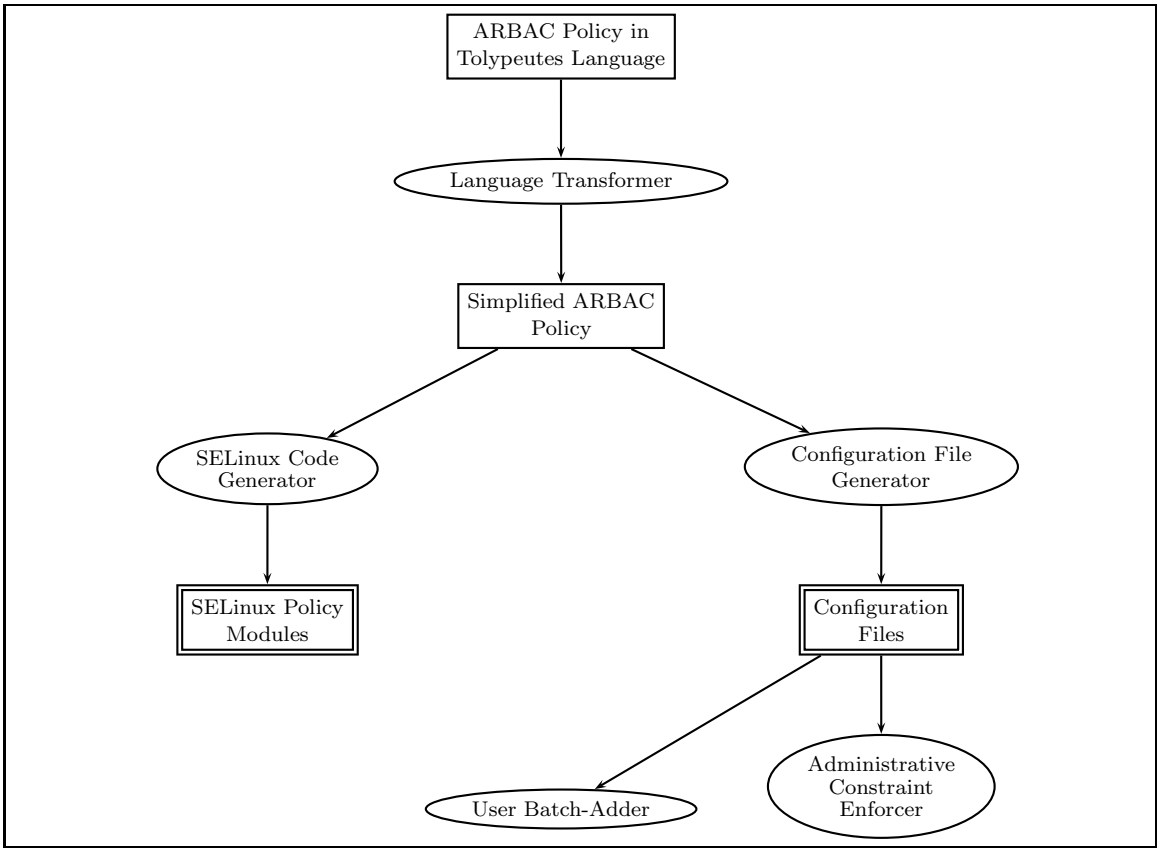


Figure 4.1: The architecture of the Tolypeutes system, which is made of five discrete modules and well-defined file interactions. The ovals are Tolypeutes modules and the boxes are files they produce.

administrative rules are so very simple and abstracted. It is very difficult to abstract access definition away from specific files and permissions on them.

**Small Core Language** In order to allow the most flexible and easily specified ARBAC possible, Tolypeutes has a very small core language, made of a small number of rules. With a smaller core language to learn and understand, policy writers should be able to start writing and modifying policies sooner than with normal SELinux and with greater ease. The only model an ARBAC policy writer should be thinking about is that of their desired security structure, not the language model or the underlying architecture of the file system (except where necessary). Though it is not currently possible, it would be greatly desirable to allow administrators to write their own abstractions — functions for ARBAC. This language extension is discussed in the Future Work section.

**Psychological Acceptability** In a way this is the most important part. The reason I believe Linux needs another security system is that the ones it currently has are too difficult to use. The Tolypeutes language is designed for people with general programming knowledge and some system administration experience with a good idea of what they want the policy to enforce. After familiarizing themselves with the basics of ARBAC, they should be able to start specifying syntactically correct, if not necessarily semantically correct, policies. The semantics are also relatively simple (certainly simpler than SELinux and, in some ways, Linux DAC), though it is possible to have unexpected role interactions in even the simplest hierarchies.

**Modularity** ARBAC is ideally suited to large, user-heavy systems, constantly in flux. It is certainly still difficult to create a policy for and manage a many hundred- or thousand-role and user system, but ARBAC provides possibly the simplest and most flexible way to do so. RBAC to administer RBAC makes it simpler to add administrative blocks to a policy, separate trees in a forest-structured policy. The Tolypeutes language reflects this by supporting a modular design. Many different policy files, specifying different part of a single policy, may be declared to be part of the same larger policy, thus allowing them to share data. If different policy files are not declared as part of the same policy, they may share no data and are considered entirely separate policies for the purposes of enforcement.

### 4.3.1 Syntax

The language has only seven main rules, with ten total built-in statements. The policy rules are separated into two kinds: regular RBAC rules and administrative rules, following the ARBAC model. The RBAC rules are as follows:

**Role hierarchy rules** The method by which the hierarchy is constructed. These have the following structure:

```
hierarchy (<senior role>|(<senior role list>), <junior role>|
  (<junior role list>)) |
admin_hierarchy(<senior admin role>|(<senior admin role list>),
  <junior admin role>|(<junior admin role list>))
```

Either or both of the senior and junior role fields may contain a list of roles. All roles in lists are directly senior or junior to all roles in the other list, with predictable consequences for singular fields.

**SMER constraints** Here they are called mutuallyExclusive rules, with the following structure:

```
mutuallyExclusive (<role1>, <role2>)
```

List fields are not valid in SMER constraints. This design choice was made to help ensure the policy writer truly understands the policy. SMER constraints of only two fields become complex enough; I consider it unwise to enable people to make them even more complicated.

**User assignment rules** These rules assign users to roles and come in both a regular user flavor and an administrative user flavor:

```
userAssign (<user>|(<user list>),<role>|(<role list>)) |
admin_userAssign (<admin user>|(<admin user list>), <admin role>|
  (<admin role list>))
```

In the case of lists, all users in the user list are assigned to all roles in the role list.

**Permission assignment rules** Assign permissions to roles. These are the most flexible rules in the Tolypeutes language, as the objects on which permissions are given can be specified in many ways.

```
permissionAssign (<role>|(<role list>),(<permission identifier>)|
  (<permission identifier list>)|(<object regexp>,(<access list>)))
```

Permission identifiers can be either user-defined or one of the few built-in, predefined permissions such as `login`. User-defined permissions within permissionAssign rules are the `(<object regexp>,(<access list>))` form. Object regular expressions should identify a group of objects on the file system and the access list can

include any access available in SELinux or the predefined, more abstracted ones such as `use` for executables and `read, write`, and `append` for files amongst others. Built-in permissions are those that are essential for a functioning system and/or are easily available from SELinux and/or are difficult to define from scratch.

**Declaration rules: users, roles, resources, and permissions** These rules declare to Tolypeutes what basic units it will be dealing with. All users to be managed in the system must be declared, a weakness discussed later, as must all roles and all objects to which roles have access. However, none of these things need to be declared in one of the special declaration rules. Users may be identified when they are assigned to roles, roles may be defined when they are put in hierarchy or when permissions and users are assigned to them (this means that typos could cause major problems, however), and permissions may be defined in permission assignment rules. All of them have very similar syntax and define identifiers usable throughout the namespace scope.

```
user (<identifier>) | users (<identifier list>)
admin_user (<identifier>) | admin_users (<identifier list>)
role (<identifier>) | roles (<identifier list>)
admin_role (<identifier>) | admin_roles (<identifier list>)
resource (<identifier>,<regexp defining a set of files>) |
resources (<identifier/regexp tuple list>)
permission (<identifier>,<resource identifier or regexp>,(<access list>)) |
permissions ((<identifier>,<resource>|<regexp>,(<access list>)) list)
```

As it stands, these are the extent to which administrators may define their own abstractions.

**Administrative user assignment rules** These define which administrators may assign users to which roles and what the preconditions for assignment are. Preconditions are not conditions the administrator must meet, but that the user that is to be assigned must meet: they are roles the user either must have, in the case of a positive precondition role, or must not have, in the case of a negative precondition role. There is no limit to the number of preconditions for an assignment rule.

```
can_assign (<admin role>|(<admin role list>),
  (<user precondition role list), <target role>|(<target role list>))
```

Having a list of administrative roles for an assignment rule means that all listed administrative roles may assign to the target role(s) as long as the user has the preconditions. A list of target roles means that the administrators may assign users to any of the target roles as long as the users have the preconditions.

**Administrative user revocation rules** Revocation rules have the exact same syntax as assignment rules (except are named `can_revoke`), except when the `autorevoke` condition is in effect. When it is, all target roles in all assignment rules are revocable by the administrators specified in the rules. In most ARBAC policies it is true that administrators who can assign may always revoke [SYRG07], so I included this functionality. Under this condition, revocation rules mean that the role *cannot* be revoked by the said administrator at all, even though they may assign to it. Target roles in revocation rules under the `autorevoke` condition need no preconditions. If there is no assignment rule for the administrative role(s) and target role(s), a reversed revocation rule means nothing and is ignored.

Normal role revocation rule syntax:

```
can_revoke(<admin role>|(<admin role list>),
  (<user precondition role list), <target role>|(<target role list>))
```

Autorevoke role revocation rule syntax:

```
!can_revoke(<admin role>|(<admin role list>),
  (<target role>|(<target role list>))
```

A small sample of language syntax, part of the University policy, is presented in Fig. 4.2, Fig. 4.3, and Fig. 4.4. Fig. 4.2 includes the core administrative branch of the policy only, Fig. 4.3 includes the core RBAC part of the policy, and Fig. 4.4 includes regular user declarations and initial assignments.

### 4.3.2 Weaknesses

The Tolypeutes policy specification language does have serious design weaknesses that I must acknowledge. Chief among them is the lack of support for substantial user-defined abstractions. I had originally planned to include extensive language support for these, but time did not permit me to properly design them to best fit ARBAC policy specification. Such abstractions could be as simple as macro equivalents, such as a function that applies a series of predefined permissions to a given hierarchy. A more complex abstraction could be a function that takes a list of users and roles, each of unknown length, and returns a list of `userAssign` rules that relates the users and the roles in some way. It is not impossible to get the functionality of these functions from the bare Tolypeutes language, but it could be much more convenient to allow these abstractions to be created and used.

The second most serious weakness is partially a result of SELinux limitations: all users confined by the policy must be declared, and so known at the outset, in the policy. Nothing stops an administrator from adding users to the policy later, of course, but doing that requires the policy to be recompiled to update the SELinux module(s) and configuration

```
module University
#The above line (the first non-whitespace line in the file)
#declares which policy this file is a part of.

autorevoke
#The above line states that all roles are revokable by the
#administrators who may assign to them.

#Declare roles
admin_roles (President, Dean, Professor, AdmOfficer, AsstProf)

#Declare hierarchy
admin_hierarchy(Professor, AsstProf)

#administration rules for Presidents. The () for preconditions
#means no preconditions.
can_assign(President, (), (Dean, Professor, AdmOfficer))
#Because this is in autorevoke mode, to make roles unrevocable
#requires a little syntactic twist.
can_revoke(President, (), (!Dean, !Professor, !AdmOfficer))

#administration for Dean
can_assign(Dean, (), Grad)
can_revoke(Dean, (), (TA, RA, Grader, Undergrad, Employee))

#administration for Professor
can_assign(Professor, Grad, (RA, TA))
can_assign(Professor, Undergrad, Grader)

#administration for AdmOfficer
#The - in front of grad means that it is a negative precondition role.
can_assign(AdmOfficer, (-Grad), Undergrad)
```

Figure 4.2: A sample of the administrative part of the University policy, in Tolypeutes language syntax. Lines that start with a sharp (#) are comment lines.

```
module University
#All files labeled with the same module name are part of the same module.

#Declare roles and administrative roles
roles (Student, Employee, Grad, Undergrad, TA, RA, Grader)

#Declare hierarchy. Remember: Senior first!
hierarchy((Grad, Undergrad), Student)
hierarchy((TA, RA, Grader), Employee)
hierarchy((TA, RA), Grad)
hierarchy(Grader, Undergrad)

#Declare the exclusive roles. These could also go in
#the administrative section.
mutuallyExclusive(TA, RA)
mutuallyExclusive(Grad, Undergrad)

#Assign permissions to roles. Note ALL, the root of all hierarchies.
permissionAssign(ALL, (login))
permissionAssign(Undergrad, (/usr/bin/drop, (use)))
permissionAssign(Student, (/usr/share/course_catalog, (read)))
permissionAssign(Employee, (/usr/bin/hours, (use)))
permissionAssign((TA, Grader), (/home/cs*/drop/*, (read)))
permissionAssign(Grader, gradebook))
permissionAssign(RA, (/usr/share/labbook, (read, write)))
permissionAssign(Undergrad, (/usr/share/undergradblog, (read, append)))
permissionAssign(Grad, (/usr/share/gradblog, (read, append)))
```

Figure 4.3: A sample of the non-administrative ARBAC University policy. As in Fig. 4.2, lines that start with a sharp are comments.

```
module university
#Connect it to the larger University policy.

#Declare users
users(Beauregard, Lisa, Bob, Leonard, Gwen)

#Assign users to roles.
userAssign(Undergrad, (Beauregard, Bob, Leonard))
userAssign(TA, Lisa)
userAssign(Grad, Gwen)
```

Figure 4.4: For organizational purposes specific to the University policy, all users and user assignments are declared in separate files. An advantage of this organizational style is that the core, general part of the policy is all found in one place and the file only has to be modified to change this core part of the policy. Users may need to be added all the time, so it is logical to place them in their own files. This is an example of what such a file would include.

files, and the SELinux module(s) must be recompiled and reloaded. This is a partial result of using SELinux as a base because in SELinux, all users, types, roles, booleans, etc. must be unambiguously declared in some module, as discussed in the Background section. Therefore, Tolypeutes must require all such data in advance if it is expected to generate correct SELinux code.

### 4.3.3 Pre-Beauregard: The Administrators create a policy

When creating an ARBAC policy using Tolypeutes, the first thing administrators have to do is specify the policy in the Tolypeutes language. In this case, the administrators have created the University policy partially presented in Fig. 4.2 and Fig. 4.3. Beauregard is an entering first-year in the undergraduate class, so he is listed in a file as a user and is initially assigned to only the Undergrad role within the policy, like in Fig. 4.4.

## 4.4 SELinux Policy Code Generator

The Tolypeutes compiler, if it were complete, would generate a generous amount of SELinux policy language code to implement and enforce non-administrative security. There were a great number of design decisions I had to make in terms of how ARBAC components will be translated into SELinux constructs and how they will be organized. My decisions were based mostly on the principles of flexibility and scalability and least privilege and fail-safe defaults, which will be discussed as they come up.

The most important thing to emphasize first is that SELinux policy is broken up between three files per module: a TE (type enforcement) file, an IF (interface) file, and a FC (file context) file. All of these serve radically different purposes and the compiler generates all three for every ARBAC policy created. That is, the compiler creates one module for each ARBAC policy and utilizes all three possible SELinux module file types. I will start with

the simplest: the FC file.

File context files, FC files, bind security contexts to files using regular expressions. This way the administrator is able to specify specific labels for important files, such as process entrypoint files, rather than relying on the system to autolabel everything correctly. If the policy writer names a resource in Tolypeutes, the compiler will include a line in the FC file that correctly labels the group of files if they haven't already been correctly labeled. An obvious difficulty arises from this process: a file may only have one security context at a time, so no two modules may name the same file and assign it to different contexts; it creates a conflict in which SELinux does not know which label to pick and so the module will not be successfully installed if it has conflicting information in its FC file. Currently I have no built-in solution to this problem. Ideally, however, the code generator would parse the SELinux `file_contexts` file, which is the conglomeration of all module FC files, and ensure that there are no conflicts and if there are, warn the user and abort compilation.

Type enforcement files, or TE files, are used to define static policy that is specific to certain programs or file system objects that may neither be changed nor accessed by other policy. The Tolypeutes compiler uses TE files to define pre-known types for an ARBAC policy, such as those used in special file contexts. It also uses TE files to call macros it defines in IF files.

Interface files, or IF files, are SELinux's attempt at modularity and abstraction. Interfaces define macros that other modules may use that generate a set of policy code that may depend on provided parameters. For example, a policy interface macro may take a type as a parameter and use it in a series of TE rules, such as to allow it access to the `passwd` program. A macro may take arbitrary parameters of any form, and this means that it may also take things like user names from which to base policy roles, types, and booleans. This is the basis of Tolypeutes's SELinux security.

These three file types represent SELinux's best effort at modularity and are sufficient to modularize the SELinux code generated for an ARBAC policy. The code generator uses interfaces to define roles that may be called on any number of users defined in type enforcement files, making the addition and removal of users very easy. Types may be defined in TE files for resources defined in the ARBAC language and used in the FC file to tie them to the objects they represent (if applicable). These types may then be utilized in AVC rules in the interfaces; this is not as modular, but the objects to which the types refer are easily changed in the FC file and the AVC rules can be modified without affecting the role structure.

In terms of Beauregard, this step is not (or at least should not) be very interesting — an administrator calls the transformer and code generator on the policy, which generates required SELinux policy modules and configuration files (discussed later). The administrator must then use the little install/boolean initialization script it generated to install the modules and set relevant user-role booleans to initial values defined in the policy.

### 4.4.1 SELinux Booleans: What and Why

SELinux policies are for the most part static once they have been loaded into the security server. That is, without reloading the policy or loading another module, the security cannot be changed. For a dynamic system, however, it is useful to have dynamic administrative capabilities that allow a change of policy and enforcement immediately. This is what SELinux policy booleans achieve, to an extent. Though they have static initial values, their values may be changed outside of the policy to effect immediate change in policy. The initial value for all booleans in the generated code is "false", meaning that no user has any roles. The code generator also generates a small script file to initialize the users to the specified roles, but the defaults are fail-safe: rather than defaulting to true, and thus giving all users every role, the generator starts them off unprivileged.

Booleans are allowed to control chunks of TE rules, limited to `allow`, `neverallow`, `dontaudit`, and `auditallow` rules. No types, roles, or attributes may be defined, types may not be associated with roles or attributes, role hierarchies may not be established, and boolean statements may not be nested. So, while they do provide the needed element of dynamism, they are extremely limited.

What does this mean for policy enforcement? `allow` rules wrapped in boolean statements may be either active or inactive. When a boolean's value changes, rule enforcement on the whole system changes immediately. The consequence of this is that a user who has just had privileges revoked does not need to restart a shell or log off and back in to experience the changes—the user will be immediately barred from accessing the restricted objects. This works in reverse as well; users previously barred from access can access objects upon boolean change. This is why I have chosen to use SELinux booleans to represent the abstract idea of ARBAC roles, both for administrators and regular users. Part of the idea of ARBAC is dynamic administration, the ability of administrators to assign and revoke users to and from roles at will and have the results be immediate.

There are many other alternative role representations I could have chosen. I shall enumerate them and explain why they are not able to achieve the results I want.

**Roles as SELinux roles** User security context: `(unique_user:Undergrad:undergrad_t:s0)`

The most obvious choice of role mapping to SELinux is to use SELinux's built-in RBAC utilities. Using SELinux roles would allow easy translation to a hierarchy SELinux implicitly enforces and supports. However, nothing about roles is at all dynamic and everything would still have to go through types anyway since all security enforcement in SELinux is based on types, not roles. Role allow rules are more limited than type allow rules. Because roles are used so rarely, parts of SELinux RBAC have already been eliminated: role transition rules are deprecated [Sma02].

The main difficulty for administration is that when an administrator wants to change a user's roles, they need to recompile and reload the resultant SELinux policy. Ad-

ditionally, if the user is currently logged in in an old role, they keep it until they log out, which is a potential security risk.

The main difficulty for users is changing roles. Even if a user is authorized for multiple roles, they will only have access of one role at a time. Users would need to explicitly change roles, using the `newrole` SELinux utility, when they need to do something in another role. It is not possible for a background process to change the user's current role without user intervention because processes may not change each other's security contexts. If this were a possibility, this option would be more viable, though still not ideal. Because something like this is not possible, there would be an unacceptable burden on the user; security should be invisible to them unless they attempt a disallowed access.

**Roles as SELinux types** User security context: (unique_user:unique_user_role:Undergrad:s0)

The next most obvious choice is to represent roles as SELinux types. This has the same problems as using roles, but without the advantages of built-in dominance capabilities. However, using types would still be easy to translate and (relatively) simple to represent in SELinux, as they are the central representation of identity.

**Roles as SELinux attributes** This option is noted only because attributes are another possible choice of representation; because they are treated exactly the same way as types, they have the same drawbacks and advantages.

While booleans do not have a built-in hierarchy mechanism and are a more complex, less obvious choice for role translation, I believe they offer the most in terms of ease of administration and user transparency. Because a large part of the point of getting ARBAC for SELinux is to have a more usable security system, using a construct to enhance usability is a great priority. An ARBAC system should not be difficult to use when its core ideas are all about simplicity.

The limitations of using booleans, including disallowed nesting of boolean statements, disallowing type declarations and type/role/attribute association in boolean statements, are large and certainly unfortunate, though understandable. What would it mean for a type to be conditionally defined? What if an object had a conditionally defined type, and the condition changed? How would the file system deal with a labeling difficulty like that? The answer is, it doesn't, and that is why I suspect things like roles, attributes, and types may not be present in boolean statements. Nested boolean statements, on the other hand, should be perfectly possible, but are just not implemented.

An example of the difficulties presented by this situation is that of the `drop` program. Students may `drop` folders and files, which each must have student-specific labels, and not be able to access each other's files in any way. TAs, Graders, and Professors, on the other hand, must have limited access to all of these files, and to all of the student-specific custom `drop`

types. The simplest way to do this would be to have an attribute, `csstaff`, that has the appropriate permissions to the per-user `drop` types. However, because attributes cannot be associated within boolean statements, it is impossible to use this construct without significant work on the part of the SELinux code generator to figure out which users are TAs, Graders, and Professors. Even then, administering these users becomes significantly more difficult because their permissions cannot be controlled by boolean (de)activation. Therefore, the `drop` program's access control and administration becomes a very difficult problem, and one that I haven't solved.

The other aspect of booleans that I must explain is the reason to use per-user rather than per-role booleans. It would certainly be simpler, and indeed create a much smaller policy, to use per-role booleans that govern a great number of users. The problem with per-role booleans is that they do not provide the fine-grained control in administration that per-user booleans do. In order to assign a user to a per-role boolean, they need to be added in the SELinux policy (that is, during compilation). In this case, turning a role boolean on and off does not have the effect of administering individual users, but of shutting down parts of the role hierarchy. Assigning and revoking specific users to and from roles would again require recompiling and reloading policy modules, and would again not have the desired immediate effects.

### 4.4.2 Types, Roles, and Users

Types that are not connected with users, but with objects, are for the most part defined in TE files and used in FC and interface files because they are not defined on a user-by-user basis. It is also impossible to define a context when part of the context is unknown, as FC files do not take parameters, and file contexts cannot be specified outside of an FC file. It is possible to specify how labeling decisions are made when new objects are created, but for objects that have static existences on the system, special file contexts must be specified in FC files.

For every user in an ARBAC policy, a unique SELinux user, role, and type must be defined. The SELinux users are generated outside of policy language code, so I will discuss them later.

User types and roles are partially defined in the administrative macros in the module's interface file and partially in the TE file. The interfaces take a single parameter: a prefix (such as 'Beauregard') and within them use the prefix to define a type and role. The role is defined purely to allow a valid security context for the user to be formed; it performs no other function. However, because of problems with valid contexts, all types to which a user may transition, for any role, must be associated with the user's unique role. Type definitions that are not input-dependent go into the module TE file. Types that are input-dependent, such as `$1_grader_t`, are defined in the administration macro, but obviously not in the boolean statements themselves. These interfaces are invoked on every user defined

in the policy in TE files.

Tolypeutes enforces separate administration, so for each module there are two interfaces: the regular user interface, which includes regular user roles, and the administrative user interface, which includes administrative user roles. Administrators may not be assigned to any regular user roles and vice versa, which means that permissions common to administrative and regular roles, if they are not for ALL, must be assigned to both the administrative and regular roles to which they are relevant. The reason there is an administrative interface at all is because administrative users need many regular permissions, not just administrative ones. This way, administrators may have mundane permissions without mixing with regular users. Separate administration has its drawbacks, but it is a simpler overall model and makes policies easier to analyze. Policy analysis is further discussed in Chapter 5.

Because the administrative macros are invoked upon every user for which the ARBAC policy is defined, access to system resources is controlled on a user-by-user basis, organized by boolean statements (roles). For the special abstract user ALL, permissions are put outside of boolean statements in both interfaces, meaning that all users, regardless of role, may have those permissions. This means that the size of the generated policy is quite large, especially for a large number of users organized in a large and complex role hierarchy.

#### Generated Policy Size

Though I have not had sufficient time to analyze the size of generated SELinux policies, it is a very relevant concern. Initially one may think that the size of the policy is of great importance, that the larger the policy, the slower it will make the system when performing lookups. However, I do not believe this is the case, based both on the SELinux architecture and on the size of shipped policies that are admittedly narrow.

The SELinux architecture includes a cache, the AVC, which stores decisions, and even if it didn't, performing a lookup on what is a (even very large) table is still a most likely constant-time operation. If the size of the policy is entirely unreasonable and the number of rules exceeds the size allotted for storing the policy in the kernel, then lookup could become slow. The reason I doubt even a large ARBAC policy could cause this is because the Targeted SELinux policy has, on my system, more than 288,000 rules. I have not found any documentation that leads me to believe that anyone has encountered problems with large policies causing slow downs before (except where resources are extremely limited [NS08]). Therefore, while I do wish to perform analyses on generated policy size vs. standard policy size, I do not believe it will generally cause problems.

### 4.4.3 Interaction with Standard SELinux Policy

Because Tolypeutes is not expected to be used on a system with a null SELinux policy, I expect that there would be interactions between generated ARBAC SELinux code and the standard policy. Because I have not had the chance the test my system, I am not entirely

sure what all of those interactions are. However, I can speculate what interactions, positive and negative, there will be based on how different SELinux modules interact with each other.

The most important thing that needs to be avoided is trying to label one file as more than one thing within file context files; conflicts produce errors in SELinux. All files have labels, but most of them are default labels inherited from their containing directory. Specifying a certain label for a file or set of files will override the default labeling, but there cannot be two such labeling specifications for a single file. The `file_contexts` conglomerate file is a good place to look for special labels, but is a sub par solution when Tolypeutes seeks to hide as many details of SELinux as possible. It is possible to change the labels on objects manually that contradict specifications in file context files, but that is both tedious and not a good idea; it brings the system closer to an inconsistent state and too much of that could lead to a required system relabeling.

Similar to file contexts but easier to avoid is duplicate types, roles, booleans, and users. Types can only be declared once, ever, in any module, though roles can be associated with types in any module and the syntax is identical. Though it is somewhat unlikely that a duplicate type will be generated, it is possible and would prevent the module from compiling. I suspect also that interfaces may not be duplicated. Instituting a naming convention in which translated types, booleans, and roles have their module names in front (such as `university_beauregard_t` and `university_ta_active`) of them should nearly gaurentee freedom from collisions.

Interactions in which the entire policy becomes less secure may occur on a policy-by-policy basis, but those are difficult to discern yet because I have not implemented the entire system and do not have test cases. However, because Tolypeutes cannot yet confine non-user processes, it is unlikely that there will be interactions that change how specific processes are confined. At the same time, allowing a user unreasonable accesses is always possible and policy-dependent.

Other than these, the ARBAC modules, because they deal with such a different component of the system than most SELinux policies, I suspect that their interactions will be few. SELinux does not currently confine user processes by default and Tolypeutes can not currently confine other processes.

### 4.4.4 Translation to SELinux: Example of Language Semantics

This section gives an example of the exact translation between Tolypeutes ARBAC and SELinux, as seen in Fig. 4.5 in the context of the University policy. Fig. 4.5 has all of the relevant rules that can be translated into SELinux policy code in a shorter version of the policy: module naming, user, role, resource, and permission declaration, and permission assignment. The rules that cannot be translated are hierarchy declarations and administrative rules (assignment, revocation, and SMER constraints). The SELinux code generated

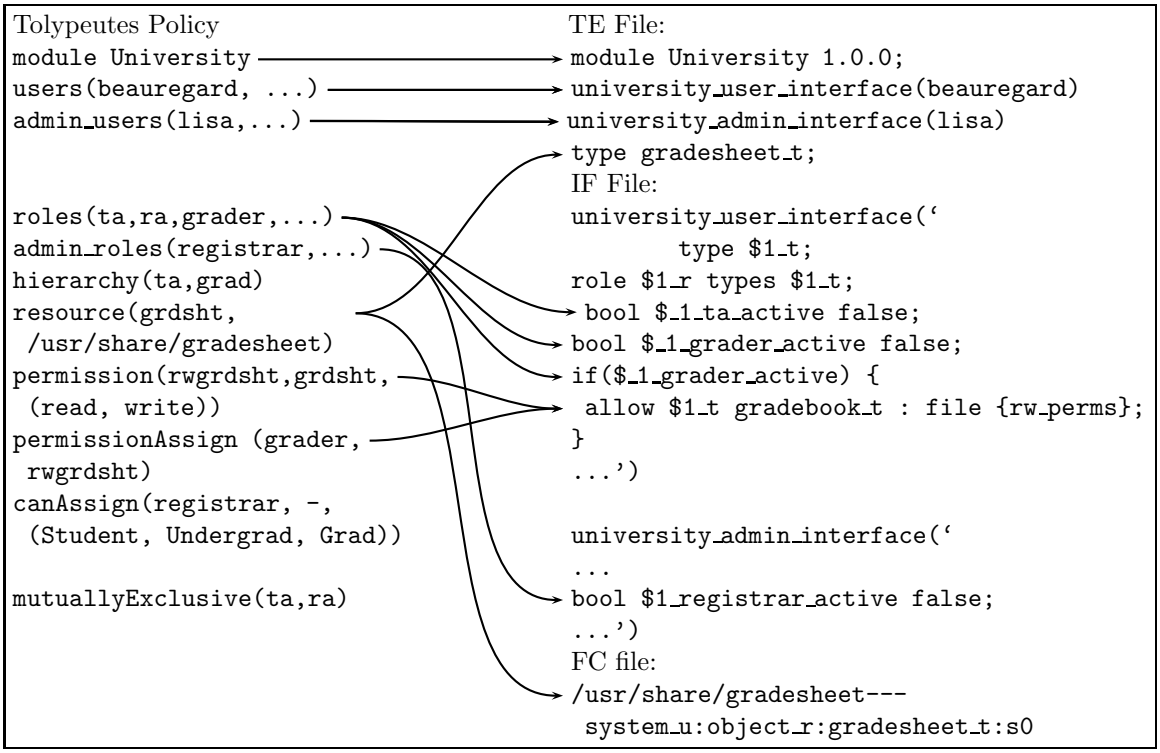


Figure 4.5: Translation of important Tolypeutes rules to SELinux code, in the context of the University example. ARBAC rules without arrows are not translated.

is split into its three files, the TE, IF, and FC files. Though the IF file is not complete, it is simple to see the extension: more boolean declarations and `if` statement blocks in each interface for each role.

### 4.4.5 Important Conditions

The generated SELinux policy code must be syntactically correct and not collide with other modules in any way. It is more difficult than it may seem to create an SELinux module that compiles without any problems. If there are any problems, the abstraction barrier between Tolypeutes and SELinux is broken and administrators would have to deal with SELinux alone, something Tolypeutes was designed to avoid. The implementation must be very strong and very well-tested to make sure there are no, or at least extremely few, circumstances under which non-compiling SELinux modules are generated.

## 4.5 Configuration File Generator

This component supports the non-SELinux utilities: the user batch-adder and the administrative constraints enforcer. It is mostly invisible, or should be, because it lies between the language, which administrators use to specify a policy, and administrative constraint enforcement and adding users, with which administrators must interface if they wish to use

their powers. No administrator needs to nor should ever see it. Because of this, psychological acceptability is not part of my design for it or the files it creates, unlike the other components. My main concern with it is economy of mechanism: I would like it to be as brief as possible and generate the simplest possible files so that the parts that need them have the simplest possible time extracting their information. Making it as simple as possible will help make it as robust as possible, further ensuring that no user needs to see its produced files.

The configuration file generator works on the same transformer-generated file as the SELinux policy code generator. Like the code generator, it only needs certain information: in its case, it needs user information, role hierarchy information, and SMER, assignment, and revocation information. It does not need permission or resource information. It uses usernames specified in the policy to generate the configuration file for the user batch-adder, which is the simplest part of the Tolypeutes system and requires only username information. The other information it uses to generate the configuration file for the administrative constraints enforcer, which needs to know what administrators are allowed to do and how the hierarchy affects them and their actions.

### 4.5.1 Important Conditions

The configuration file generator absolutely cannot generate malformed files because it is outside of any human's control and the generated files are difficult to decipher. Should a problem ever occur such that it starts generating malformed files, administrators would have to be able to parse and then modify the generated files, breaking an abstraction barrier and significantly decreasing the system's usability. This is also assuming that any administrator has access to the generated files without deactivating SELinux, which may be the case depending upon the policy and if it changes how the ARBAC utilities are secured. I have not implemented this component yet, so I am not sure how the standard security around it would be. Because of its importance in enforcing administrative constraints, I would assume that its integrity should be highly valued and protected.

## 4.6 Linux/SELinux User Batch-Adder

One of the more esoteric areas of SELinux is how SELinux users are associated with Linux users and how to specify SELinux user identifiers for Linux user logins. Because there are supposed to be substantial abstraction barriers between Tolypeutes and SELinux, Tolypeutes must be able to take care of this problem. Luckily, this is also the easiest subproblem and also possible to use without the rest of the Tolypeutes infrastructure, though it would be annoying for a human to generate its configuration file.

The batch adder takes a simple file the configuration file generator creates and reads from it all of the users and administrative users in the ARBAC policy. It assumes that

none of these users already exist in the system, whether Linux or SELinux. However, if they do already exist, then no harm is done because both Linux and SELinux will not allow duplicate users (if there is a typo, however, a new user will be created). For each user it generates the required login mapping file (`/etc/selinux/⟨type⟩/contexts/users/⟨user⟩`), appends the required lines to the default contexts file (`/etc/selinux/⟨type⟩/contexts/default_contexts`), adds an SELinux user via `semanage`, adds a Linux user via `useradd`, and creates the custom mapping via `semanage`.

## 4.7 Tolypeutes Administrative Constraints Enforcement

Because SELinux does not have the capabilities to enforce administrative and SMER constraints, I had to devise a way to do so that would both affect how the SELinux policy is enforced and be very easy to use. The path I chose is modifying the `setsebool` utility. `setsebool` is the only tool with which SELinux boolean values can be changed, allowing any user with sufficient administrative privilege to change the value of any policy boolean. This utility can be made into the administrative constraints enforcer by modifying it to check who's doing it and if they have the right roles.

The most relevant design principle relating to this Tolypeutes component is psychological acceptability. User administration must be easy for the entire system to be worthwhile. Else, I have missed one of my main goals: improving the state of security on Linux systems by improving usability. This is why I have chosen to keep the simple syntax `setsebool` uses and call the program `userrole`. If I had kept the `setsebool` name, it would bring SELinux to the front in administration, which is not something I want to do. Though it still uses the exact SELinux boolean names (`<user>_<role>_active`), I expect it to be obvious that these names correspond to users' roles and their state.

### 4.7.1 Implementation

As stated before, the `setsebool` wrapper takes its input the same way `setsebool` itself does:

```
userrole [-P] boolean value | bool1=val1 bool2=val2 ...
```

The `-P` option ensures that the boolean default value is changed (without it, the next time the system boots the boolean reverts to its previous value). It takes either a single boolean or a number of booleans. Valid values are on/off, true/false, and 1/0.

The program checks the caller's SELinux ID and all booleans against a configuration file. The configuration file contains the administrative rules defined in the ARBAC policy the administrators defined, albeit in a different form, and the caller's ID and requested boolean changes are checked against those rules. The change succeeds if there is a rule present for each requested change that allows it. However, if an administrator attempts a change for which he or she is not authorized, or which is malformed, that particular change

fails. If other changes are allowed, they succeed. Each denial comes with a warning that the specific change was denied by the ARBAC policy. I hope that this design makes it easy for administrators to administer users and helps them have a good idea of what they can and cannot do.

### 4.7.2 Important Conditions

The most obvious weakness of this component is that it must be very secure itself, resistant as possible to compromise by any means. The program file must be untouchable, used only in its official capacity by authorized users: this can be accomplished by means of an SELinux module. I have written such a module, but it is not complete. The program itself must be sure to perform only what it's meant to perform and without obvious weaknesses like unchecked input. Unfortunately it is very difficult to verify even a small imperative program such as the `setsebool` wrapper, though I have done my best to ensure I use the safest mechanisms.

### 4.7.3 Weaknesses

The constraints enforcer is a single point of failure on the system — if it is compromised, the ARBAC policy can no longer be securely administered, assuming it was secure in the first place. However, to have multiple methods of administration would mean a larger and more complex system. I am balancing both time constraints and economy of mechanism with fail-safe defaults and defense in depth.

SELinux is not, in fact, as hidden as it might be. Though `setsebool` is wrapped, `getsebool` is not. In order for administrators to view the current policy boolean values (and thus the state of all users constrained by the Tolypeutes system), they must still call `getsebool`. This breaks the abstraction barrier between Tolypeutes and SELinux and is a candidate for future work tightening up the system.

## 4.8 Beauregard, Continued

So far, administrators have come up with and implemented an ARBAC policy to cover the University. One of the modules included is Beauregard's entering undergraduate class, all of the members of which are in the Undergrad role (and in the Student role implicitly, due to the hierarchy). After the administrators, probably AdmOfficers, have added Beauregard and his classmates in another Tolypeutes language file (in a `users` statement), they have to compile that file. The SELinux code generator creates a little module (perhaps named class_of_2013) and the Admissions Officers (AdmOfficer role) run the generated install script, installing the SELinux class_of_2013 module.

The configuration file generator makes files for the administrative constraints enforcer and the user batch-adder. The Dean runs the batch-adder on its configuration file, success-

fully adding Beauregard and his classmates as users on the system with unique SELinux identifiers. This is the end of the initial setup; Beauregard and all of his classmates have uniform ARBAC constraints.

Beauregard gets to the University and does very well in his first semester CS1 class, for which he was able to register because of his Undergrad permissions. The professor asks him to be a grader for the next semester and he accepts. In order to be able to perform Grader functions, the professor, in the Professor administrative role, must assign Beauregard to the Grader role. So the professor uses `userrole` to add Beauregard to the Grader role, which implicitly also adds him to the Employee role.

The years pass by and Beauregard is a happy grader for CS1. However, in his senior year, his grades start to drop off so sharply that his Dean decides that he is no longer fit to be a grader for CS1 and must concentrate on his studies. The Dean has revoke power for users assigned to Grader (though not assign power) and revokes Beauregard from the Grader role and the Employee role. The Dean must explicitly revoke Beauregard from the Employee role because it is possible, though not the case here, that Beauregard has a non-technical job as well with the University that requires him to still be in the Employee role but otherwise does not require additional access rights. Revoking him from the Employee role deprives him of the permission to use the Timesheet program and any other Employee-specific accesses and revoking him from the Grader role deprives him of write access to the Gradesheet and other Grader permissions.

Despite his academic troubles, Beauregard manages to graduate from the University. Indeed, he does more than merely graduate: he had applied to the masters program at the University in his fall semester and was accepted. When the administrators must get him into the Grad role, they must do their assignments and revocations in a specifif order. Because there is a SMER constraint barring users from being in both the Grad and an Undergrad roles, the Dean must first unassign Beauregard from the Undergrad role and then assign him to the Grad role.

Beauregard arrives at the the University the next fall for his first semester of graduate studies. He registers for graduate-level classes, thanks to his Grad role permissions. Because his parents disowned him over the summer for getting a tattoo, he has no money. The graduate school is kind enough to offer him a teaching assistantship, which he gladly accepts. The professor for whom he is TAing is an Assistant Professor, but still able to assign him to the TA role (and implicitly the Employee role, again). This gives him permission to write the Gradesheet again, amongst other TA permissions.

Beauregard completes his masters in two years. Right after his graduation, the Dean revokes him from the TA, Grad, Employee, and Student roles. Administrators edit the ARBAC policy to remove Beauregard (and anyone else leaving that year), remove his user and SELinux files from the system, and no trace of him whatsoever is left.

## 4.9 Tolypeutes Design Strengths

Some of Tolypeutes's strengths are derived directly from SELinux, others from the strength of the ARBAC model, and the others from its design and implementation. Those strengths that come from Tolypeutes being implemented over SELinux are improvements over Linux DAC, those that come from the ARBAC model are over SELinux's design flaws in psychological acceptability, and those from the Tolypeutes design itself are improvements on SELinux's usability.

I have already explained, in Chapter 2, that SELinux fixes Linux DAC's privilege escalation problem, or does if it is used correctly. I did this in terms of the `passwd` program, which is easy to use as an example because it is so critical and sensitive and yet has this security weakness. This is very important in its own right but is also indicative of a more overarching problem with Linux DAC that results in privilege escalation.

In some ways Linux DAC is more flexible than SELinux and can allow a greater variety of accesses (an example is the drop problem, discussed previously and in Chapter 5), but in a critical way SELinux is more flexible. This is the distinction between users and processes. In Linux DAC, processes have user IDs (UIDs) and group IDs (GIDs). A process's UID and GID is derived from the user that started it. A program can be set to be setUID or setGID, changing their effective user and group IDs to be better able to access resources, but this is the reason privilege escalation is a problem. The reason setUID and setGID need to be done is because the resources on the system may only have one owner and one group. The single owner is not a problem, but limiting resources to one group can be a huge problem. This is demonstrated in a simple gradebook example. A gradebook can only be read by students, but can also be written by TAs, Graders, and Professors. The owner would probably be the professor. Then there are two distinct groups that need different sets of accesses to the file, but only one group is available. The solution would be to make one group the official group and build an executable to allow writing/reading through setGID.

The advantage that SELinux has over this model is that of distinct identifiers for each process (if desired), each with a possibly unique set of accesses to whichever resources it needs. When a process forks a different process (if it is allowed), the child process can gets its own SELinux security context, probably different from its parent's. The child process's accesses requests are mediated by means of its own context. There is no need to use things like setUID and setGID if each process can transition to only those processes it's allowed to and it has all the accesses to files it needs. Tolypeutes inherits this advantage, allowing policy writers to create as fine-grained access control as they wish (though currently only for user processes). For users it confines, it neutralizes the privilege escalation problem insofar as SELinux and SELinux-confined processes do.

Tolypeutes's design strengths in terms of the ARBAC model are primarily in usability, through its improvements in economy of mechanism and scalability and flexibility. It should be clear from Chapter 2 that ARBAC is a far simpler and more understandable model

than TE. Its scalability and flexibility strengths are mostly derived from the administrative model, though having a role hierarchy as the most fundamental policy structure is certainly advantageous if the core of a policy needs to be modified, allowing administrators to think in terms of first roles, then hierarchy, then how users should be associated with the hierarchy. In SELinux, it is possible to compartmentalize parts of the TE policy, as is demonstrated by the very nature of this project, but it is still necessary to understand many, many parts of the underlying system in order to use it effectively. In addition, administering the policy for a dynamic system is extremely challenging with such a brittle policy design. To change anything, the correct module must be located and changed in such a way that it does not adversely affect the rest of the policy, which can be very difficult. It is not that unintended consequences of changing the policy are not seen in ARBAC policies, but the simpler mechanism makes it less likely. To administer a system that uses an ARBAC policy, one changes roles to which users are assigned. In staying close to the ARBAC model, Tolypeutes is able to hide system details SELinux has exposed, making it much easier for policy writers to specify the security policies they see in their minds, and simplified administration.

Tolypeutes's inherent design strengths, not derived from ARBAC or SELinux, are mostly to do with administration. The ARBAC administration model is very simple, but the design decisions I made for Tolypeutes not only keeps it close to that simple model but also makes it dynamic and immediately enforced. Deciding to use SELinux booleans to represent roles makes the resultant SELinux policy less readable and larger in the end, but its advantages outweigh its costs, both for usability and security. The user batch-adder, though it is ultimately administrative sugar and not truly required, makes administrators' lives easier by allowing them to avoid as much of the SELinux details as possible and expediting the process in general.

# Chapter 5

# Conclusion

## 5.1 Summary

Tolypeutes is not yet a fully functional system, but the idea and design are good. Using SELinux as a base for enforcing more readable, writable, and understandable policies is a way to both get power and avoid making and dealing with low-level specifications.

ARBAC represents an excellent model especially for reasons of usability, flexibility, and scalability. It confines users, a subset of the system that SELinux does not, but is logically extensible to processes. Though it has not been fully implemented on any system that I can find, it is a good paradigm and I hope that it is soon taken out of the realm of theoretical analysis and into the realm of the average system administrator where it can benefit people like those currently trying to use things like SELinux to define policies.

SELinux, for all its power and fine-grained specificity, is not in wide use by those to whom it is available because it is just too difficult to understand and use. Even if it's true that it doesn't create more complexities than are already present on the average system, that doesn't excuse it from usability issues. However, this does not mean that it should be abandoned. It is extremely useful and powerful as a base for higher-level systems because it implements hooks into the kernel for checking interactions, expands the inode, and any number of other deep kernel things that I don't know about. What I would ideally like to see happen is the integration of some higher-level system into it, so that debugging the standard policy becomes more reasonable and implementing more higher-level tools and languages that use SELinux is made simpler.

Tolypeutes is but a first step towards that ultimate goal: an way to specify truly, or at least reasonably, secure policies in an abstracted way that will be rigorously enforced. I have chosen ARBAC as a security model because it, to me, makes the most sense for combining strict user confinement and easy, dynamic administration of those users' accesses. It keeps as close as it reasonably can to the ARBAC model, and its language is as abstract, given the realities of system administration and complexities of having SELinux as its base.

The Tolypeutes implementation is also very much incomplete. The transformer and

configuration file generator have not been written and the SELinux code generator and administrative constraint enforcer are incomplete. These are components that are future work.

## 5.2 Future Work

Many Tolypeutes components I did not get to finish, as I stated before. More relevant for this section, however, are ideas and extensions that I did not get to explore but would nonetheless be extremely valuable, both for better security and usability.

### 5.2.1 Tolypeutes Extensions

**Confining Processes with ARBAC, Replacing SELinux?** Though this project concentrates on confining users, ARBAC is not intrinsically limited to user process confinement. As it stands, there needs to be existing an SELinux policy confining/defining what users may access, or the administrators must write it. I would like it to be possible for administrators to avoid working directly with SELinux at all, and to do this would require extending Tolypeutes to processes. Users are just glorified processes, so it should be possible to specify ARBAC policies about processes in the same way as users. There is currently no proof that TE and ARBAC are able to express the same set of things, or that ARBAC can only express a subset of the things TE can express, and I have no proof myself one way or the other. Despite this, I can offer a small example of how Tolypeutes might in the future be used to confine processes in the same way SELinux TE can currently, in the form of the `passwd` program:

```
roles(terminaluser,shadowaccess)
resource(/lib/*, library)
resource(/dev/tty*, ttydevices)

permission_assign(terminaluser, (library (read), ttydevices (use)))
permission_assign(shadowaccess, (/etc/shadow (read, write)))

user_assign(/usr/bin/passwd, (terminaluser, shadowaccess))
```

This small example could be processed the same way as a regular user process would be, but it would have to create two types for the 'user' field of the `user_assign` rule instead of one (the process type and entrance file type) and leave out adding the 'user' to the system. Otherwise its translation is identical to that of users with the same administration model.

In terms of what advantages there are to using ARBAC for this when processes have less generalized needs than users, there are certainly fewer. It is possible that each

process would need either a few special, process-specific roles with all of its special permissions, or a large collection of roles that have very few, but more general, permissions. Both of these models are more difficult to administer and organize.

Replacing SELinux entirely with any form of ARBAC would be a serious undertaking simply because hacking the kernel is a very complicated endeavor. I would not venture such a thing and do not believe it is necessary; there is a perfectly okay language for specifying low-level security, so why bother doing that all over again? If ARBAC is able to specify any TE confinement, it is just as reasonable to implement it on top and not instead of, though additional modifications to support dynamic administration would be helpful. In addition, it could be difficult to confine from scratch all of the processes SELinux currently confines in its shipped policy. The makers of SELinux have already figured out all of the interactions for the daemons they have confined.

**Dealing with the `drop` Problem** There was one problem that I was faced with that I could not find a good solution for in ARBAC. This is the `drop` problem, first presented in Chapter 4. The reason this is a problem for my implementation of ARBAC is that SELinux booleans cannot contain type-attribute associations. Staff need to be able to access user-specific types, but giving them access to those specific types within the TA/Grader/Professor roles requires either a more complex compiler routine to give special `drop`-related permissions to specific users (not roles) or have the `drop` objects be more simply labeled. Neither of these are ideal situations. Drops are class-specific and cannot necessarily be tied to the role booleans TA and Student. Even class-specific roles, such as `CS101_TA`, would have the same problem of needing access to user-specific types that are not available before the policy is fully defined.

If the more complex compiler process were chosen and user-specific permissions inserted, it could be unreasonable because administering them could become a deeper matter than changing the value of their role values, especially if attributes were used as the structure giving access to user types. However, it is conceivable that it could be done with plain `allow` rules within the boolean statements if the roles are class-specific. More simply labeling the `drop` objects is a bad idea because then they could be less secure.

At this time, it seems that the `drop` program security is best left to Linux DAC. It does a good job of keeping students out of each other's files and does not involve privilege escalation problems. While it would be ideal to achieve defense in depth with SELinux/ARBAC, problems like this don't lend themselves well to it.

**User-created Language Abstractions** What programming language is complete without constructs to allow users to specify their own abstractions? Though the Tolypeutes policy specification language is declarative, there is still a place for them.

**SELinux Compatible Macro Library** There are a huge number of very useful SELinux macros. These make it possible for people without in-depth knowledge of the entire system to customize policy and constrain aspects of the system they wouldn't otherwise be able to. It would be a great advantage to be able to use any macros that are compatible with Tolypeutes to be able to deal with these same issues.

**ARBAC Policy Analysis** Policy analysis is the process of finding out interesting properties of the policy, such as information flow and whether roles are reachable through any series of assignments/revocations [SYSR06], [SYRG07]. Being able to analyze policy to make sure certain properties hold true makes it easier to make sure it's secure. Therefore policy analysis would be a worthy extension to an ARBAC system; there can be many unforeseen role and user interactions in a policy of even modest size. Luckily, there has been much work on ARBAC policy analysis. One unfortunate conclusion is that many unbounded ARBAC policy analyses have a complexity level beyond the capability of modern machines [SYSR06], [SYRG07]. Nevertheless, there should be some analyses that could be performed on a policy to answer basic security questions.

**A Tolypeutes GUI** Early in the design phase for Tolypeutes, it was suggested that it be implemented as a graphical language. I rejected this suggestion because it would require the development of the language and all of the underlying tools anyway, and I didn't have time to do both that and develop a sound GUI. However, it is a great idea, especially for an ARBAC language due to their easy graph visualizations. It is also a good way to visualize and work with the role booleans.

**Extensible Access Control Markup Language (XACML)** XACML is an OASIS ratified, general-purpose access control policy specification language using XML schemata [GM03]. It is not specified for a certain file system, operating system, or even access control paradigm; it is self-contained and lives outside of the kernel, enforcing access control policies as well as providing a way to specify them. That it lives entirely outside the kernel and enforces security policies from userspace concerns me, but because it is a well-defined and well-reviewed standard, it is an excellent choice for large organizations wishing to create access control policies that will work across systems and withstand radical changes. I do not know very much about XACML, but it is certainly worthy of future investigation as a possible implementation substructure simply because it is a standard.

### 5.2.2 Incomplete Implementations

**Language Transformer and Configuration File Generator** Neither of these components have been written.

**SELinux Code Generator and Constraint Enforcer** Neither of these components are fully implemented. The code generator has some has some setup for organizing input and output data and reading input data from the transformer, but does not actually generate any SELinux code. The constraint enforcer does not have its precondition and constraints checking fully implemented.

**Securing the Linux Utilities** The importance of the policy-related Linux utilities to the security of the system, most especially the `setsebool` wrapper, requires them to be as secure as possible. This means that only the right users may use them, ever, that they cannot be redirected to do malicious things, and that their stated functionality is spotless. To ensure this would require a rigorous analysis of their actions and a well-defined SELinux module dedicated to them in addition to DAC restrictions. As it stands, the `setsebool` wrapper has a minimal SELinux module confining it. The user batch-adder is currently unconfined except by DAC. An SELinux module would be simple to create because it would be very similar, at least in structure, to the `setsebool` wrapper, though it would have to have access to some delicate utilities such as `useradd` and `semanage`.

Similar to the Linux utilities security is the security around the configuration files that they rely upon. The exception is that these must be inaccessible to anyone, and anything, other than the configuration file generator. As long as the configuration file generator is sure to generate correct files, no administrator should be able to modify them, as doing such could give them the ability to change the administration powers confined by the `setsebool` wrapper, which is the entire point of the wrapper and configuration in the first place. Ideally these would be encrypted files that the `setsebool` wrapper and other utilities would be able to decrypt, but that is currently beyond the scope of this project and my knowledge.

# Bibliography


[Bau06] Mick Bauer. Paranoid Penguin - An Introduction to Novell AppArmor. http://www.linuxjournal.com/article/9036, July 2006.

[GLS+09] Mikhail Goffman, Ruiqi Luo, Ayla Solomon, Yingbin Zhang, Ping Yang, and Scott Stoller. RBAC-PAT: A policy analysis tool for role-based access control. In *Fifteenth International Conference on the Tools and Algorithms for the Construction and Analysis of Systems*. ETAPS, 2009.

[GM03] Simon Godik and Tim Moses. eXtensible access control markup language (XACML) version 1.1: Committee specification. Technical report, OASIS Open, August 2003.

[LM07] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *ASIACCS'07*. ACM, 2007.

[MMC07] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example*. Pearson Education, Inc, 2007.

[NS08] Yuichi Nakamura and Yoshiki Sameshima. SElinux for consumer electronic devices. In *Proceedings of the SELinux Symposium*, pages 125–133, 2008.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[OS00] Sylvia Osborn and Ravi Sandhu. Configuring role-based access control to enforce mandatory and discretionary access control paradigms. *ACM Transactions on Information and System Security*, 3(1):85–106, May 2000.

[Ott07] Amon Ott. Rsbac handbook. June 2007.

[Qui07] David Quigley. PLEASE: Policy language for easy administration of SElinux. Master's thesis, Department of Computer Science, Stony Brook University, May 2007.

[Sha07] Rebecca Shapiro. Cooperative and democratic system management, June 2007. Wellesley College bachelor's thesis.

[SM08] Sean Smith and John Marchesini. *The Craft of System Security*. Addison-Wesley, 2008.

[Sma02] Stephen Smalley. Re: Role allow rules. June 2002.

[SS75] Jerome Saltzer and Michael Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 9(63):1278–1308, 1975.

[SYRG07] Scott Stoller, Ping Yang, C.R. Ramakrishnan, and Mikhail Gofman. Efficient policy analysis for administrative role based access control. In *Conference on Computer and Communications Security*, pages 445–455. ACM, 2007.

[SYSR06] Amit Sasturkar, Ping Yang, Scott D. Stoller, and C.R. Ramakrishnan. Policy analysis for administrative role based access control. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 124–138. IEEE, 2006.

[Whi08] Peter White. Security configuration DSL. In *SELinux Developers Summit*, 2008.

# Appendix A

# The University Policy in Tolypeutes Language

This is as seen in Chapter 4. The first part is the administrative components of the policy. All of the different parts have their own files.

```
module University
#The above line (the first non-whitespace line in the file)
#declares which policy this file is a part of.

autorevoke
#The above line states that all roles are revokable by the
#administrators who may assign to them.

#Declare roles
admin_roles (President, Dean, Professor, AdmOfficer, AsstProf)

#Declare hierarchy
admin_hierarchy(Professor, AsstProf)

#administration rules for Presidents. The () for preconditions
#means no preconditions.
can_assign(President, (), (Dean, Professor, AdmOfficer))
#Because this is in autorevoke mode, to make roles unrevocable
#requires a little syntactic twist.
can_revoke(President, (), (!Dean, !Professor, !AdmOfficer))

#administration for Dean
can_assign(Dean, (), Grad)
can_revoke(Dean, (), (TA, RA, Grader, Undergrad, Employee))

#administration for Professor
can_assign(Professor, Grad, (RA, TA))
can_assign(Professor, Undergrad, Grader)

#administration for AdmisOfficer
can_assign(AdmisOfficer, (-Grad), Undergrad)
```

The second section is the non-administrative part of the University policy.

```
module University
#All files labeled with the same module name are part of the same module.

#Declare roles and administrative roles
roles (Student, Employee, Grad, Undergrad, TA, RA, Grader)

#Declare hierarchy. Remember: Senior first!
hierarchy((Grad, Undergrad), Student)
hierarchy((TA, RA, Grader), Employee)
hierarchy((TA, RA), Grad)
hierarchy(Grader, Undergrad)

#Declare the exclusive roles. These could also go in
#the administrative section.
mutuallyExclusive(TA, RA)
mutuallyExclusive(Grad, Undergrad)

#Assign permissions to roles. Note ALL, the root of all hierarchies.
permissionAssign(ALL, (login))
permissionAssign(Undergrad, (/drop, (use)))
permissionAssign(Student, (/course_catalog, (read)))
permissionAssign(Employee, (/hours, (use)))
permissionAssign((TA, Grader), (/dropfiles, (read)))
permissionAssign(Grader, gradebook))
permissionAssign(RA, (/labbook, (read, write)))
permissionAssign(Undergrad, (/undergradblog, (read, append)))
permissionAssign(Grad, (/gradblog, (read, append)))
```

The third part is the declaration of users and what roles they are assigned to.

```
module university
#Connect it to the larger University policy.

#Declare users
users(Beauregard, Lisa, Bob, Leonard, Gwen)

#Assign users to roles.
userAssign(Undergrad, (Beauregard, Bob, Leonard))
userAssign(TA, Lisa)
userAssign(Grad, Gwen)
```

# Appendix B

# Codebase

Many parts of Tolypeutes are not complete and the configuration file generator has not been started. The partially complete parts are the SELinux code generator and the administrative constraints enforcer, and the user batch-adder is complete.

## B.1 Tolypeutes Language Transformer

The Tolypeutes transformer has not been started. It was going to be written in Aurochs, a parser-generator for Objective CAML. It would generate files of the following format, though this example was handwritten.

```
University
##########
roles

Undergrad
Grad
Employee
Student
TA
RA
Grader
##########
aroles

Dean
Professor
AsstProf
AdmOfficer
#########
users

Beauregard
Lisa
Leondard
Stacy
Wendy
```

```
Diana
Steve
Bob
#########
ausers

Greta
Seymour
Wendell
Imogen
#########
hierarchy

Grad,Student
Undergrad,Student
TA,Employee
TA,Grad
RA,Employee
RA,Grad
Grader,Employee
Grader,Undergrad
#########
SMER

Grad,Undergrad
RA,TA
#########
ahierarchy

Professor,AsstProf
#########
can_assign

AsstProf,Grad,RA
AsstProf,Grad,TA
AsstProf,Undergrad,Grader
AdmOfficer,-,Undergrad
Professor,-,Grad
#########
can_revoke

Dean,-,TA
Dean,-,RA
Dean,-,Grader
#########
permissions

/gradebook,rwperms,p1
/gradfile,readperms appendperms,p2
/ugradfile,appendperms readperms,p3
...
#########
PA

RA,p1
TA,p1
```

```
Grad,p2
Undergrad,p3
...
#########
UA

Beauregard,Grad
Stacy,TA
Leonard,Undergrad
Lisa,Grad
Wendy,Undergrad
Wendell,AdmOfficer
Greta,AsstProf
Seymour,Professor
Imogen,Dean
```

This example is from the University policy. These files are easier for the configuration file generator and SELinux code generator to parse easily to do their jobs.

## B.2 SELinux Code Generator

The code generator is written in Objective CAML. It is incomplete and there are no guarantees that it is correct.

```
module type NODE = sig

    module StrSet :
      sig
        type elt = String.t
        type t = Set.Make(String).t
        val empty : t
        val is_empty : t -> bool
        val mem : elt -> t -> bool
        val add : elt -> t -> t
        val singleton : elt -> t
        val remove : elt -> t -> t
        val union : t -> t -> t
        val inter : t -> t -> t
        val diff : t -> t -> t
        val compare : t -> t -> int
        val equal : t -> t -> bool
        val subset : t -> t -> bool
        val iter : (elt -> unit) -> t -> unit
        val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
        val for_all : (elt -> bool) -> t -> bool
        val exists : (elt -> bool) -> t -> bool
        val filter : (elt -> bool) -> t -> t
        val partition : (elt -> bool) -> t -> t * t
        val cardinal : t -> int
        val elements : t -> elt list
        val min_elt : t -> elt
        val max_elt : t -> elt
        val choose : t -> elt
```

```
        val split : elt -> t -> t * bool * t
      end
  (*type set = StrSet.t*)
  type node = Node of string * StrSet.t ref * StrSet.t ref * StrSet.t ref *
              StrSet.t ref
  (*val make : string -> node*)
  val label : node -> string
  val juniors : node -> StrSet.t
  val addJunior : node * StrSet.elt -> unit
  val addJuniorAll : node * StrSet.elt -> unit
  val seniors : node -> StrSet.t
  val addSenior : node * StrSet.elt -> unit
  val addSeniorAll : node * StrSet.elt -> unit

end

  (* To get all seniors and all juniors of each node, do a final pass over
     all nodes after they've been totally compiled. *)

module Node : NODE = struct

  module StrSet = Set.Make(String)

  (*type set = StrSet.t*)

  type node = Node of string * StrSet.t ref * StrSet.t ref *
              StrSet.t ref * StrSet.t ref

  (*let make name = let x = Node("", StrSet.t ref,StrSet ref,
                            StrSet,StrSet ref) in x(name,_,_,_,_)*)

  let label (Node (lbl, _,_,_,_)) = lbl

  let juniors (Node (_, jcell,_,_,_)) = !jcell

  let addJunior (Node (_, jcell,_,_,_), newjun) = jcell :=
                StrSet.add newjun !jcell

  let addJuniorAll (Node (_,_,_,jcellall,_), newalljun) = jcellall :=
                   StrSet.add newalljun !jcellall

  let seniors (Node (_,_, scell,_,_)) = !scell

  let addSenior (Node (_,_, scell,_,_), newsen) = scell :=
                StrSet.add newsen !scell

  let addSeniorAll (Node (_,_,_,_,sallcell), newallsen) =
                   sallcell := StrSet.add newallsen !sallcell


end


module type RBAC_RULES = sig

  module StrSet :
```

```
      sig
        type elt = String.t
        type t = Set.Make(String).t
        val empty : t
        val is_empty : t -> bool
        val mem : elt -> t -> bool
        val add : elt -> t -> t
        val singleton : elt -> t
        val remove : elt -> t -> t
        val union : t -> t -> t
        val inter : t -> t -> t
        val diff : t -> t -> t
        val compare : t -> t -> int
        val equal : t -> t -> bool
        val subset : t -> t -> bool
        val iter : (elt -> unit) -> t -> unit
        val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
        val for_all : (elt -> bool) -> t -> bool
        val exists : (elt -> bool) -> t -> bool
        val filter : (elt -> bool) -> t -> t
        val partition : (elt -> bool) -> t -> t * t
        val cardinal : t -> int
        val elements : t -> elt list
        val min_elt : t -> elt
        val max_elt : t -> elt
        val choose : t -> elt
        val split : elt -> t -> t * bool * t
      end
  type rbac_rule = Pa of string * string * StrSet.t ref | Ua of string * string
  val role : rbac_rule -> string
  val user : rbac_rule -> string
  val perms : rbac_rule -> StrSet.t
  val path : rbac_rule -> string
end

module Rbac_rule : RBAC_RULES = struct

  module StrSet = Set.Make(String)

  type rbac_rule = Pa of string * string * StrSet.t ref
       (* rolename, path, permission set *)
   | Ua of string * string (* rolename, username *)


  let role rule =
    match rule with
      Pa (r, _,_) -> r
    | Ua (r, _) -> r

  let user rule =
    match rule with
      Pa (_,_,_) -> ""
    | Ua (_,u) -> u

  let perms rule =
    match rule with
```

```
      Pa (_,_,set) -> !set
    | Ua (_,_) -> StrSet.empty

  let path rule =
    match rule with
      Pa (_,p,_) -> p
    | Ua (_,_) -> ""


end


module Outputs (* : OUTPUTS *) = struct

  open Hashtbl
  open Str

  type dotfc = string * string (* Path, Context . For .fc files. *)

  type dotte =
      TeTypes of string list ref
      (* List of types that need to be declared in a .te file. *)
    | TeReqs of string list ref
      (* List of types that are required in a .te file. *)
    | TeRules of string list ref (
      * List of rules that are required in a .te file. *)

  let addte elt eltlist =
    let typeexp = regexp "^type [a-zA-Z0-9]*;[\t ]*$"
    and reqexp = regexp "^\(type\)\|\(class\)\|\(attribute\) .*;[\t ]*$" in
    match (string_match elt typeexp) with
      true ->
match eltlist with
  TeTypes (l) -> l := elt :: !l
| TeReqs (l) -> l := elt :: !l
| TeRules -> raise (Failure "Cannot add a type expression to a rule list.")
      false ->
        match (string_match elt reqexp) with
  true ->
    match eltlist with
      TeTypes -> raise (Failure "Cannot add a requirement to a types list.")
    | TeReqs (l) -> l := elt :: !l
    | TeRules -> raise (Failure "Cannot ad a requirement to a rule list.")
  false ->
    match eltlist with
      TeTypes -> raise (Failure "Cannot add a non-type to a type list.")
    | TeReqs -> raise (Failure "Cannot add a non-requirement to a requirement list.")
    | TeRules (l) -> l := etl :: !l


  type sbools = (string,string*string) Hashtbl.t (* boolean, contents*interface *)

  type iftypes = (string,string*(string list)*string) Hashtbl.t
       (* type, role*reqtypes*interface . For interface files.*)

  type reqmisc =
```

```
      Require of string * string list ref(* Interface name, requirements *)
    | Misc of string * string list ref(* interface name, misc rules *)

  let getreqs rule =
    match rule with
      Require (_,reqs) -> !reqs
    | Misc (_,rules) -> !rules

  let inface rule =
    match rule with
      Require (iface,_) -> iface
    | Misc (iface,_) -> iface

  let addreq rule newb =
    match rule with
      Require (_,reqs) -> reqs := newb :: !reqs
    | Misc (_,rules) -> rules := newb :: !rules

end


(* This is for parsing input files and filling input structures. *)

#use "StringUtils.ml"
#use "File.ml"
#load "str.cma"

module CreateInput = struct

  open Hashtbl

  open Str

  let input_roles = ref []

  let input_users = ref []

  let input_aroles = ref []

  let input_ausers = ref []

  let input_hier = ref []

  let input_perms = ref []

  let input_pa = ref []

  let input_ua = ref []

  let inputtable = create(45)

  let outputtable = create(45)

  let inputs filename =
    File.fileToLines(filename)
```

```
  (* Takes a list of lines and splits it into a list of x lists of lines*)
  (* delineated by the ---* line, with the first name under being the section name.*)
  let organize list =
    let tline = regexp "^[ \t]*--[-]+\n$" in
    let rec genlists list res =
      match list with
        [] -> res
      | x :: xs' ->
      if (string_match tline x 0) then (*Delimiter line--start new section.*)
        genlists xs' ([]::((List.rev (List.hd res))::(List.tl res)))
      else                              (*Add stuff to the previously created section.*)
        genlists xs' ((x :: (List.hd res)) :: (List.tl res))
    in let alllist = genlists list [[]] in
    ((List.rev (List.hd alllist))::(List.tl alllist))


  let fill toplist =
    let rec filler list active mode =
      match mode with
0 -> (* start section *)
  (match (List.hd active) with
    "users" -> filler list (List.tl active) 1
  | "roles" -> filler list (List.tl active) 2
  | "ausers" -> filler list (List.tl active) 3
  | "aroles" -> filler list (List.tl active) 4
  | "perms" -> filler list (List.tl active) 5
  | "hierarchy" -> filler list (List.tl active) 6
  | "pa" -> filler list (List.tl active) 7
  | "ua" -> filler list (List.tl active) 8
  | _ -> filler (List.tl list) (List.hd list) 9)
      | 1 ->
  (let x = (input_users := !input_users @ active) in
  match list with
    [] -> () (* do nothing--done!*)
  | _ -> filler list active 0)
      | 2 ->
  (let x = (input_roles := !input_roles @ active) in
  match list with
    [] -> ()
  | _ -> filler list active 0)
      | 3 ->
  (let x = (input_ausers := !input_ausers @ active) in
  match list with
    [] -> ()
  | _ -> filler list active 0)
      | 4 ->
  (let x = (input_aroles := !input_aroles @ active) in
  match list with
    [] -> ()
  | _ -> filler list active 0)
      | 5 ->
  (let x = (input_perms := !input_perms @ active) in
  match list with
    [] -> ()
  | _ -> filler list active 0)
      | 6 ->
```

```
  (let x = (input_hier := !input_hier @ active) in
  match list with
    [] -> ()
  |  _ -> filler list active 0)
      | 7 ->
  (let x = (input_pa := !input_pa @ active) in
  match list with
    [] -> ()
  |  _ -> filler list active 0)
      | 8 ->
  (let x = (input_ua := !input_ua @ active) in
  match list with
    [] -> ()
  |  _ -> filler list active 0)
      | 9 ->
  (match list with
    [] -> ()
  |  _ -> filler list active 0)
      | _ -> raise (Failure "This ain't supposed to happen--illegal loop value")
    in filler toplist (List.hd toplist) 0

end
```

## B.3 Configuration File Generator

The configuration file generator has not been written. It would produce files that look like this, an example taken from the University policy, for the administrative constraints enforcer:

```
*_student_active: adm_officer;;
*_employee_active: asst_professor professor;student;
*_grad_active: dean professor;!undergrad;student
*_ta_active: professor;!ra grad;employee
*_ra_active: professor;!ta grad;employee
*_undergrad_active: adm_officer;!grad;student
*_grader_active: professor;undergrad;employee
```

The first field is the boolean. The second is the set of administrators allowed to affect it, the third is the preconditions and SMER constraints, and the last is the booleans that must also be assigned to adhere to the hierarchy.

At the moment, the user batch-adder uses the file the transformer would generate to get the relevant users. However, it can be imagined that the configuration file generator would produce a better-suited file, looking like this:

```
Beauregard,Grad
Stacy,TA
Leonard,Undergrad
Lisa,Grad
Wendy,Undergrad
Wendell,AdmOfficer
```

```
Greta,AsstProf
Seymour,Professor
Imogen,Dean
Diana,
Steve,
Bob
```

This file contains only user information: their names and to what roles they are to be initially assigned, if any. Because the configuration file generator is nonexistent, these files are handwritten. However, it is not difficult to see how easily they could be produced from the transformer-generated file.

## B.4 Administrative Constraint Enforcer

The administrative constraint enforcer is written in Perl. It is incomplete.

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;


#Separate the ARBAC boolean arguments from the non-ARBAC boolean arguments.
#Return them as two separate arrays.
sub cleanse_arbac {
    my @arbac_args;
    my $manybool = $_[0];
    foreach (@ARGV) {
if ($manybool) {
    if ($_ =~ /^.*_.*_active=(1|0)|([oO][nN]|[oO][fF][fF])|
           ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])/) {
push(@arbac_args, $_);
    }
} else {
    if ($_ =~ /^.*_.*_active$/) {
push(@arbac_args, $_);
    }
}
    }
    return @arbac_args;
}

#This subroutine grabs the non-ARBAC arguments.
sub cleanse_reg {
    my @reg_args;
    my $manybool = $_[0];
    foreach (@ARGV) {
#Don't grab -P or blank arguments.
if (($_ =~ /^-P$/) || ($_ =~ /^\s*$/)) {
    next;
}
if ($manybool) {
    if ($_ !~ /^.*_.*_active=(1|0)|([oO][nN]|[oO][fF][fF])|
```

```
           ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])/) {
push(@reg_args, $_);
    }
} else {
    if ($_ !~ /^.*_.*_active$/) {
push(@reg_args, $_);
    }
}
    }
    return @reg_args;
}

#Checks to see if 2nd arg is in array 1st arg.
sub exists_inarr {
    my @admins = $_[0];
    my $admin = $_[1];
    foreach ($_[0]) {
if ($_ =~ /^($admin)$/) {
    return 1;
}
    }
    return 0;
}



#This is the wrapper to the setsebool SELinux utility.
#It takes input from an administrator of the form
#   setsebool [-P] <boolean> <value> | <bool1=val1> <bool2=val2>...
#It takes into account the identity of the calling user. If the
#calling entity is an administrator and is changing booleans
#that it has the permissions to, as defined in a configuration
#file, then the command succeeds as long as the syntax is valid.

#Get the calling administrator. The name is derived from the role prefix.
my $context = qx(id -Z);
my @parts = split(/:/, $context);
my @tparts = split(/_/,$parts[1]);
my $admin = $tparts[0]; #This is the operative field for use with the config file



my $numargs = $#ARGV + 1; #number of arguments to the program.

#Make sure there are arguments. There must be at least one.
if ($numargs < 1) {
    print "Usage: not enough arguments\n";
    exit -1;
}

#persist: call for persistent changes to boolean values
#manybool: many booleans may be changed; no limit to number of args.
#sdex: Starting index of booleans to be checked against config file.
my $persist = 0;
my $manybool = 0;
my $sdex;
```

```
#Determine what kind of arguments this call has: many, or singular.
if ($ARGV[0] =~ /^-P$/) {
    if ($ARGV[1] =~ /^.*=(1|0)|([oO][nN]|[oO][fF][fF])|
                ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])$/) {
#There can be many arguments, not just 3.
$manybool = 1;
    } elsif ($numargs != 3 ) {
print "Usage: not enough arguments.\n";
exit -1;
    }
    $persist = 1;
    $sdex = 1;
}

#If no -P, check for correct argument numbers.
if (!$persist){
    if ($ARGV[0] =~ /^.*=(1|0)|([oO][nN]|[oO][fF][fF])|
                ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])$/) {
#There can be many arguments, not just 3.
$manybool = 1;
    } elsif ($numargs != 2) {
print "Usage: incorrect arguments.\n";
exit -1;
    }
    $sdex = 0;
}

#This will hold the final arguments to setsebool.
my @outputall;
#This holds getsebool argumets that will check the preconditions
#of the user being assigned.
my @outputgetprec;
#Holds getsebool arguments that will check hierarchical roles for a user.
my @outputgetco;

#Open the configuration file (sebool.conf) and get its lines.
open CONFIG, "./sebool.conf" or die $!;
my @allfile = <CONFIG>;
my $numlines = $#allfile + 1;


#Get all of the ARBAC and non-ARBAC/illegal arguments separately.
my @arbargs = cleanse_arbac($manybool);
my @notargs = cleanse_reg($manybool);



######################################################################
######################################################################
#Parse the config file and the input and match them up.
#Start with the config file lines.
#sub parse_config {
my $x = 0; #x and y are loop counters.
my $y = 0;
my $arg; #The
```

```
my @line; #The line of the config file currently working on.
my $boolean; #The boolean currently working on.
while ($y < ($#arbargs + 1)) {
    $arg = $arbargs[$y];
    #Extract boolean field from compact form.
    if ($manybool) {
$arg =~ /(.*)=.*$/;
$arg = $1;
    }
    $x = 0;
    #for each config line, check it against all ARBAC boolean arguments.
    while ($x < $numlines) {
#skip blank lines.
if ($allfile[$x] =~ /^\s*$/) {
    $x = $x + 1;
    next;
}
@line = split(/: /,$allfile[$x]);
if ($#line + 1 != 2) {
    print "Malformed configuration file at line $x. Please recompile your ARBAC policy.\n";
    exit -1;
}
my @checkfields = split(/;/,$line[1]);
#Get the admin, preconds, and jun roles. This should be 3 long.
if ($#checkfields + 1 != 3) {
    print """Malformed configuration file at line $x,
          list fields. Please recompile your ARBAC policy.\n""";
    exit -1;
}

#Extract the operative parts of the ARBAC boolean.
if ($line[0] =~ /^[*]_(.*_active)$/) {
    $boolean = $1;
} else {
    print """Malformed configuration file at line $x,
          boolean field. Please recompile your ARBAC policy.\n""";
    exit -1;
}

#Check if the config file contains the given boolean.
if ($arg =~ m/^(.*)_.*_($boolean)$/) {
    my $user = $1;
    #The user in the argument currently working on
    #(this is part of a program arg, not in the file)
    #If the config file has it, check to see if the
    #change is allowed to this admin.
    my @admins = split(/ /,$checkfields[0]); #All admins allowd for this role.
    if (exists_inarr(@admins, $admin)) {
#Then we need to check the user's booleans against
        #the preconditions to see if this is allowed.
my @preconds = split(/ /, $checkfields[1]);
if ($#preconds != -1) {
    #There are preconditions!
    push(@outputgetprec, "$user"."_$boolean");
} else {
    #There aren't preconditions!
```

```
}
#Then the change is allowed! Format of output string depends upon
#the format of the input: many booleans, or single.
if ($manybool) {
    push(@outputall, $arbargs[$y]);
} else {
    if ($persist) {
push (@outputall,"$arbargs[$y] $ARGV[2]");
    } else {
push(@outputall, "$arbargs[$y] $ARGV[1]");
    }
}
    } else {
print "Warning: Requested change to boolean $arg is not allowed.\n";
    }
    #If the requested change boolean appears in the file, no need to look further.
    last;
}
$x = $x + 1;
    }
    $y = $y + 1;
}
#}
################################################################################
####################################


my $header = "";
if ($persist) {
    $header = "setsebool -P ";
} else {
    $header = "setsebool ";
}

#Parse and ensure correctness of non-ARBAC booleans.
foreach (@notargs) {
    #Don't deal with the -P or blank arguments.
    if ($_ =~ /^-P$/) {
next;
    }
    if ($_ =~ /^\s*$/) {
next;
    }
    if ($manybool) {
#Check to see if the true/false/0/1/on/off is formatted correctly.
if ($_ !~ /^.*=(1|0)|([oO][nN]|[oO][fF][fF])|
       ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])$/) {
    print "Warning: Malformed boolean statement: $_\n";
} else {
    push (@outputall, $_);
    print "Added: $_\n";
}
    } else {
if ($ARGV[2] !~ /(1|0)|([oO][nN]|[oO][fF][fF])|
             ([tT][rR][uU][eE]|[fF][aA][lL][sS][eE])$/) {
    print "Warning: Malformed boolean statement: $_ $ARGV[2]\n";
```

```
} else {
    if ($#notargs > 1) {
       #There are only 3 args, so if all three are in the list, this is required.
push (@outputall, "$_ $ARGV[2]");
    }
}
    }
}

#print "Output lines:\n";
#print "@outputall\n";

foreach (@outputall) {
    system("$header $_");
}
```

## B.5 User Batch-Adder

The user batch-adder is written in Perl. It is complete.

```
#!/usr/bin/perl
use strict;
use warnings;
use diagnostics;

###########################################
####
#Get the arguments and parse them: They are the names of the files to work on.

my $numargs = $#ARGV + 1;
for (my $x = 0; $x < $numargs; $x++) {
    my $filename = $ARGV[$x];
    print ("I have $filename to open!\n");
    &open_file($filename);
}

sub open_file {
    my $flag = 0;
    my @users;
    open(CONFFILE, $_[0]) or die("Could not open policy file.\n");
    while (my $line = <CONFFILE>) {
chomp($line); #Remove newline.
if ($flag == 1) {
    if ($line eq "") {
next;
    } elsif ($line eq "\tadminusers") {
$flag = 2;
    } else {
push(@users, $line);
print ("$line\n");
    }
} elsif ($flag == 2) {
    if ($line eq "") {
last;
    } else {
```

```
push(@users, $line);
    }
} else {
    if ($line eq "\tusers") {
$flag = 1;
next;
    }
}
    }

    ##Add them to the Unix and SELinux user registry.
    foreach (@users) {
my $user = $_;
my $selcontext = "$user"."_u:"."$user"."_r:"."$user"."_t:s0";
print ("SEL Context: $selcontext\n\n");
my $usersfile = "Users File: \nsystem_r:local_login_t:s0\t$user".
   "_r:$user"."_t:s0\nsystem_r:remote_login_t:s0\t$user"."_r:$user".
   "_t:s0\nsystem_r:sshd_t:s0\t$user"."_r:$user"."_t:s0\nsystem_r:crond_t:s0\t$user".
   "_r:$user"."_t:s0\nsystem_r:xdm_t:s0\t$user"."_r:$user".
   "_t:s0\nsystem_r:initrc_su_t:s0\t$user"."_r:$user"."_t:s0\n$user".
   "_r:$user"."_t:s0\t$user"."_r:$user"."_t:s0\n";

my $default_type = "$user"."_r:"."$user"."_t\n";

my $semanage_line1 = "semanage user -a $user"."_u";

my $semanage_line2 = "semanage login -a -s $user"."_u $user";

my $linux_adduser = "useradd -mU -Z $user"."_u $user";


`echo $usersfile > /etc/selinux/targeted/contexts/users/$user"."_u`;
$? = $? >> 8;
if ($? != 0) {
    print ("Could not create SEL users configuration file for $user".
    ". Please check your SELinux configuration and reinput correct path.\n");
    next;
}

`echo $default_type >> /etc/selinux/targeted/contexts/default_type`;
$? = $? >> 8;
if ($? != 0) {
    print ("Could not modify the default_type file to include $user".
    ". Please check your SELinux configuration and reinput correct path.\n");
    `rm -f /etc/selinux/targeted/contexts/users/$user"."_u`;
    next;
}
`$semanage_line1`;
if ($? != 0) {
    print ("SELinux user for $user already exists.\n");
}

`$linux_adduser`;
if ($? != 0) {
    print ("Could not add $user to Linux.\n");
    next;
```

```
}
‘semanage_line2‘;
if ($? != 0) {
    print ("SELinux to Linux user mapping already exists for $user".".\n");
    next;
}
   }
}
```

# Appendix C

# Concept of Generated SELinux Code

Because the SELinux code generator is incomplete, I have no SELinux code that is its product. This is what it would ideally produce in the case of the example University policy; it was written by hand. The first section is the generated interface file, which contains the most important elements of the module.

```
########################################
## <summary>
## Create a regular ARBAC user.
## </summary>
## <param name="domain">
## <summary>
## User's name to be used to derive type, role, and booleans.
## </summary>
## </param>
#

template(`arbac_user_administration_template',`

# Create all possible types for this user. Policy relating to them
# is disabled by default and requires a call to setsebool to activate
# and deactivate them.

require {
type hours_t, hours_exec_t;
type ugrad_register_t, ugrad_register_exec_t;
type grad_register_t, grad_register_exec_t;
type grading_t, grading_exec_t;
type resedit_t, resedit_exec_t;
}

require {
class fd {all_fd_perms};
class process {all_process_perms};
class fifo_file {all_fifo_file_perms};
class filesystem {all_filesystem_perms};
```

```
class dir {all_dir_perms};
class file {all_file_perms};
class chr_file {all_chr_file_perms};
class key {all_key_perms};
class unix_dgram_socket {all_unix_dgram_socket_perms};
class unix_stream_socket {all_unix_stream_socket_perms};
class netlink_route_socket {all_netlink_route_socket_perms};
class netlink_selinux_socket {all_netlink_selinux_socket_perms};
class netlink_audit_socket {all_netlink_audit_socket_perms};
class netlink_socket {all_netlink_socket_perms};
class tcp_socket {all_tcp_socket_perms};
class udp_socket {all_udp_socket_perms};
class rawip_socket {all_rawip_socket_perms};
class shm {all_shm_perms};
class sem {all_sem_perms};
class msgq {all_msgq_perms};
class msg {all_msg_perms};
class socket {all_socket_perms};
class lnk_file {all_lnk_file_perms};
class capability {all_capability_perms};
class sock_file {all_sock_file_perms};
class blk_file {all_blk_file_perms};
class peer {all_peer_perms};
class association {all_association_perms};
class packet {all_packet_perms};
class netif {all_netif_perms};
class node {all_node_perms};
class security {all_security_perms};
class memprotect {all_memprotect_perms};
class system {all_system_perms};
role system_r;
}

drop_typedec($1)

userdom_unpriv_user_template($1)
role $1_r types {resedit_t grading_t grad_register_t ugrad_register_t hours_t $1_drop_t};


bool $1_student_active false;
bool $1_employee_active false;
bool $1_undergrad_active false;
bool $1_grad_active false;
bool $1_grader_active false;
bool $1_ra_active false;
bool $1_ta_active false;


#Compound conditions are a last resort security measure for enforcing
#SMER constraints. They are last resort because there is no way to
#inform administrators that they are violating them if they manage to
#activate both booleans in the condition.

if ($1_ra_active && !$1_ta_active) {
   #Include RA, grad, student, and employee stuff.
   #domtrans_pattern($1_t, resedit_exec_t, resedit_t)
```

```
   allow $1_t research_notebook_t : file {rw_file_perms};
}

if ($1_ta_active && !$1_ra_active) {
   #Include TA, grad, student, and employee stuff.
   #domtrans_pattern($1_t, grading_exec_t, grading_t)
}

if ($1_student_active) {
   #Allow this domain access to student stuff, including
   #the ability to log in.
   drop_per_user_template($1, $1_t)
}

if ($1_employee_active) {
   #Allow access to employee stuff.
   domtrans_pattern($1_t, hours_exec_t, hours_t)
   allow hours_t hours_exec_t : file entrypoint;
}

if ($1_grad_active && !$1_undergrad_active) {
   #Include grad and student stuff.
   #domtrans_pattern($1_t, grad_register_exec_t, grad_register_t)
   allow $1_t grad_file_t : file {rw_file_perms};
}

if ($1_undergrad_active && !$1_grad_active) {
   #Include undergrad and student stuff.
   #domtrans_pattern($1_t, ugrad_register_exec_t, ugrad_register_t)
   allow $1_t ugrad_file_t : file {rw_file_perms};
}

if ($1_grader_active) {
   #Include grader, undergrad, and student stuff.
   domtrans_pattern($1_t, grading_exec_t, grading_t)
   #typeattribute $1_t cs_staff_t;
}

')

########################################
## <summary>
## Create an administrative ARBAC user: Professor. This distinction
## exists because the administrative users often need access to many
## regular users' files that are specific to them. So, to facilitate
## this, the administrative users need access to attributes that cannot
## because declared or associated within boolean statements. This results
## in an additional constraint upon the flexibility of the system:
## administrators cannot be subject to the same dynamic changes as the
##  regular users. To change an administrative user's role, the policy
## will need to be edited to reflect the change. can_assign and can_revoke
## rules will no longer apply to administrators. Unfortunate, but
## needed to have a working system.
## </summary>
## <param name="domain">
## <summary>
```

```
## Administrator's name to be used to derive type, role, and booleans.
## </summary>
## </param>
#

interface(`arbac_administrative_administration_template',`

require {
type sebool_wrapper_t;
type sebool_wrapper_exec_t;
}

require {
class fd {all_fd_perms};
class process {all_process_perms};
class fifo_file {all_fifo_file_perms};
class filesystem {all_filesystem_perms};
class dir {all_dir_perms};
class file {all_file_perms};
class chr_file {all_chr_file_perms};
class key {all_key_perms};
class unix_dgram_socket {all_unix_dgram_socket_perms};
class unix_stream_socket {all_unix_stream_socket_perms};
class netlink_route_socket {all_netlink_route_socket_perms};
class netlink_selinux_socket {all_netlink_selinux_socket_perms};
class netlink_audit_socket {all_netlink_audit_socket_perms};
class netlink_firewall_socket {all_netlink_firewall_socket_perms};
class netlink_tcpdiag_socket {all_netlink_tcpdiag_socket_perms};
class netlink_nflog_socket {all_netlink_nflog_socket_perms};
class netlink_xfrm_socket {all_netlink_xfrm_socket_perms};
class netlink_ip6fw_socket {all_netlink_ip6fw_socket_perms};
class netlink_dnrt_socket {all_netlink_dnrt_socket_perms};
class netlink_kobject_uevent_socket {all_netlink_kobject_uevent_socket_perms};
class netlink_socket {all_netlink_socket_perms};
class tcp_socket {all_tcp_socket_perms};
class udp_socket {all_udp_socket_perms};
class packet_socket {all_packet_socket_perms};
class rawip_socket {all_rawip_socket_perms};
class appletalk_socket {all_appletalk_socket_perms};
class shm {all_shm_perms};
class sem {all_sem_perms};
class msgq {all_msgq_perms};
class msg {all_msg_perms};
class socket {all_socket_perms};
class lnk_file {all_lnk_file_perms};
class capability {all_capability_perms};
class sock_file {all_sock_file_perms};
class blk_file {all_blk_file_perms};
class peer {all_peer_perms};
class association {all_association_perms};
class packet {all_packet_perms};
class netif {all_netif_perms};
class node {all_node_perms};
class security {all_security_perms};
class memprotect {all_memprotect_perms};
class system {all_system_perms};
```

```
role system_r;
}

type $1_t;
role $1_r types {$1_t};

domtrans_pattern($1_t, sebool_wrapper_t, sebool_wrapper_exec_t)

seutil_read_config($1_t)
seutil_read_src_policy($1_t)

bool $1_dean_active false;
bool $1_professor_active false;
bool $1_asst_prof_active false;
bool $1_adm_officer_active false;

if ($1_dean_active) {
   #No especial regular permissions.
}
if ($1_professor_active) {
   #No especial regular permissions. Inherits from asst_prof, though.
}
if ($1_asst_prof_active) {
   allow $1_t gradebook_t : file {rw_file_perms};
}
if ($1_adm_officer_active) {
   #No especial regular permissions.
}
')
```

This second section is the TE file with the policy interface file.

```
module arbac_attempt 2.3.9;

################################################
################################################
#
# TE file for type declarations for the example
# University policy.
#

require {
class dir {all_dir_perms};
class file {all_file_perms};
class process {all_process_perms};
class fd {all_fd_perms};
class fifo_file {all_fifo_file_perms};
}

type resedit_t;
type resedit_exec_t;
type grading_t;
type grading_exec_t;
type ugrad_register_t;
type ugrad_register_exec_t;
```

```
type grad_register_t;
type grad_register_exec_t;
type hours_t;
type hours_exec_t;

type sebool_wrapper_t;
type sebool_wrapper_exec_t;

type research_notebook_t;
type grad_file_t;
type ugrad_file_t;

seutil_domtrans_setsebool(sebool_wrapper_t)
```

The last section is the generated file contexts file for the university policy.

```
# This file contains contexts relevant to to the user part of the testing
# ARBAC module: programs for registration, timesheet submission, grading,
# and research thingydingy.
#

/usr/bin/uregister --system_u:object_r:ugrad_register_exec_t:s0
/usr/bin/gregister --system_u:object_r:grad_register_exec_t:s0
/usr/bin/timesheet --system_u:object_r:hours_exec_t:s0
/usr/bin/gradehelper --system_u:object_r:grading_exec_t:s0
/usr/bin/resedit --system_u:object_r:resedit_exec_t:s0
/usr/local/etc/special_grad_file --system_u:object_r:grad_file_t:s0
/usr/local/etc/special_ugrad_file --system_u:object_r:ugrad_file_t:s0
/usr/local/etc/research_notebook --system_u:object_r:research_notebook_t:s0
```

In addition to these main components of the policy, there is an additional outside module that adds users to the system, calling the interfaces defined in the previous interface file.

```
module class_2013_test 1.0.17;

##########################
#
# Testing the ARBAC module by creating users. Very simple.
#

arbac_user_administration_template(beauregard)
arbac_administrative_administration_template(lyn)
```