Model Carrying Code with RBAC
Safe Execution of Untrusted Applications in RBAC Environment

Thesis to be submitted in Partial Fulfillment of the Requirements for the Award of the Degree

Master of Technology
In
Computer Science and Engineering

By
Rajib Ranjan Maiti
[06CS6018]

Under the guidance of

Prof. Pallab Dasgupta
&
Prof. Soumya Kanti Ghosh

Indian Institute of Technology, Kharagpur
May, 2008
Table of Content

1. Introduction
   1.1 Introduction to security policies
   1.2 Overview of access control policy
      1.2.1 Discretionary access control
      1.2.2 Mandatory access control
      1.2.3 Role Based access control
   1.3 Security Verification Challenges
   1.4 Problem Statement
   1.4 Summary of contribution

2. Previous works
   2.1 Overview of proof carrying code
   2.2 overview of model carrying code

3. Model carrying code with RBAC
   3.1 Security policy specification
      3.1.1 Representation of security policy with REE and EFSA
      3.1.2 Illustrative example
   3.2 Role change policy in RBAC
      3.2.1 Introduction to RBAC and role change
      3.2.2 Representation of role change policy through REE
      3.2.3 Representation of role change policy through EFSA
   3.3 Representation of program model

4. Verification of untrusted code model

5. Verification results

6. Conclusion

7. References
Chapter 1

Introduction

The term “information security” means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction [8]. The security from this view is very important because mostly of three reasons (A) integrity, which means guarding against improper information modification or destruction, and includes ensuring information of no repudiation and authenticity, (B) confidentiality, which means preserving authorized restrictions on access and disclosure, including means for protecting personal privacy and proprietary information, and (C) availability, which means ensuring timely and reliable access to and use of information. Access to such sensible information must be restricted to people who are authorized to access the information. These requirements lead us to have some rules for ensuring security requirements.

1.1 Introduction to security policy

A security policy defines rules for executions that, for one reason or other, has been consider in a particular way unacceptable [6]. For instance, policy may concern any or all of the following

- Access control, limits which operation can be performed on what object. Object here can be files, processes, memory modules, methods, functions (can be system call or programmer defined software critical functions) etc…
- Information flow, restricts which rules can be formed about some objects by observing system behavior
- Availability, limits the use of resources by denying others
Here we shall mainly concern about access control policies and described in the following section.

1.2 Overview of Access control policy

The computer programs, and in many cases the computers that process the information, must be authorized. This requires that mechanisms be in place to control the access to protected information. Access control policies are kind of that mechanism which specifies few rules to meet organizations access control requirements and security requirements. Access control policies can broadly be categorized into two parts – discretionary or non-discretionary. The three most widely recognized models are Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role Based Access Control (RBAC). MAC and RBAC are both non-discretionary.

1.2.1 Discretionary access control

Discretionary access control (DAC) is an access policy determined by the owner of an object. The owner decides who is allowed to access the object and what privileges they have. Two important concepts in DAC are

- **File and data ownership**: Every object in the system has an owner. In most DAC systems, each object's initial owner is the subject that caused it to be created. The access policy for an object is determined by its owner.
- **Access rights and permissions**: These are the controls that an owner can assign to other subjects for specific resources.

1.2.2 Mandatory access control

Mandatory access control (MAC) is an access policy determined by the system, not the owner. MAC is used in multilevel systems that process highly sensitive data, such as classified government and military information. A multilevel system is a single computer system that handles multiple classification levels between subjects and objects. Levels can be categorized into two parts.
- **Sensitivity labels:** In a MAC-based system, all subjects and objects must have labels assigned to them. A subject's sensitivity label specifies its level of trust. An object's sensitivity label specifies the level of trust required for access. In order to access a given object, the subject must have a sensitivity level equal to or higher than the requested object.

- **Data import and export:** Controlling the import of information from other systems and export to other systems (including printers) is a critical function of MAC-based systems, which must ensure that sensitivity labels are properly maintained and implemented so that sensitive information is appropriately protected at all times.

### 1.2.3 Role based access control

RBAC is an access policy determined by the system, not the owner [3]. RBAC is used in commercial applications and also in military systems, where multi-level security requirements may also exist. A role in RBAC can be viewed as a set of permissions. Basically three primary rules are defined for RBAC. Firstly, Role assignment - A subject can execute a transaction only if the subject has selected or been assigned a role. Secondly, Role authorization - A subject's active role must be authorized for the subject. With rule 1 above, this rule ensures that users can take on only roles for which they are authorized. Thirdly, Transaction authorization - A subject can execute a transaction only if the transaction is authorized for the subject's active role. With rules 1 and 2, this rule ensures that users can execute only transactions for which they are authorized. Most IT vendors offer RBAC in one or more products.

After representing a brief introduction of security and access control system, now question comes how to verify certain code (untrusted code) that it confronts the security requirements and access control policies.
1.3 Security verification challenges and existing solution

We can not always trust computer programs to be safe. Given that many programs are of questionable or unknown origin, for example, some online business transactions, mobile agents, games downloaded from shady websites, etc…, there are often reasons to believe that a piece of code may attempt to gain access to secure data or cause harm to the system.

Consider the following scenario: a computer user with some certain roll and permissions has obtained a program or a program component in binary form and wished to execute it without the risk of compromising a given security policy of that roll; for example, certain files may not be altered. Typically the user will not have the access to the source code of the program fragment and often will not trust the code producer, either due to fear of malicious intent or because of software bugs.

This is a major challenge in the verification field. Towards the solution of this problem few methodologies has been proposed – Proof Carrying Code (PCC) [2] and Model Carrying Code (MCC) [1] are major in this part. Both of these approaches works in a flat system where user is not a part of level or hierarchy like role. In such systems a user can have multiple roles and depending upon the requirement, the user can be assigned some certain roles to work with.

1.4 Problem Statement

Problem of verifying an untrusted code against user's security policy with the change of role of the user has got a severe challenge. Given a role change policy and a set of access control policies associated with each role, how to verify an untrusted code so as to ensure the code wouldn't hamper security. Our model carrying code with RBAC framework is given in figure 1.
In this paper, we are going to represent the verifier and its working. The use of security behavior specification in the model enables us to decompose the verification into two parts: policy satisfaction and model enforcement (section 1.4) [9]. Firstly, the policy satisfaction can be mapped to the problem of model checking by representing the policy through an automata model. So, once we represent the policy and model, rest of the work is just to find product automaton, \( B[P] \times B[RP] \times B[M] \). Then the condition \( B[M] \cap B[P] \cap B[RP] = \phi \) will be sufficient to say the model is safe.

Secondly, the model enforcement is to be done. With our approach we found model enforcement is more convenient and easier rather than policy enforcement. The policy automatons are large non-deterministic automata where as model may be larger but they are completely deterministic. So. With problem we have considered model enforcement. Sometimes it is obvious that rather than policy to be changed the model or in essence the code should not be executed further. In the overall verification process, the difficulty is in representing the security policy, role change policy and the program model. Representation of each of these has been described in detail in next chapter. Representation is basically in the form of an automata model. The model safety part of this project has been done by my project partner, so that is out of the scope of this paper.

1.5 Summary of contribution

The important issue in our approach (see Figure 1.1) is the introduction of a model having program’s behavioral aspects. These models capture security-related properties of the code, but do not talk about the functional correctness of the program. The model is stated in terms of security-relevant operations made by the code, the arguments of these operations, and the sequencing relationships among them. In our implementation, we have considered both type of operations correspond to system calls and some application critical high level user defined functions (for example role change in RBAC environment).
Our concern is verification only. Verification now becomes a model checking problem.

The user or consumer of the code receives both the model and the program from the producer. The consumer wants to be assured that the code will satisfy a security policy selected by him/her. The use of a security behavior model enables us to decompose this verification argument into two parts [1]:

Say, the application is denoted by $A$, security policy by $P$, role change policy denoted by $RP$ and $B$ is a function that maps a policy or model to the set of behaviors satisfied by the policy or model then,

- **Policy satisfaction:** check whether the model satisfies the security policy and role change policy, i.e., the behaviors captured by the model are a subset of the behaviors allowed by the policy. This can be expressed symbolically as
\[ B[M] \subseteq B[P] \times B[RP] \]

- **model safety**: check if the model captures a safe approximation of program behavior — more precisely, that any behavior exhibited by the program is captured by the model: 
  \[ B[M] \cap B[P] \cap B[RP] = \phi \]

\[ B[A] \subseteq B[M] \]

Together these two implies that the application satisfies the security policy and role change policy i.e. 
\[ B[A] \subseteq B[P] \times B[RP] \]. In order to be confirmed that that an untrusted code is safe to execute, the both of these verifications are mandatory. Because of the way the model is generated, the model safety is a necessary step in our approach even after model verification.
Chapter 2

Previous Works

2.1 Proof carrying code

Proof-Carrying Code (PCC) and related approaches involve associating safety information in the form of a certificate (or a proof) to programs. Proof is created at compile time by the code supplier, packaged with untrusted code. Code consumer can then run a verifier which, by a straightforward inspection of code and certificate, verifies validity of certificate and thus compliance with safety policy. The key benefit of this "certificate-based" approach to mobile code safety is that the burden of ensuring compliance with the desired safety policy is shifted from the consumer to the supplier which in other way means exploring security to the world of producer. PCC framework is given below in figure 2.1.

Figure 2.1: Proof Carrying Code Framework
Proof carrying code framework certifies a code with a condensed formal proof of a desired property. The proof generated by the certifying compiler is checked with consumer’s proof checker before installation/execution. Proof generation for a code is a complex problem whereas checking the proof is much easier. Proof carrying code works without considering the complexity of the network through which the code and proof travels. Proof carrying code is not hampered form being copied the proof for the code illegally. Basically, a certifying compiler uses types and other high-level source information to create the necessary proof to accompany the corresponding machine code.

Shortcomings: the entire code verification overhead is handed over to the producer of the code which means producer has to know about the consumer security related information. The fact that the certified code is no longer machine independent, applicability of the code is limited. Another factor is the space is required for the proof is also a bottleneck.

2.2 Model carrying code

On the other hand, Model Carrying Code (MCC) has got the idea of making a model of the code; producer of the code provides code with this model. The model captures security related behavior of the code. The security behavior is specified in terms of critical system calls. The model bridges the semantic gap between low-level binary code and high-level security policies (of consumer). The producer has no need make any assumption about consumer’s security policies. In general, models being much simpler than programs, automation of consistency checking is easier (between consumer policy and the model). Model carrying code framework is given below (figure 2.2).

The verifier on the consumer side checks the model against the high level security policies and reports if any violation can happen. By looking at the report, consumer can regenerate his policy or discard the code execution. If the verifier
certifies the model then code may run on. Model enforcement component of the MCC framework does an exciting work of inspecting the system calls and its argument values while running the code to see whether any violation can be attempted. If it does so a model enforcement rather than policy enforcement is done.

Shortcomings: The verification of the model gets challenge when the model needs to be checked not only with the policies but with the possibility of person’s role change also.

Figure 2.2: Model Carrying Code Framework
Chapter 3

Model Carrying Code with RBAC

In addition to the MCC verifier another component called role change policy is now considered for verification of the model and generating the comprehensive conflict feedback report. We begin our description of the technique of verification with an overview of security policy language and role change policy.

In this thesis we mainly discuss the verification part in the consumer side. This involves majority of the complexity. Verification of the program model involves representing the different policies and the program model itself. Before we start discussion on representation we should look into few definitions.

3.1 Security policy specification

Applicability of any security policy depends on whether the policy is enforceable and what cost it may incur in terms of how much overhead is associated with it. Enforcement mechanism which works by monitoring execution steps of a system, sometimes referred as target, and terminating the target's execution if it is about to violate the policy being enforced is our concern [11,12,7]. Our target may be objects, modules, processes, subsystems, or entire system; the execution steps monitored may range from fine-grained actions (like memory accesses) to higher level operations (like method calls) to operations that change the security-configuration and there by restricts subsequent execution [14,13].

Mechanisms that use more information than would be available only from observing the steps of a target execution are not execution monitoring. So, execution monitoring mechanisms does not have sufficient information to know the steps before hand the target machine will take in future or to know alternative possible execution paths. That’s why the compilers of theorem-provers which
analyses the static information about all its possible execution are not execution monitoring and hence are not applicable for untrusted code execution [2]. Hence with our approach the verification of an untrusted code is concerned only with enforceable security policies.

Model carrying code with role based access control relies on execution monitoring for model enforcement, and hence only enforceable security policies [6, 9] are of interest. Such policies constitute safety properties. Common examples of enforceable policies include access-control and resource-usage policies. Java 2 security model [7] supports standard access-control policies, but can handle applications that consist of code from multiple producers. Naccio [11] supports specification of both access control and resource usage policies. The security automaton formalism [7,4] can support safety properties that involve sequencing relationships between operations. However, this formalism does not provide the ability to remember argument values such as file descriptors for subsequent comparisons with arguments of other operations. Our policy language is based on extended finite state automata (EFSA) that extend standard FSA by incorporating a finite set of state variables to remember argument values [5,11]. For instance, we can associate a write operation with the file name involved in writing by recording the return value from an open operation (a file descriptor), and comparing it with the argument of the write operation. Below, we describe this policy language and illustrate it through examples.

3.1.1 Security policy with REE and its equivalent EFSA

We model behaviors in terms of externally observable events. In modern operating systems, security related actions of programs need to be ultimately effected via system calls. Hence system calls constitute the event alphabet in our policies. Naturally, it is possible to define behaviors in terms of operations other than system calls, e.g., arbitrary function calls. Higher level policies can often be stated more easily and accurately in terms of function calls. For instance, a policy
that permits a program to make name server queries can be stated as “program is allowed to use the function gethostbyname” rather than the more complicated (and less precise) version “program allowed to connect to IP address on port 53.”

We use the term *history* to refer to a sequence of events. A history includes events as well as their arguments. A *trace* is a history observed during a single execution of a program. The behavior of a program, denoted by $B[A]$, is defined to be the set of all traces that may be produced using any execution of $A$.

Policies capture properties of traces. They are expressed using EFSA. Like security automata, EFSA express negations of policies, i.e., they accept traces that violate the intended policy. The state of an EFSA is characterized by its *control state* (the same notion as the “state” of an FSA), plus the values of (a finite set of) state variables. These state variables can take values from possibly infinite domains, such as integers and strings. Each transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. For a transition to be taken, the associated event must occur and the enabling conditions must hold. When the transition is taken, the assignments associated with the transition are performed. Events in a more formal way can be defined as follows.

Event (reproduced from [1]): Event may be further classified as follows:
- **Primitive events**: There are two primitive events associated with each system call, one corresponding to the system call invocation and the other to its exit. The invocation event has the same name and arguments as the system call, while the return event has an “exit” appended to its name. The arguments of the entry event include all of the arguments at the point of call. The arguments to an exit event include all of the arguments at the point of return, plus the value of the return code from the system call.
- **Abstract events**: Abstract events can be used to denote classes of primitive events, e.g., we may define FileModificationOps as an event that
corresponds to a set of events that modify files. More generally, abstract events may be defined using the notation \( \text{event(args)=pat} \), where \( \text{event} \) denotes the abstract event name, and \( \text{pat} \) is further defined below.

Patterns [1]: The simplest patterns, called \textit{primitive} patterns, are of the form 
\( \text{e(x1, ..., xn)|cond/asz} \), where \( \text{cond} \) is a boolean-valued expression on the event arguments \( x_1, ..., x_n \) and state variables, and \( \text{asz} \) contains zero or more assignments to state variables.

Example: say, one of the high level function call \textit{roleRequest} with its arguments and enabling conditions and some assignment we have. 
\text{roleRequest}(id,role) | (sv = OU) 
Here, \( e \) is \textit{roleRequest}, \((x1,x2)=(id,role)\) and assignment, \( \text{asg} \) is “sv=OU”.

The scope of event arguments is limited to the primitive pattern within which it occurs, that is why we need to have some assignment statement whose value would be used in future.

\textit{Compound patterns} are obtained by composing primitive patterns using \textit{sequencing operators} similar to those in regular expressions. The meaning of patterns is best explained by the following definition of what it means for a history \( H \) to satisfy a pattern:

- \textit{event occurrence}: \( e(x_1, ..., x_n)|\text{cond} \) is satisfied by the event history consisting of the single event \( e(v_1, ..., v_n) \), if \( \text{cond} \) evaluates to true when variables \( x_1, ..., x_n \) are replaced by the values \( v_1, ..., v_n \).
- \textit{alternation}: \( \text{pat1}||\text{pat2} \) is satisfied by \( H \) if either \( \text{pat1} \) or \( \text{pat2} \) is satisfied by \( H \).
- \textit{sequencing}: \( \text{pat1} \cdot\text{pat2} \) is satisfied by an event history \( H \) of the form \( H_1H_2 \) provided \( H_1 \) satisfies \( \text{pat1} \) and \( H_2 \) satisfies \( \text{pat2} \).
- \textit{repetition}: \( \text{pat}^* \) is satisfied by \( H \) if and only if \( H \) is empty, or is of the form \( H_1H_2 \) where \( H_1 \) satisfies \( \text{pat} \) and \( H_2 \) satisfies \( \text{pat}^* \).
negation: !pat is satisfied by H if and only if pat is not satisfied by H.

(Use of negation, is not permitted in BMSL if pat involves sequencing)

The notion of satisfaction extends in the obvious way when state variables are included, and the details can be found in [7].

Example: with the help of basic events like open, close, other, any, etc… we can create compound patterns like (open() || close() || other)*, which means a transition can be taken if either of these events has occurs. The meaning of any and other must be understood clearly. Say we have a pattern like any*.open(), then there will be two transitions if there is a match found with open, where as other*.open() will produce a single transition with the event open() in the match case. So any is non-deterministic in nature where as other is deterministic.

3.1.2 Regular expressions over events and policy specification

Figure 3.1 illustrates a simple policy example using REE. Note that, the special event any stands for any event, while other stands for an event other than those matching the rest of the transitions on the same state. Since a history matches an REE whenever a prefix of H satisfies the REE, the REE patterns do not need to have the any transitions that occur in the final state of the EFSA policies.

The policy of figure 3.1 describes a security policy that prevents a sequence of two file write operations, where files are belongs to administrative file list. Note that the operator || is overloaded so that it can represent pattern alteration as well as the Boolean-or operation. If any of the prohibited operations are performed by a program, then the automaton makes a sequence of transitions to reach final state (marked with a double circle) from the start state (marked with a “>” symbol). For any other operations, the transition marked “other” is taken, i.e., the EFSA stays either in the start state or in the intermediate state.
Regular Expressions over Events [7]: The alphabet of REEs is denoted by $\Sigma$. A member of the alphabet $\Sigma$ is an event of the form $e(x_1, x_2, \ldots, x_n)$ where, $e$ is the name of the event and $x_1, x_2, \ldots, x_n$ are variables which denote the arguments of the event. The domains of these event arguments are $D_1, D_2, \ldots, D_n$. These domains can be unbounded. State variables hold values from these domains. We denote the set of state variables used in an REE as $V$.

Definition: A regular expression over events on the alphabet $\Sigma$ is defined as follows:

- $\varepsilon$ is an REE representing an empty history.
- A $\text{PrimPat}$ is an REE, where $\text{PrimPat}$ is defined by the productions:

  $$
  \text{PrimPat} \to ["!"]\text{EventSet}
  \text{EventSet} \to \text{EventCond}["\mid"]\text{EventSet}
  \text{EventCond} \to e(x_1, \ldots, x_n)["\mid"]C[/A]]
  $$

Here, $e(x_1, \ldots, x_n) \in \Sigma$.

Notation | Meaning
--- | ---
$R$ | REEs are denoted using upper case alphabets in math font
$e$ | Events are represented in lower case italics.
$A$ | Represents a sequence of assignments of the form $v_i \leftarrow expr_1, \ldots, v_n \leftarrow expr_n$ where $v_1, \ldots, v_n \in V$ and each $expr_i$, where $1 \leq i \leq n$, is a simple arithmetic expression over constants, event arguments and state variables.
$C$ | Represents boolean-valued expressions involving simple comparison and arithmetic operator on event arguments, $x_1, \ldots, x_n$ and the state variables over $V$.
“literals” | Literals are enclosed within double-quotes whenever they may be confused with the meta-symbols in the grammar such as alternation ($\mid$).
Now
• If R and Q are REEs, then so are:
  – (R)
  – R · Q
  – R*
  – R||Q

Only those expressions that can be produced by the rules above are REEs.

Let us take an example security statement “prevent write to all files but any file from administrative right can only be read and at most two files can be opened in this mode”. This statement can be represented with REE as follows

\[
\text{List admFiles} = \{"/ \text{etc} / f1", "/ \text{etc} / g1"\}
\]

\[
\text{other}^* . \text{open}(f, m) | (f \in \text{admFiles} \& \& m \neq O_{RDONLY}).
\]

\[
\text{other}^* . \text{open}(f, m) | (f \in \text{admFiles} \& \& m \neq O_{RDONLY}).
\]

\[
\text{any}^*
\]

Figure 3.1: REE of a policy

In this policy, the event is opening an administrative file, system call is “open” and the arguments are: f is the file name of administrative right, m is the mode of opening the file. The policy says any application which needs to open administrative file, it can open a single file only for modification in that session. Once it reaches the final state the security policy automata will raise a violation.

### 3.1.2 Representing Security Policy through EFSA

Extended Finite State Automata [Reproduced from 7]: An EFA is a septuplet:

\[(Q, q_0, F, V, Env, \Sigma, \delta)\]

where:
- \(Q\) is a finite set of states.
- \(q_0 \in Q\) is the start state.
- \(F \subset Q\) is the set of final states.
• \( V \) is the finite set of state variables.
• \( Env \) is the set of all possible environments, representing distinct valuations of the variables in \( V \).
• \( \Sigma \) is the input alphabet, and incorporates event names as well as arguments.
• \( \delta : Q \times \Sigma \times Env \to 2^{(Q \times Env)} \), is the transition function.

\( Q, q_0 \) and \( F \) are identical to their counterparts in FSA. However unlike FSAs, \( \Sigma \) is an alphabet not only on events but also on their arguments. FSAs do not have \( V \) and \( Env \). The transition relation \( \delta \) differs from the transition relation of a FSA, in that the state of an EFA is given by its current control state \( q \in Q \) as well as the valuations of state variables given by the current environment \( E \in Env \). For this reason, occurrences of \( Q \) in FSA definitions are replaced by \( Q \times Env \).

The reflexive and transitive closure \( \delta^* \) of \( \delta \) are defined as follows:
\[
\delta^*(q, \epsilon, E) = \{(q, E)\}
\]
\[
\delta^*(q, e_1 e_2 \ldots e_n, E) = \bigcup_{(q', E') \in (q, e_1, E)} \delta^*(q', e_2 \ldots e_n, E')
\]
Here \( e_1 e_2 \ldots e_n \) is a history.

An EFA accepts an event history \( H \) if and only if there exists a \( q \in F \) and an environment \( E \) such that \((q, E) \in \delta^*(q_0, H, E_\phi)\).

EFSA representation of the same is shown in figure 3.1 corresponding to the REE of figure 3.2. Here \( Q \) is the set \{s₁, s₂, s₃\}, \( q_0 \) is \( s_1 \), \( F \) is the set \{s₃\}, no state variable has been used here, \( f \) can take any name in the form of a finite length string that is the actual value of the argument of the function call which forms the set \( Env \), \( \Sigma \) is \( \text{open()} \) system call and its arguments. A transition \( \delta : s_1 \to s_2 \) is possible only when \text{open()} call in model with its argument values will satisfy the enabling condition.
3.2 Role change policy in Role based access control

Before we discuss the issues related to the representation of role change, let’s have a brief introduction of role based access control. Role based access control [3] allows some certain permissions to a role to perform an operation on an object, not to users directly. Users are assigned a role depending on their job functionality in the organization.

3.2.1 Introduction to role based access control and role change

Role-based access control (RBAC) features have been implemented in database management, security management, and network operating system applications. The concept of role-based access control (RBAC) began with multi-user and multi-application on-line systems. The central idea of RBAC is that permissions
to access some resource are associated with roles, and users are assigned to appropriate roles to which they are responsible. This greatly simplifies the management of resource access permissions. Roles are created for the various job functions in an organization and users are assigned roles based on their responsibilities and qualifications. Depending on the need users can even easily be reassigned from one role to another. Roles can be granted new permissions as new applications and systems are incorporated, and permissions can be revoked from role’s requirements.

For example, a role can represent the ability to do specific tasks, such as a physician or a pharmacist. A role can embody authority and responsibility, e.g., project supervisor or administrator. Authority and responsibility are distinct from eligibility. Say, Jane Doe may be competent to head several departments, but is assigned to head one of them. Roles can reflect specific duty assignments that are rotated through multiple users, e.g., a duty physician or shift manager. RBAC models and implementations are able to conveniently accommodate all of these manifestations of the role concept.

Depending on the organization’s requirements and imposing the condition and constraints on the several roles, the RBAC model comes in four flavors: RBAC0, RBAC1, RBAC2 and RBAC4. Here we shall discuss the base model i.e. RBAC0, for better understanding of the RBAC models refer [3].

**Definition: Base RBAC model, RBAC0 (reproduced from [3])**

The RBAC model has the following components:

- U, R, P, and S (users, roles, permissions and sessions respectively)
- \( PA \subseteq P \times R \), a many-to-many permission to role assignment relation,
- \( UA \subseteq U \times R \), a many-to-many user to role assignment relation,
- User: \( S \rightarrow U \), a function mapping each session \( s \) to the single user \( user(s) \) (constant for the session’s lifetime), and
• roles: \( S \rightarrow 2^R \), a function mapping each session \( s_i \) to a set of roles

\[
\text{roles}(s_i) \subseteq \{r | (\text{user}(s_i), r) \in UA\},
\]

which can change with time, and the session \( s_i \) has the permissions \( \bigcup_{r \in \text{roles}(s_i)} \{p | (p, r) \in PA\} \).

Let's have an example to illustrate the concept. Say, we have five users: \( f_1...f_5 \), two roles: super user and student (as ordinary user), permissions: reading administrative files, modifying administrative files, connecting to the internet. A possible permission assignment can be

- super user \( \rightarrow \) \{modifying administrative files, connecting to internet\},
- student \( \rightarrow \) \{reading administrative files, connecting to internet\},

and a possible user assignment could be \( \{f_1,f_3,f_5\} \rightarrow \text{student} \) and \( \{f_2,f_4,f_3\} \rightarrow \text{super user} \). Sessions are under the control of individual users. As far the model is concerned, a user can create a session and choose to activate some subset of the user's roles. Roles active in a session can be changed at the user's discretion. The session terminates at the user's initiative.

RBAC directly supports three well-known security principles: least privilege, separation of duties, and data abstraction. Least privilege is supported because RBAC can be configured so only those permissions required for the tasks conducted by members of the role are assigned to the role. Separation of duties is achieved by ensuring that mutually exclusive roles must be invoked to complete a sensitive task, such as requiring an accounting clerk and account manager to participate in issuing a check. Data abstraction is supported by means of abstract permissions, e.g., credit and debit for an account object, rather than the read, write and execute permissions, typically provided by the operating system. However, RBAC cannot enforce application of these principles. The security officer could configure RBAC, so it violates these principles. Also, the degree to which data abstraction is supported will be determined by the implementation details.
Drawbacks: Major draw back of RBAC model is it can not check whether a sequence of actions is violating any access restrictions. By sequence we mean order in which the objects can be accessed or not. For example, after reading some secure file the user can not open a file in write mode. Taking this drawback into account we shall propose a new security policy called role change policy which will be responsible of ensuring a sequence to be safety.

For example, a role change is not possible of a user after he/she has done some certain sequence of actions. Such a role change policy can be like after closing all the files during a role, role change is possible. Defining such policies in terms of EFSA is non-trivial work; need to have complete domain knowledge of the system where this role change is going to be effective. In the following section we are representing such a policy through REE and its equivalent EFSA.

3.2.2 Representing Role Change Policy through REE

Let’s consider a role change policy where we say that role change is possible only when the user have closed all the files he/she has opened in current role (in figure 3.3). In order to specify the policy, here we have consider some high level application critical functions, e.g. roleRequest(). fileList is a List which can contain some generic items like a file name, some integer or some other values of a system call argument. OU or SU are the roles for example in the domain RBAC which a user can acquire. We needed to have a state variable (sv) here to remember the role of the user for future purpose and has been used by other states. f and m are some file name and its mode with which the file has been opened and other symbols has the same meaning as in the access policy described in previous section. add(), remove() are typical list operations.
List fileList = {}
((roleRequest(OU) | sv = OU) || (roleRequest(SU) | sv = SU))
(((open(f, m) | add(f, fileList)) || (close(f) | remove(f, fileList)) || other)*. roleRequest(x) | x ≠ sv & & fileList is "not empty")

Figure 3.3: REE for role change in RBAC

3.2.3 Representing Role Change Policy through EFSA

Representation of role change policy (in figure. 3.4 corresponding to figure. 3.3) automata is similar to that of policy automata. The state set $Q = \{S_1, S_2, S_3, S_4\}$, starting state $q_0$ is $S_1$ shown by ‘V’ and the final state set $F$ is $\{S_4\}$ shown by double concentric circle.

Figure 3.4: An Example of Role Change Policy
3.3 Representation of Program model

Let us assume we have already extracted model out of a program and that is represented through EFSA as given below and see how this model can be verified (Figure 3.5). As stated earlier the program model must capture security related function calls only, preferably. Generally the program model is generated through test cases which leave the scope of some loop wholes that is the model may ignore some hidden function calls which may cause security violation. To ensure that we have model safety part, similar to that of the model carrying code model safety i.e. $B[A] \subseteq B[M]$. Ensurity of model safety has not been discussed in this paper. Program starts with state $q1$ and final state is $q6$. For illustration purpose only very simply model has been considered.

![Figure 3.5: A Program Model capturing security behavior](image)
Chapter 4

Verification of untrusted code model

Verification is concerned with determining whether or not a model \(M\) satisfies a security policy \(P\) and role change policy \(R\). Formally, we need to check whether behaviors captured by \(M\) is a subset of behaviors permitted by the policy \(P\) and \(R\) — \(B[M] \subseteq B[\overline{P}] \times B[\overline{R}]\) where \(M\), \(B\), \(R\) and \(P\) were introduced earlier. Noting that the policy automaton actually represents the negation \(\overline{P}\) of \(P\) and \(\overline{R}\) of \(R\), we simply need to determine if \(B[M] \times B[\overline{P}] \times B[\overline{R}] = \phi\). Thus, our verification approach is to build the product automaton \(M \times \overline{P} \times \overline{R}\), which will accept the intersection of the behaviors accepted by \(M\), \(\overline{R}\) and \(\overline{P}\). If there are feasible paths in this product automaton that lead to final (i.e., violating) states of \(P\) or \(R\), then the policy is violated and \(M \times \overline{P} \times \overline{R}\) is a representation of all such violations.

All common operations, such as computing the product of two automata and checking it for reachability, have well-known solutions in the case of FSA, but become complex in the case of EFSA due to the presence of infinite domain variables. We begin by computing the EFSA product in much the same way as an FSA product construction. Specifically, the product automaton \(MPR = M \times \overline{P} \times \overline{R}\) is constructed as follows:

- The state variable set of \(MPR\) is the union of the state variables of \(M\), \(P\) and \(R\). The start state of \(MPR\) is a tuple \((m_0, p_0, r_0)\), where \(m_0\), \(p_0\), and \(r_0\) are the start states of \(M\) and \(P\) and \(R\), respectively. Similarly, the final state set is
- \(F_{MPR} \subseteq F_M \times F_P \times F_R\), where \(F_M\) is the set of all states in \(M\), \(F_P\) denotes the set of final states in \(P\) and \(F_R\) denotes all the final states in \(R\).
Whenever there exists a transition from a state $s$ to $s'$ in $M$ on event $e$ with condition $C_1$ and assignment $A_1$, a transition from $p$ to $p'$ in $P$ on the same event $e$ with condition $C_2$ and assignment $A_2$ and a transition from $r$ to $r'$ in $R$ on the same event $e$ with condition $C_3$ and assignment $A_3$, then (and only then) there is a transition from $(s,p,r)$ to $(s', p', r')$ in $MPR$ on condition $C_1 \cap C_2 \cap C_3$ with assignment $A_1 \cup A_2 \cup A_3$.

A transition in the product automaton is said to be enabled only when the associated condition $C_1 \cap C_2 \cap C_3$ is satisfiable. Given that our EFSA is defined over infinite-domain variables representing strings and integers, the problem of determining satisfiability of arbitrary arithmetic constraints appearing as enabling conditions of transitions is, in general, undecidable. We focus, therefore, on a tractable subset of constraints over infinite-domain variables; Specifically equality (=) and inequality (\neq) relationships between the variables.

An example illustrating the above concept is given below. Assume the same model and the following access policies associated as follows. Policy of figure.2 is associated with role OU and policy of figure.4.1 is associated with SU.

\[
\text{List UserList} = \{ \};
\]

![Diagram](image_url)

Figure 4.1: Policy regarding deleting a user account
The product automaton is represented in figure 8. Let us take an example transition. Say we are at state \( \{4,2,1\} \): first numeral represent state number in program model, second numeral represents state number in role change policy and third numeral represents a state in a policy of the role. At this state of product model, we have three transitions coming out of program model state-4. So, corresponding to each transition of the program model, we look for a match in role change policy as well as in policy automaton. This way we have a transition \( \{4,2,1\}\rightarrow\{3,2,2\} \) with system call “\(\text{open}(\ldots)\)”, a transition \( \{4,2,1\}\rightarrow\{5,3,1\} \) with system call “\(\text{requestRole}\)“ and a transition \( \{4,2,1\}\rightarrow\{6,2,1\} \) with “\(\text{close}(\ldots)\)“. This procedure is repeated until no more new state can be added to the product model. When a state containing final state of either role or access policy is called a final state in product model with indeed a violation state.

The product state starts with the start states of model’s start state, \( q1 \), role policy start state \( r2 \) and the start state \( p1 \) of access policy of role \( r2 \) that is role of OU. Each double circled state that we have reached the final state of the model and underlined state represents a violation for example \( \{q4,r2,p3\} \).
Figure 4.2: Product EFSA
Chapter 5
Verification results

Any violation in the product automaton is indicated by final state which contains any of the final states either of role change or of policy automaton. In the above example, we have two violation paths (figure 5.1, 5.2).

Violation of policies in our approach can occur in two ways. Firstly, violation is possible from access policy in the same policy, i.e. the running application trying to access some resource for which the user’s current role have no permission, shown in figure 5.1.
For example, without closing a file opened in transition \[ \{q_4, r_2, p_1\} \rightarrow \{q_3, r_2, p_2\} \] the application is trying to change user’s role from state \( \{q_4, r_2, p_2\} \) shown in figure 8 Secondly, violation is possible from role policy, i.e. the running application trying to change user’s role which is not permitted by the role change policy. For example, without closing a file opened in transition \[ \{q_4, r_2, p_1\} \rightarrow \{q_3, r_2, p_2\} \] the application is trying to change user’s role from state \( \{q_4, r_2, p_2\} \) shown in figure 5.2.

![Diagram](image)

**Figure 5.2: Role policy violation Path**

Hence forth, we have proposed a procedure of verification of a program model with respect to the RBAC model. A possible execution sequence of an application intended for RBAC environment across the role can be verified with our approach.

Our verification procedure has been done by generating some random program model. Random in the sense that the state and transition s has been generated randomly out of a pull of system calls and high level application critical function of about hundred entries. Policy selection has also been done randomly. We had to
take a special care in order to check whether the generated program model is connected.

The verifier size in terms of lines of code is 2kLOC. A tentative execution time when the verifier runs with some certain number of states and transitions and with few roles is given in the table below.

<table>
<thead>
<tr>
<th>#states</th>
<th>#transitions</th>
<th>#times the role change occurs</th>
<th>Time (msec)</th>
<th>Space(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>400</td>
<td>5</td>
<td>1.5</td>
<td>0.5</td>
</tr>
<tr>
<td>250</td>
<td>500</td>
<td>8</td>
<td>1.8</td>
<td>0.7</td>
</tr>
<tr>
<td>60</td>
<td>150</td>
<td>3</td>
<td>1.4</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Figure 5.1: Results on verification of model
Chapter 6
Conclusions

We have discussed the information security issues and its importance for an enterprise’s smooth working. We also have discussed who all are the people are allowed to access an organization’s critical and public information and how security threats comes into the picture. Towards the solution of this problem various solution techniques have been proposed such as PCC where code comes with a certificate, MCC where a model describing high level security related information of the code comes with the binary itself etc... We have shown how these methodologies are unable to addresses certain problem in securing a system of certain framework like role based access control. After giving the problem statement we have proposed an efficient solution of it.

With role based access control, a user can have multiple roles and a set of permissions corresponding to each and every individual role. In our technique, we have devised policies to change role safely in RBAC together with security policies for individual role. Now verification of incoming model is not only with the current role's security policies but also with role changing policy so as to how the code can be run safely. This generates a nice comprehensive report on security policy violation and role change policy violation. We have discussed about the implementation of this technique and results out of the verification.
Chapter 7

References


