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

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

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
Content

1. Introduction
2. Problem Statement
3. Summary of Contribution
4. Verification of MCC in RBAC
5. Conclusion
6. References
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 so important because of mostly 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.

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.

RBAC is an access policy determined by the system, not the owner. 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.

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.

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.

On the other hand, Model Carrying Code (MCC) - code producer provides code, plus a high-level model of its security related behavior. The model bridges the semantic gap between low-level binary code and high-level security policies (of consumer). The producer need not guess consumer security policies. In general, models being much simpler than programs, automation of consistency checking is feasible (between consumer policy and the model). But the verification of the model gets challenge when the model needs to be checked not only with the policies but with persons role change also. So the problem statement follows in the following section.

2 Problem Statements

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 below in figure 1 (reproduced from [1]).

![Figure 1: MCC in an RBAC environment](image)

In this paper, we are going to represent the verifier and its working. The use of security behavior model enables us to decompose the verification into two parts policy satisfaction and model enforcement [1], can be described in the following way:

Say, $B$ is a function which maps the Model/Policy/Role change/Application into its actual behavior. Then
- Policy Satisfaction: $B[M] \subseteq B[P] \times B[RC]$ → The behavior described in Model M has to be a subset of the behavior allowed by policy P and role change policy RC.
- Model Safety: $B[A] \subseteq B[M]$ → The actual behavior of application A has to be a subset of the behavior described in model M.

If both the requirements are fulfilled, the wanted effect is reached: the application's behavior is allowed by the policy P and role change: $B[P] \times B[RC]$.

### 3 Summary of Contribution

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 Representation of Security Policy

#### 3.1.1 Representing Security Policy through REE

**Event [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 $\text{event}(\text{args})=\text{pat}$, where $\text{event}$ denotes the abstract event name, and $\text{pat}$ is further defined below.

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

The scope of event arguments is limited to the primitive pattern within which it occurs. Compound patterns are obtained by composing primitive patterns using 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:

- **event occurrence:** $e(x_1, \ldots, x_n)|\text{cond}$ 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 \( v_1, ..., v_n \).

- **alternation**: \( \text{pat}_1 \mathbin{|} \text{pat}_2 \) is satisfied by \( H \) if either \( \text{pat}_1 \) or \( \text{pat}_2 \) is satisfied by \( H \).
- **sequencing**: \( \text{pat}_1 \cdot \text{pat}_2 \) is satisfied by an event history \( H \) of the form \( H_1 H_2 \) provided \( H_1 \) satisfies \( \text{pat}_1 \) and \( H_2 \) satisfies \( \text{pat}_2 \).
- **repetition**: \( \text{pat}^* \) is satisfied by \( H \) if and only if \( H \) is empty, or is of the form \( H_1 H_2 \) where \( H_1 \) satisfies \( \text{pat} \) and \( H_2 \) satisfies \( \text{pat}^* \).
- **negation**: \( \neg \text{pat} \) is satisfied by \( H \) if and only if \( \text{pat} \) is not satisfied by \( H \). (Use of negation is not permitted in BMSL if \( \text{pat} \) involves sequencing or repetition.)

**Regular Expressions over Events** [7]: The alphabet of REEs is denoted by \( \sum \). A member of the alphabet \( \sum \) is an event of the form \( e(x_1, x_2, ..., x_n) \) where, \( e \) is the name of the event and \( x_1, x_2, ..., x_n \) are variables which denote the arguments of the event. The domains of these event arguments are \( D_1, D_2, ..., D_n \). 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 \( \sum \) 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:

\[
\begin{align*}
\text{PrimPat} & \rightarrow ["!"]\text{EventSet} \\
\text{EventSet} & \rightarrow \text{EventCond}["||"]\text{EventSet} \\
\text{EventCond} & \rightarrow e(x_1, ..., x_n)["|"]C/[A]
\end{align*}
\]

Here, \( e(x_1, ..., x_n) \in \sum \).

**Notation**

| \( R \) | 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 \( v_1 \leftarrow expr_1, ..., v_n \leftarrow expr_n \) where \( v_1, ..., v_n \in V \) and each \( expr_i \) is a simple arithmetic expression over constants, event arguments and state variables. |
| \( C \) | Represents a boolean-valued expression involving simple comparison and arithmetic operators on event arguments, \( x_1, ..., x_n \) 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 ( | ).

[] Represents optional symbol.

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

\[
\text{List adminFiles-} \{\"/etc/f1\"{-}/etc/g1\}\n\text{other*.open(f,mode)}\{\text{admFiles}\}|\{\text{model=O.RDONLY}\}.other*.\n\text{open(f,mode)}\{\text{admFiles}\}|\{\text{model=O.RDONLY}\}.other*
\]

3.1.2 Representing Security Policy through EFSA

Extended Finite State Automata [7]: An EFA is a septuple:
\[(Q,q_0,F,V,Env,\Sigma,\delta)\], where:

• \(Q\) is a finite set of states.
• \(q_0 \in Q\) is the start state.
• \(F \subseteq Q\) 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.
• \(\delta: Q \times \Sigma \times Env \rightarrow 2^{(Q \times Env)}\), 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:

$$\delta^*(q, e, E) = \{(q, E)\}$$
$$\delta^*(q, e_1, e_2, \ldots, e_n, E) = \bigcup_{(q', e_1, e_2, \ldots, e_n, E') \in \delta^*} \delta^*(q', e_1, e_2, \ldots, e_n, E')$$

Here $e_1, e_2, \ldots, e_n$ is a history.

An EFA accepts an event history $H$ if and only if there exists an $q \in F$ and an environment $E$ such that $(q, E) \in \delta^*(q_0, H, E_0)$.

EFSA representation of the same is shown in figure 2.

![Figure 2: Policy Example](image)

### 3.2 Representation of Role Change Policy

One probable role change policy could be, after getting a default role some one can change his/her role only after closing all the files he/she has opened. Representation of such policy through REE can be as follows and EFSA is given subsequently.

#### 3.2.1 Representing Role Change Policy through REE

```
List adminFiles = { all admin files };

S1

Open(file) \{ E \in \text{adminFiles } \& \& \text{ mode } = 0 \text{ RONLY } \}

S2

Open(file) \{ E \in \text{adminFiles } \& \& \text{ mode } = 0 \text{ RONLY } \}

S3
```

#### 3.2.2 Representing Role Change Policy through EFSA

To represent the role change policy, we have to consider few function calls which are not system calls but they are application critical. For example, changing the role of a user is not a system call but is an application critical function call. In order to build role change policy few function calls like this has been considered and in the form of a condition few set operation also has been considered like addition, deletion, belongs to etc...
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.

3.4 Verification

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.5 is associated with SU.
The product automaton is represented in figure 6. Let us take an example transition. Say we are at state \( \{4,2,1\} \): first numeral represents 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 “open(...), a transition \( \{4,2,1\} \rightarrow \{5,3,1\} \) with system call “requestRole” and a transition \( \{4,2,1\} \rightarrow \{6,2,1\} \) with “close(...). 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.

### 3.5 Result

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.7).
Figure 6: Product EFSA

Figure 7: Violation Paths

Violations from Access Policy  
Violations from Role Change Policy

q4,r2,p2  
Open(h.mode) [h(’/etc/’)]  
and mode=0_RDONLY  
requestRoleid(su)
4 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. After giving the problem statement we have proposed an efficient solution of it. With RBAC, 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.

5 References


