

Logic Based Authorization Policy Engineering

Jie Dai

Computer Science Department, Central Michigan University
Mount Pleasant, MI 48859, USA

and

Jim Alves-Foss

Center for Secure And Dependable Software, University of Idaho
Moscow, ID 83844, USA

ABSTRACT

This paper presents an engineering process for authorization policy development. This process includes formal specification, verification, testing and integration. A general architecture along with supporting toolset is described. In addition, a practical solution based on logic programming is further discussed. Finally, an example demonstrating the application of the methodology is provided.

Keywords: Access Control, Authorization Policy Engineering, Prolog, Policy Verification, Policy Testing.

1 INTRODUCTION

Among various security policies, authorization policies are a core requirement for a computer or network system to prevent malicious attacks. In the system, unauthorized users should not be allowed to access any resources by any means. This is presently guaranteed by authentication and access control mechanisms, which are jointly treated as authorization mechanisms. These mechanisms are actually implementations of authorization policies. Therefore, to some extent, we can say that the security of a computer or network system depends on the quality of its authorization policies and their implementation. The primary indication of the quality of a set of policies is that they are correct, consistent, not ambiguous, and not redundant.

Unfortunately, the establishment of correct and conflict free authorization policies in a computer or network system is a complicated issue. These policies have characteristics of multiple *goals*, *levels*, and *conflicts*. *Multiple goals* mean that a computer system usually serves various users in different ways. *Multiple levels* refer that access control usually involves a management hierarchy, and that the policies are often separated by their abstractions and functionalities. *Multiple conflicts* refer to the logical inconsistency, ambiguity and redundancy of authorization policies.

Due to these characteristics, a systematic engineering approach is desirable to help establish authorization policies and remove conflicts in the policies *before* they are integrated with other system functionalities. Currently there exist various models [9, 7, 5, 25, 12, 18, 19] explaining security properties and approaches dealing with delegations [1, 6, 15] and conflicts [14, 4]. We argue that an engineering approach based on software engineering technologies can be developed to assist users and designers in better implementing the models and applying the available approaches.

2 AUTHORIZATION POLICY ENGINEERING

Authorization policies in computer systems are authentication and access control strategies that can be implemented in

authorization mechanisms. These strategies establish a practical way to realize security models or requirements. They are often rules for logical deduction with the purpose of making authorization decision to protect resources from illegal usage. Meanwhile, a sequence of authorization policies can be seen as a plan in a form of a program about relationships between subjects and objects. By executing the program, a system can automatically decide whether or not to permit an access to its resources. Therefore, from this point of view, an engineering approach similar to software engineering can be applied in the generation and analysis of authorization policies. We call the engineering process *authorization policy engineering*. It is a methodology with the goal of allowing system users (end users, system admin, and security officers) to create correct and conflict free policies and software engineers to integrate the policies into other software with the help of *Computer Aided Policy Engineering* (CAPE) Tools. Compared with software engineering, authorization policy engineering has the following similarities:

- *Multiple levels of development*: abstract level for analysis, end user level for unskilled people writing policies, system level for policy refinement, and programmer level for policy coding and integration;
- *Multiple representation languages*: mathematical symbols for formal specification and analysis, visual language for unskilled people writing policies, programming language for implementing the policies and enforcement;
- *Multiple validation approaches*: formal verification and testing can be applied to validate policies.

However, authorization policies distinguish themselves from system or application software of other functionalities by their logical nature. Authorization policy enforcement is actually a logical deduction process, and specified policies may logically contradict to each other. Consequently, conflict detection is extremely important during authorization policy development.

It is important to mention that static conflict analysis in authorization policies heavily depends on the semantics of the description logic as well as the authorization logic itself. For space reasons, this work is not presented in this paper. Interested readers can refer to [11] or future paper for detailed theories that we have developed for this purpose. In this paper, we focus on introducing our authorization policy engineering architecture and how to apply it.

2.1 Characterizing Authorization Policy Engineering

It is important to distinguish our *authorization policy engineering* from other currently researched engineering process.

Compared with *security engineering*, which is a discipline for building systems to remain dependable in the face of malice, error or mischance, and focusing on the tools, processes and methods needed to design, implement and test complete systems in order to adapt existing systems as their environment evolves [2], authorization policy engineering is a smaller arena. In authorization policy engineering, policies are separated away from the system and application software of other functionalities and treated specifically. The rationale lies in:

- *Convenience for security analysis.* Security analysis of a system can be separated into two parts: interface implementation analysis and policy analysis. The interface primarily refers to the policy enforcement mechanism that may be provided as an application programming interface (API) and integrated into other functionalities of a program. Once the interfaces have been proved to be correctly and safely established, security analyzers only need to focus on validation of the policies;
- *Convenience for coding.* Software developers can focus on implementing non-security related algorithms and embedding the general policy enforcement mechanism in their software. This decreases the complexity of the code – as programmers do not have to entwine user policies in their software.
- *Scalability of system security.* With the general CAPE tools, system and end users have the capability to flexibly create, modify, delete, and analyze their security policies for a system or an application based on their own needs.

We assume that the separated authorization policy system is tamper-proof and can be incorporated with other software securely. Due to the separation, the policies can be treated with more rigorous methods during their development process. The correctness and consistency of the policies are the necessary condition for the security of the entire system.

In addition, authorization policy engineering is not *general policy engineering*. In [23], Sloman *et al.* advocate policy driven management for distributed systems, where policies involve both authorization and obligation. There is a series of research on this track [17, 8, 22]. However, we argue that since obligatory policies are usually hard to distinguish from the workflow of a system, we should not separate them from the software. Therefore, our policy set only contains authorization policies, and the conflict that concerns us is the presence of rules that both authorize access and deny access.

2.2 Architectures

The authorization policy engineering architecture is comprised of a *general architecture*, *development architecture*, and *user architecture*.

General architecture. The general architecture for authorization policy engineering can be described with processes of *establishment*, *maintenance*, and *analysis*.

Establishment is about generating a set of executable policies regarding authorization control strategies and then integrating them with software of other functionalities. In this process, policies may be specified, refined and programmed in a process called policy generation. Then some of the policies need to be deployed and associated with entities of a specific computer system, if the system is a distributed one.

Maintenance refers to updating expired policies through adding, deleting (if they are local), delegation, or revoking (if they are remote) policies in a way that the main security model is not affected. For instance, during maintenance, outdated users' accounts or resources are removed and new ones may be added. These operations change the original policies of the

system, but they do not affect authorizations on other users and resources.

Both establishment and maintenance require an *analysis* process for validation and evaluation. In validation, we can conduct theorem proving on formal specifications and testing on the final code in order to guarantee the correctness of the implementation. In evaluation, we should be able to measure the security of the system in a way that we can know the effectiveness of our policy systems. An initial evaluation approach [11] has been established, but it is not presented in this paper for space reasons.

Development architecture. The development architecture refers to the typical *specification-refinement-integration* process in writing authorization policies.

Specification of an authorization policy system includes an abstract level description about security models and system configurations. While security models are mostly focused on relations between a subject and an object, system configurations are mostly about the relations within the subjects and the objects. Therefore, specification on both of them is often disjoint, and we can specify them separately.

Refinement of authorization policies generates authorization programs from policy specifications. Since the specification is focused on what the problem is instead of how to solve the problem, we need a transformation from one level of representation to another where abstract and composite concepts in the specification are replaced with more concrete ones. For instance, a *read* access for an abstract resource in a specification can be refined with many reads of different resources of the system that make up the abstract resource.

Integration. After authorization policies have been converted into executable programs, they are ready to provide authorization services. These services are required to be integrated with other software that desires to utilize resource protection and accesses.

User architecture. Unlike other software, authorization policies are not just implemented by software programmers, but also by system administrators, analyzers and end users. Each of them is concerned with a different aspect of the policies, and their work has to be connected with each other in order to make the entire policy system work. Figure 1 shows our three level architecture which illustrates the abstraction and dependency relationships of the policies generated by the three types of the policy makers.

End user level. Policies at this level are usually made by managers, individuals, or administrators with a user friendly policy description language or graphical user interface (GUI). End user level policies are system independent and conceptual, which means that policy makers do not have to know the underlying system platform. However, these policies may be application dependent so that policy makers may control the usage based on the requirements of a specific application, organizational structure or project management.

System Admin level. Policies at this level are usually made by system administrators or security officers who have more system configuration knowledge and may be trained to use more complicated lower level policy description languages. Policies at this level are both system and application software dependent and may be the refinement of end user level policies. The refinement decomposes policy subject and object domains, and converts a general permission to the more specific lower-level disjoint permissions concerning a system or an application function.

Programmer level. Policies at this level are usually about strategies to control access to an individual file or process. Higher level policies are finally converted to the executable

programs and integrated to the application or system programs through built in interfaces based on the strategies. The general strategies can be built with meta policies and implemented as an application programming interface (API). Policies at this level are only system specific, which means that policies are built based on the lowest policy enforcement mechanism. For instance, if a system uses the ACL based access control mechanism, then the provided APIs are different from those of a system using capabilities.

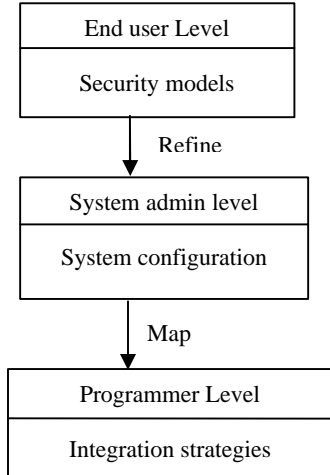


Figure 1 Authorization Policy Development Architecture

Figure 1 also shows the entire policy development scheme, where policy specification, refinement and integration can be applied. More specifically, specification can be used by end users to specify security models and by system administrators to configure their system; refinement can be used by system administrators and programmers to refine higher level policy concepts and generate executable code; and finally, integration is used by programmers to build up API and integrate authorization services in the application and system programs. Secure policy synthesis must follow the following *conformance rule*.

CONFORMANCE RULE: *The lower level security world defined by a new set of derived policies can only have security higher than, if not equal to, that of the higher level world.*

2.3 CAPE Toolset

As in software engineering, we need powerful CAPE tools to support authorization policy development jointly conducted by the end users, system administrators, and software programmers. We identified that two environments are needed to accomplish this. The first environment consists of programming language libraries for software developers to integrate refined authorization policy programs with other functionalities of system and application software. With the API, software developers can map higher-level policies regarding applications, files, and users to the programmer level policies regarding

processes and data structures easily. The second environment is a set of tools, called CAPE toolset, for end users and system administrators. The environment helps the users specify their policies, refine them, and analyze them separately or jointly w/o other system or application functionalities. This toolset can be built as an application in a system to manipulate *end user level* and *system admin level* policies for the system. Figure 2 is our architecture for the CAPE toolset.

Functionally, the CAPE toolset can be separated into *viewers*, *writers*, and *analyzers*, which present policies in different format for users at each level. The following are some of the functions that need to be provided:

- *System overview:* shows principals, subjects, objects and their relationship on the entire system;
- *Policy test:* shows test cases, and conducts testing for policy validation at different levels;
- *Policy verification:* shows specification, axioms, and theorems that have already been built up, records proofs, and supports deductions;
- *Visual policy writing and representation:* represents policies as relational diagrams, such as UML [26] specifications;
- *Policy generation:* refines policy specifications to executable programs. Traceable linkage needs to be built to connect the specification and the programs;
- *Measurement:* calculate domain divisions and rule values regarding restriction on access for each rule and policy;
- *Policy delegation and revocation:* deploy policies or withdraw them to/from other nodes with certification and XML [28] techniques.

3 A LOGICAL SOLUTION

Authorization policies are strategies controlling the access from subjects to objects in order to prevent illegal usage of computer resources. Presently, authorization policies are represented and implemented as simple ACLs, tables, complicated databases, or a collection of declarative statements of rules and facts. Correspondingly, ACL or table checking, database searching, or logical reasoning is used to implement various authorization deductions.

It is nontrivial to realize complicated authorization relations through writing policy programs and conducting automatic reasoning with computers. The difficulty is due to the fact that fundamental logic systems supporting authorization deduction have not been successfully developed yet. For instance, current logic programming (LP) techniques can handle default cases but are still not good at providing negation, inheritance and closure reasoning. Other techniques such as ACLs or databases can provide even less flexibility in representation and no mechanism for logical reasoning, which therefore have to be implemented by programmers from scratch.

In order to realize our idea of authorization policy engineering, we refine the preceding general architecture with currently available LP technologies for the following reasons:

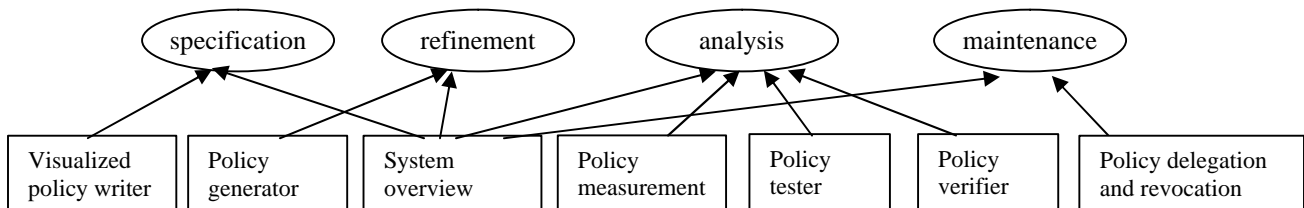


Figure 2 A CAPE Architecture for System Users

Users = {jim, Daniel, jie, nate}	(1)
Groups = {faculty, grad, system_admin}	(2)
Jim ∈ faculty, [Daniel, jie] ∈ grad, nate ∈ system_admin	(3)
Files = {fun, cbass, public, cs445}	(4)
Classified projects = {aro_p, cbass_p}	(5)
Project classifications = {top_secret, confidential, secret, non_classified}	(6)
Top_secret > confidential > secret > non_classified	(7)
Aro_p ∈ top_secret, cbass_p ∈ secret	(8)
Fun ∈ aro_p, cbass ∈ cbass_p	(9)

Figure 3 Configuration of Virtual

An Executable Formal Specification Language. The criticality of authorization policies requires that rigorous mechanisms such as formal methods be applied in the implementation. Basically, there are two typical approaches – *algebraic* and *logic programming* (LP).

From a strictly software engineering point of view, LP can be viewed as a simplified but executable formal specification language. Although LP does not support strong theorem proving, the highly efficient inference engine provides a mechanism to automatically deduce authorization decisions at runtime and statically analyze consistency and correctness of authorization policies, to some extent. With algebraic systems, which have stronger theorem proving support, programmers have to convert the policy specification into executable programs for runtime checking. Whenever such a conversion of notations is required, there is an increased chance of error or disconnect between the levels.

A Knowledge Representation Language. From the logical system point of view, an authorization system can be seen as a specific knowledge representation and reasoning system. In the system, policies represent the knowledge base of security models and system configuration for the intended and secured world, and authorization decisions are derived from the knowledge base with the inference engine of the supporting logic system. Among the various logic systems supporting knowledge representation and intelligent reasoning, general Horn Clause-based LP has been widely used.

Verification. A logic programming system such as Prolog is a declarative formal specification language based on *Horn Clause Logic*. Its procedural semantics allows us to draw logical conclusions in a limited deductive way. Prolog will allow us to ask queries in a form of $L_1 \wedge \dots \wedge L_n$, where L_i is called a *literal*.

In the queries, variables are allowed in the *literals* and are quantified with an *existential* quantifier \exists (although it is generally omitted). By utilizing this, we may verify a system by challenging the policies at various levels with queries. The specification verification in Prolog we used is similar to the approach used in PVS [24]. The rationale of the approach is that if we assume that the specification is correct, then from the specification, desired properties should be derivable from the specification and some established axioms. If the desired property cannot be derived logically, then the previous assumption is wrong, i.e. the specification is incorrect. In section 4, we demonstrate how to conduct verification on incomplete end user level specification and complete system specifications.

Testing. Compared with verification techniques, testing in Prolog can be done rather thoroughly and elegantly. We may directly use the Prolog inference engine to conduct validation on all policy levels by applying different sets of test cases. Since we may ask queries with variables and Prolog retrieves all the matched answers in any case derivable from the program, we

may conduct the test mechanically by making a test processing program that stores the test results and later compares them with our intended answers. Any discrepancy generated by the comparison will be flagged for the tester to check whether it is allowable or not. We may apply an incremental testing method, which runs test cases automatically to validate all three levels of policies when policies have been modified. Traceable links may also be established between the test cases and the policies.

4 EXPERIMENT

Our experiment is conducted on a flexible and dynamic distributed authorization testbed, called “certificate based authorization simulation system” (CBASS) [10], which allows us to manipulate authorization policies from four aspects: generation, refinement, delegation and validation. In the testbed, policies are separated from application and system software. This enables us to conduct policy generation and validation systematically and independently through an engineering process. In the following sections, we use an imaginary example to demonstrate how to apply the developed approach to synthesize and validate authorization policies.

4.1 User requirements

An authorization policy system, called *Virtual*¹, is implemented in CBASS. *Virtual* is configured as depicted in Figure 3. Users who create or use files in a project are also classified as the same security level as the files. In addition, *Virtual* enforces following security models: Bell_LaPadula security model (BL), System admin model (SA), and Discretionary access control (DAC) model. We assume access to files is limited to only read and write operations.

4.2 Specification

We first specify security models. Lower level system details are not considered now, so that the formalized specification can be applied to systems with different configurations. Therefore, we only need to consider requirements from the models BL, SA and DAC, and how these multiple policies are merged together. The specification is made in Prolog.

Figure 4 specifies the BL policy and the combination of the DAC and SA policies. These are then merged through logical disjunction, which distinguishes the system admin from other users. The **DAC+SA+BL** listing specifies the meta-policy used to combine DAC+SA policy with the BL policy. For example, rule (r1) states that any user can read any file when his clearance dominates the classification of the file. Rule (r3) supports DAC policies by stating that a read is permitted if the requestor created the file, or if another user generated local permissions to

¹ The example showed here has been simplified.

read the file or the users is the system administrator. And, rule (r6) merging these two rules requires both policies to be enforced. For a more detailed interpretation, refer to [11].

Now we specify system configuration, which may be seen as a refinement of concepts in the security models at the specification level. Figure 5 refines the subjects and objects of the system and their security levels. The system admin can be at any security level depending on system requirements. In this example, it is defined as in the lowest level primarily for the reason of validation demonstration. This may be not consistent with the situation in the real world. We have also defined the security level of a file to be the same as the level of the files creator. Therefore, there also exists a file hierarchy corresponding to the user hierarchy. We omit its definition here to save space.

We lastly specify the local permissions set by users for their own files. This is usually done at the discretion of each of the users. We put them together here in order to give an overview of entire authorization policies. In addition, this helps to centrally detect an *authorization chain* within a distributed environment – all possible permissions are included in one program. Figure 6 shows some examples of the local permissions.

```

% Bell_LaPadula policies -----
bl_permit (Anyuser, read, Anyfile) :-
    dominate (Anyuser, Anyfile).          (r1)
bl_permit (Anyuser, write, Anyfile) :-
    dominate (Anyfile, Anyuser).          (r2)
dominate(X, Y) :-
    level(X, L1), level(Y, L2), L1 >= L2. (r3)

% DAC and System Admin policies -----
dac_permit (Anyuser, read, Anyfile) :-    (r4)
    is_created (Anyfile, Anyuser);
    is_created (Anyfile, Owner), user (Owner),
    local_permit (Anyuser, read, Anyfile, Owner),
    \+(is_created (Anyfile, Anyuser)),
    \+(system_admin (Anyuser));
    sys_admin (Anyuser),
    \+(is_created (Anyfile, Anyuser)).

dac_permit (Anyuser, write, Anyfile) :-   (r5)
    is_created (Anyfile, Anyuser);
    is_created (Anyfile, Owner), user (Owner),
    local_permit (Anyuser, write, Anyfile, Owner),
    \+(is_created (Anyfile, Anyuser)),
    \+(sys_admin (Owner)),
    \+(system_admin (Anyuser));
    sys_admin (Anyuser),
    \+(is_created (Anyfile, Anyuser)).

% DAC-SA and BL Merged -----
permit (Anyuser, read, Anyfile) :-
    user (Anyuser), file (Anyfile),
    bl_permit (Anyuser, read, Anyfile),
    dac_permit (Anyuser, read, Anyfile).  (r6)
permit (Anyuser, write, Anyfile) :-
    user (Anyuser), file (Anyfile),
    bl_permit (Anyuser, write, Anyfile),
    dac_permit(Anyuser, write, Anyfile).  (r7)

```

Figure 4 BL and DAC-SA Policies Individually and Merged

```

% Users -----
user (Anyuser) :-
    classified_user (Anyuser);
    non_classified_user (Anyuser);
    sys_admin (Anyuser).                  (r8)

% Non Classified Users -----
non_classified_user (Anyuser) :-
    faculty (Anyuser);
    grad (Anyuser).                       (r9)
level (Anyuser, 0) :-
    non_classified_user (Anyuser);
    sys_admin (Anyuser).                  (r10)

% Classified Users -----
classified_user (Anyuser) :-
    top_secret (Anyuser);
    confidential (Anyuser);
    secret (Anyuser).                     (r11)
level (Anyuser, 3) :- top_secret (Anyuser). (r12)
level (Anyuser, 2) :- confidential (Anyuser). (r13)
level (Anyuser, 1) :- secret (Anyuser).      (r14)
top_secret (Anyuser) :- aro_user (Anyuser). (r15)
secret (Anyuser) :- cbass_user (Anyuser).    (r16)

% Files -----
file (Anyfile) :- is_created (Anyfile, Anyuser),
    user (Anyuser).                       (r17)
level (Anyfile, L) :- is_created (Anyfile, Anyuser),
    user (Anyuser), level (Anyuser, L).      (r18)

% Project Users -----
aro_user (daniel).                       (r19)
cbass_user (jie).                        (r20)
sys_admin (nate).                        (r21)
faculty (jim).                            (r22)

% User Files -----
is_created (fun, daniel).                 (r23)
is_created (cbass, jie).                  (r24)
is_created (public, nate).                (r25)
is_created (cs445, jim).                   (r26)

```

Figure 5 Refinements

```

% Local Permission -----
% Owner of the file cbass is jie
local_permit (Anyone, read, cbass, jie) :-
    aro_user (Anyone).                    (r27)
local_permit (jim, read, cbass, jie).     (r28)

% Owner of the file public is nate
local_permit (Anyuser, read, public, nate) :-
    user (Anyuser).                       (r29)

% Owner of the file cs445 is jim
local_permit (jie, read, cs445, jim).     (r30)
local_permit (jie, write, cs445, jim).    (r31)

```

Figure 6 Local Permissions

All other unstated relations are treated as nonexistence due to the *negation as finite failure* rule of Prolog. By combining the

specification of required security models and system configuration, an executable policy program can be generated. Once this program has been loaded into a Prolog system, we may enforce the policies by asking queries to the Prolog system. For instance, we may present a query “*permit (jie, read, public).*” to the Prolog system. By executing the program, the Prolog system will provide an answer “yes”, granting jie permission to read the file. More complicated queries can be made and will be shown in the examples of the following sections.

4.3 Integration

After we have constructed a Prolog-based authorization policy program for *Virtual*, we can deploy the policies and store them distributively over CBASS. Figure 8 shows the application and system programs that have been implemented in CBASS. All these programs have been built with interfaces to connect the policy system (including static policies as well as Prolog system). Each of the programs (*e.g.* Machine, User, CM, Cat, Cat2, FS, Clock and ClockServer) and data files (*e.g.* fun, cbass, mili, hummer, cs445, intrusion, public) associates with a policy and a credential. The policy has all the rules regarding run, serve, and change. Run policies define who can execute the program, serve policies define under which conditions services of the program can be provided, and change policies define under which conditions the policy and the credential can be modified. The credential stores policies of *facts* which usually identify a program or a user.

To deploy *Virtual* policies on CBASS, we only need to store corresponding policies developed into the system. These policies and credentials will be automatically loaded into the runtime environment if needed. For more information please refer to [10, 11].

4.4 Validation

We have conducted partial verification and testing on *Virtual*. We apply the *challenge* approach in our verification. Since there is no generic definition for computer security, we just probe the specification with some obvious properties in order to show how to conduct the policy verification in Prolog.

Partial verification of security models. Formal verification of specification can be conducted as early as when the security models have been specified, i.e. the entire authorization policy system has been partially specified due to the lack of system configuration information. We have discovered that the proving process with Prolog at this stage is different from that of a traditional algebraic approach. In order to prove a property, instead of trying to find a logical deduction step by step from the premises to a conclusion, we look for a set of reasonable facts to support the goal. If we are able to find the facts, then we say that the intended goal is derivable from the specification, and therefore, the specified system has the intended property. Otherwise, the specified system does not have the intended property, i.e. it is not correctly specified. In the following, we

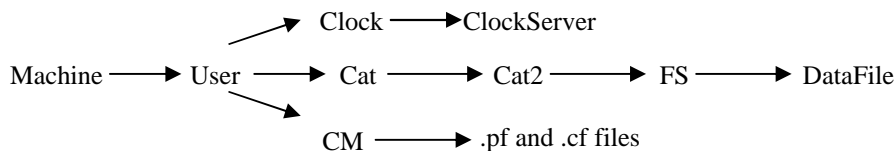


Figure 8 Three Applications in *virtual*

give two examples on how to verify security models without system configuration knowledge.

CHALLENGE 1 (Access for System Administrator). *In the Virtual system, a system administrator cannot access (read and write) all files at any levels.*

In Figure 7, we show a set of facts that have been proved enough to support Challenge 1. Since a system administrator can only be at one level, his access to other levels will be restricted by the BL policies. Figure 7 shows an example that when the system administrator *john* is at level zero, his access to level one file *cbass* would only be *write* but not *read*. This has been proved in our XSB system. In Figure 7, “*is_created (nofile, noone)*” must be added in order to pass the compilation of the Prolog engine. It does not provide extra information regarding *john* and *cbass*, and therefore would not affect our proving logically.

```

level (john,0).
level (cbass,1).
sys_admin (john).
user (john).
file (cbass).
is_created (nofile, noone).
% Queries & Answers -----
| ?- permit (john, write, cbass).
yes
| ?- permit (john, read, cbass).
no
  
```

Figure 7 Facts for CHALLENGE 1

CHALLENGE 2 (Access to Lower Level Files). *A user can read lower level files iff the owner of the files has locally granted the permission.*

In Figure 9, we show our facts to support the Challenge 2. Unfortunately, this is a very weak proof for the challenge – we only show the case when the user is at level 1 and the file is at level 0. In order to complete the proof, we need to consider all security levels in a system. The complete verification for the challenge is what we cannot do now in the Prolog system. As Prolog cannot directly derive based on ungrounded variables, the complete verification in Prolog has to be made based on a complete set of facts, which are still not available at this high level.

From the above two examples, we can see that the practical verification with Prolog at this stage is only suitable for some properties such as Challenge 1. For the properties as Challenge 2, we have to wait until the system configuration is also specified. Or we may construct a complete program for the specification and convert the complete program to another format of specification such as PVS, which can deal with variable deduction to better conduct the high level of formal verification.

```

level (john, 1).
level (cbass, 0).
user (john).
file (cbass).
is_created (cbass, jie).
local_permit (john, read, cbass, jie).
sys_admin (noone).
user (jie).

```

```

%Queries & Answers ----
| ?- permit (john, read, cbass).
yes

```

Figure 9 Facts for Challenge 2

```

| ?- sys_admin (Anyuser), permit (Anyuser, read, Anyfile),
    permit (Anyuser, write, Anyfile), file (Anyfile).
Anyuser = nate
Anyfile = public;
Anyuser = nate
Anyfile = cs445;
no

```

Figure 10 Test on Access for System Administrator

```

| ?- permit (Anyuser, read, Anyfile),
    level (Anyuser,L1), level (Anyfile,L2),
    L1>L2, is_created (Anyfile, Owner),
    user (Anyuser), user (Owner), file (Anyfile),
    \+(local_permit (Anyuser, read,
                    Anyfile, Owner)).           (Q1)
no
| ?- permit (Anyuser, read, Anyfile),
    level (Anyuser, L1), level (Anyfile, L2),
    L1>L2, is_created (Anyfile, Owner),
    user (Anyuser), user (Owner),
    file (Anyfile).                               (Q2)

```

```

Anyuser = daniel
Anyfile = cbass
L1 = 3
L2 = 1
Owner = jie;
Anyuser = daniel
Anyfile = public
L1 = 3
L2 = 0
Owner = nate;
Anyuser = jie
Anyfile = public
L1 = 1
L2 = 0
Owner = nate;
Anyuser = jie
Anyfile = cs445
L1 = 1
L2 = 0
Owner = jim;
no

```

Figure 11 Test on Access to Lower Level Files

Verification of entire specification. Verification for a fully specified authorization policy system checks whether the general security models are correctly applied to a specific system with certain hierarchical structure of subjects and objects. The challenge approach is used for the verification. Since full knowledge about the system is now specified within a program, specific information about users and files is provided. Instead of creating facts to support a theorem as we did in the security model verification, we simply utilize Prolog to search all satisfied answers based on the defined set of facts developed for a specific system configuration, then check whether there are illegal answers there.

The previous two challenges are reused to show how to conduct the verification in a complete specification setting. In Figure 10, we solve the Challenge 1 by making a query “what are the files that a system administrator can read and write in Virtual?” The Prolog engine answers with files *public*, and *cs445*. All the files are at the zero level. This justifies that our program is correctly refined with respect to the system administrator *nate*, who is classified as *non_classified_user* and consequently cannot read all higher level files due to the BL policies.

In Figure 11, challenge 2 can be proved in two ways. Q1 asks “Is there any lower level files allowed to be read if they are not set as readable?” The answer to the query is no. Q2 asks “who are those lower level files that can be read by higher level users?” The answer lists all the matched files whose read permissions have all been locally granted.

Automatic testing for authorization policies. As we know from the previous section, if we ask a query with variables, Prolog will find all matched answers in a program. Therefore if we ask a generic query such as the one in Figure 12, we can actually generate a database by storing all the searched results in a file.

Later, when we modify the program, we can query again and store the results in another database. By comparing the two databases (which can be easily done in a computer), we know whether or not the modification is the intended one or not. By generating a set of such generic queries, called typical test cases, we may conduct automatic testing in the life cycle of a policy program.

Integration testing can also be conducted through the application and system programs in CBASS. We omit it here for space reasons. Interested readers can refer to [11] for more information.

```

| ?- permit (Anyuser, X, Anyfile),
    is_created (Anyfile, Owner), user (Owner),
    user (Anyuser), file (Anyfile).

```

```

Anyuser = daniel
X = read
Anyfile = fun
Owner = Daniel;

```

```

Anyuser = Daniel
X = read
Anyfile = cbass
Owner = jie;

```

```

...

```

Figure 12 Automatic Test Case Generation

5 RELATED WORK

Authorization policy generation is a systematic and independent process including specification, refinement, maintenance, and analysis. However, most of security policy studies were focused on classification and modeling. How to implement these policies into executable programs of a system is still not well known. Recent research has made some progress to improve the situations:

Jajodia [14] provided a logical language to write authorization policies and solve conflicts. In [14], policy conflicts are solved by including conflict solving rules in the policy set and detecting conflicts at the run time, instead of trying to avoid conflicts beforehand.

Policy management research at Imperial College [23, 20, 21, 17] is the most similar work we have found. This research analyzed policies for distributed systems from a birds-eye view. They analyzed conflicts based on the domain overlapping theory. These studies have resulted in the development of some policy tools for detecting conflicts before policies are enforced. We extend the research to the system admin and programmer levels.

From the knowledge representation and reasoning point of view there are basically two approaches to applying logic programming when writing policies. One approach is to develop an authorization specific language which is a subset/superset of general logic program. There is a lot of work on this track. For instances, Li et. al. have specified a logical language for access delegation [16]; Jajodia designed a general language for access control policies [14]; Woo and Lam have proposed a superset propositional language to describe authorization policies [27]; Grosz and Labrousse have designed a priority language to handle policies with conflicts [Gro97]. One advantage of this approach is that users can have a more authorization specific language to write their policies. However, most of them choose to translate the described policies into Prolog programs in order to utilize Prolog's well-established automatic inference engine for deduction. A pure authorization logic based inference engine has not been seen in the literature yet. Another approach is to directly apply logic programming to write authorization policies, including access control, authentication, delegation, policy merging, etc. With this approach, a sequence of policies is seen as a logic program. A typical example of this research is Bertino's work [3, 99], which uses LP to represent subjects (individual, group, or role), objects, and privilege hierarchy in authorization policies and to analyze policies purely through the general semantics of LP programs. Our work extends Bertino's work by developing a formal authorization policy model and connected the meaning of the programs with the model [11].

6 SUMMARY

In this paper, we have presented an *authorization policy engineering approach* to generating authorization policies. A hands-on experience on *Virtual* has demonstrated an exemplary engineering process in Prolog environment. Multiple authorization policies are described with rules and merged with logical connectives including conjunction and disjunction. These policies are separated into end user level policies, which focus on security models, and system admin policies, which deals with system configuration. These specified policies comprise the formal specification of *Virtual*. Formal verification has been shown to verify the correctness of the specification. Deployment strategies and integration are also briefly introduced. The following are the major lessons that we have learned from *Virtual*:

- *Prolog is a programming environment, not a theorem prover.* Prolog has only limited proving capability on finite systems based on a set of complete facts. In other words, in order to conduct rigorous formal verification, we need to transform the completion of the Prolog specifications into another format which is supported by other stronger theorem proving mechanisms. However, the limited proof support is still much better than the lack of mechanism in languages such as 'C'.
- *Authorization policies can be very specific.* It is well known that in a typical software engineering process, user requirements can be very specific. When trying to design a software system to implement the requirements, we usually need to creatively establish an abstract model, called high level design, in order to derive the entire software program by refining the model. During our policy engineering process, we found that most of the authorization policies except for those that have already been summarized by a few security models are very specific.
- *Policies (BL, DAC, and SA) specified in Virtual establish only the relationship between users and files.* Virtual policies cannot guarantee the security of a file system unless other policies controlling the process invocation, policy modification, delegation and revocation are properly established. However these policies are difficult to be derived from the Virtual policies and other available security models therefore have to be established based on experiences. In other words, the establishment of these policies is still ad hoc.

REFERENCES

- [1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," ACM Transactions on Programming Languages and Systems, 15(4): 706-734, Oct. 1993
- [2] Ross Anderson, "Security Engineering: A Guide to Building Dependable distributed Systems," John Wiley & Sons 2001, ISBN: 0471389226.
- [3] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo, "An authorization model and its formal semantics," In Proceedings of 5th European Symposium on Research in Computer Security, pages 127-143, September 1998.
- [4] E. Bertino, F. Buccafurri, E. Ferrari, and P. Rullo, "A Logical Framework for Reasoning on Data Access Control Policies," in Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW-12), pp. 175-189, IEEE Computer Society Press, Los Alamitos, CA, 1999.
- [5] D. E. Bell and L. J. LaPadula, "Secure Computer System: Unified exposition and multics interpretation," Tech. Report MTR-2997, The MITRE Corporation, Bedford, MA, July 1975.
- [6] Matt Blaze, Joan Feigenbaum, Martin Strauss, "Compliance Checking in the PolicyMaker Trust Management System," Financial Cryptography 1998, Anguila, 1998.
- [7] D. Brewer, and M. Nash, "The Chinese Wall Security Policy," Proc. Of IEEE Symposium on Security and Privacy, IEEE Computer Society Press, 1989, pp.206-214.
- [8] L. Cholvy, and F. Cuppens, "Analyzing Consistency of Security Policies." In Proceedings of the 1997 IEEE Symposium on Security and Privacy. Oakland, CA, USA: IEEE Press, 1997. p. 103-112.
<http://citeseer.nj.nec.com/laurence97analyzing.html>

- [9] D. D. Clark, and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," IEEE Security and Privacy Symposium, p. 184-194, April 1987.
- [10] J. Dai, and J. Alves-Foss, "Authorization Policy Engineering in the CBASS," *25th Anniversary Annual International Computer Software and Applications Conference (COMP.SAC 2001)* in Chicago, USA Oct 8-12, 2001.
- [11] J. Dai, "Logic Based Policy Engineering in Distributed Authorization," Dissertation, Computer Science Department, University of Idaho, Idaho, Nov. 2001.
- [12] J. Goguen, and J. Meseguer, "Security Policies and Security Models," Proc. 1982 IEEE Symposium on Security & Privacy, Oakland, CA., IEEE Computer Society, April 1982.
- [13] B. N. Grosf, and Y. Labrou, "An Approach to using XML and a Rule-based Content Language with an Agent Communication Language," IBM Research Report RC 21491, May 28 1999, available at: <http://www.research.ibm.com/rules/papers.html>
- [14] S. Jajodia, P. Samarati, and V. S. Subrahmanian, "A logic Language for Expressing Authorizations," *Proc. Of 1997 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, May 4-7, Oakland, California, page 31-42.
- [15] N. Li, J. Feigenbaum, and B. N. Grosf. "A Logic-Based Knowledge Representation for Authorization with Delegation (Extended Abstract)" In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, IEEE Computer Society Press, June 1999, pp. 162-174. Full paper available as IBM Research Report RC21492.
- [16] N. Li, B. Grosf, and J. Feigenbaum, "A Practically Implementable and Tractable Delegation Logic", In Proc. of the 21st IEEE Symposium on Security and Privacy, Oakland CA, May 2000, Available at: <http://cs1.cs.nyu.edu/ninghui/>
- [17] E. Lupu and M. Sloman, "*Conflict Analysis for Management Policies*," Fifth IFIP/IEEE International Symposium on Integrated Network Management IM'97, San-Diego, May 1997
- [18] D. McCullough, "Specifications for multi-level security and a hook-up property," proc. 1987 IEEE Symposium on Security and Privacy, pages 161-166, 1987.
- [19] D. McCullough, "Noninterference and the composability of security properties," *Proc. Of IEEE Symposium on Security and Privacy*, page 177-186, 1988.
- [20] J. D. Moffett, "Specification of Management Policies and Discretionary Access Control," In M. S. Sloman (Ed.), *Network and Distributed Systems Management* (pp. 455-479, Chapter 17). Addison-Wesley.
- [21] J. D. Moffett and M. S. Sloman, "Policy Conflict Analysis in Distributed System Management," *Ablex Publishing Journal of Organisational Computing*, vol. 4, no. 1, pp. 1-22, 1994.
- [22] R. Ortalo, "A Flexible Method for Information System Security Policy Specification," in Proc. of the 5th European Symposium on Research in Computer Security (Esorics'98), pages 68-84, LNCS 1495, Springer, September 1998
- [23] M. Sloman, "Policy Driven Management For Distributed Systems", *Plenum Press Journal of Network and Systems Management*, vol 2, no. 4, Dec. 1994, pp. 333-360
- [24] SRI, "The PVS Specification and Verification System – PVS Manuals," Computer Science Laboratory, SRI, 2001, available at: <http://pvs.csl.sri.com/manuals.html>
- [25] I. Sutherland, "A model of information," Proc. Ninth Nat'l Computer Security Conf.,Faithersburg, Md., 1986.
- [26] "UML Resource Page," Available at: <http://www.omg.org/technology/uml/>, April 26, 2001.
- [27] T. Woo, and S. S. Lam, "Authorization in Distributed System: A Formal Approach", *Proceedings of IEEE Symposium of Research in Security and Privacy*, Oakland, CA, May 1992.
- [28] W3C Architecture Domain, "Extensible Markup Language (XML)," available at: <http://www.w3.org/XML/>