Access Control Matrix Model

January 14, 2014
1 Modeling
2 What is an ACM?
3 Some examples
   - Boolean expressions for database control
   - History for program execution control
4 Formal model
   - Primitive operations
   - Types of commands
5 Propagating rights
   - Copy and own
   - Attenuation of privilege
6 What Next?
7 Decidability of security
   - Mono-operational command case
   - General case
Models

- Abstract irrelevant details of entity or process being modeled
  - Allows you to focus on aspects that are of interest
  - *If done correctly*, results from analyzing the model apply to entity or process
- Assumption: nothing you omit affects the application of the results
Protection State

Protection state of system describes current settings, values relevant to protection

- Access control matrix representation of protection state
  - Describes protection state precisely
  - Matrix describing rights of subjects (rows) over objects (columns)
  - State transitions change elements of matrix
- **Subject** is active entities (processes, users, etc.)
- **Object** has 2 meanings:
  - Passive entity (*not* a subject)
  - Any entity acting passively (so can be a subject)

Context tells you which sense is used
Description

- Subjects $S = \{s_1, \ldots, s_n\}$
- Objects $O = \{o_1, \ldots, o_m\}$
- Rights $R = \{r_1, \ldots, r_k\}$
- Entries $A[s_i, o_j] \subseteq R$
- $A[s_i, o_j] = \{r_x, \ldots, r_y\}$ means subject $s_i$ has rights $r_x, \ldots, r_y$ over object $o_j$
Access Control Matrix for System

- Processes $p$, $q$
- Files $f$, $g$
- Rights $r$, $w$, $x$, $a$, $o$
  - Rights are merely symbols; interpretation depends on system
  - Example: on UNIX, $r$ means “read” for file and “list” for directory

<table>
<thead>
<tr>
<th></th>
<th>$f$</th>
<th>$g$</th>
<th>$p$</th>
<th>$q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$rwo$</td>
<td>$r$</td>
<td>$rwxo$</td>
<td>$w$</td>
</tr>
<tr>
<td>$q$</td>
<td>$a$</td>
<td>$ro$</td>
<td>$r$</td>
<td>$rwxo$</td>
</tr>
</tbody>
</table>
Access Control Matrix for Program

- Procedures *inc ctr*, *dec ctr*, *manage*
- Variable *counter*
- Rights +, −, x, *call*

<table>
<thead>
<tr>
<th></th>
<th>counter</th>
<th>inc _ctr</th>
<th>dec _ctr</th>
<th>manage</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>inc _ctr</em></td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>dec _ctr</em></td>
<td>−</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>manage</em></td>
<td></td>
<td>call</td>
<td>call</td>
<td>call</td>
</tr>
</tbody>
</table>
Access Control Matrix for Database

- Access control matrix shows allowed access to database fields
  - *Subjects* have attributes
  - *Verbs* define type of access
  - *Rules* associated with objects, verb pair

- Subject attempts to access object
  - Rule for object, verb evaluated
  - Result controls granting, denying access
Boolean Expressions and Access

- **Subject** *annie*: attributes role (artist), groups (creative)
- **Verb** *paint*: default 0 (deny unless explicitly granted)
- **Object** *picture*: Rule is

\[
paint: \text{‘artist’ in } \text{subject.role and ‘creative’ in } \text{subject.groups and } time.hour \geq 0 \text{ and } time.hour < 5
\]
Example: ACM at 3 a.m. and 10 a.m.

At 3 a.m., time condition met; ACM is:

At 10 a.m., time condition not met; ACM is
Executing Downloaded Programs

- Downloaded programs may access system in unauthorized ways
  - Example: Download Trojan horse that modifies configuration, control files
- Condition access rights upon the rights of previously executed code (i.e., history)
  - Each piece of code has set of static rights
  - Executing process has set of current rights
  - When piece of code runs, its rights are set of current rights ∩ set of static rights
Example Programs

main runs, loads helper_proc and runs it

// This routine has no filesystem access rights
// beyond those in a limited, temporary area
procedure helper_proc()
    return sys_kernel_file;
// But this has the right to delete files
program main()
    sys_load_file(helper_proc);
    file = helper_proc();
    sys_delete_file(file);

sys_kernel_file is system kernel
tmp_file file in limited, temporary area helper_proc can access
## History for program execution control

### Accesses

- **Initial static rights:**

<table>
<thead>
<tr>
<th>sys_kernel_file</th>
<th>tmp_file</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>delete</td>
</tr>
<tr>
<td>helper_proc</td>
<td></td>
</tr>
</tbody>
</table>

- **Program starts; its rights are those of **main**:**

<table>
<thead>
<tr>
<th>sys_kernel_file</th>
<th>tmp_file</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>delete</td>
</tr>
<tr>
<td>helper_proc</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>delete</td>
</tr>
</tbody>
</table>

- **After **helper_proc** called, process loses right to delete kernel:**

<table>
<thead>
<tr>
<th>sys_kernel_file</th>
<th>tmp_file</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>delete</td>
</tr>
<tr>
<td>helper_proc</td>
<td></td>
</tr>
<tr>
<td>process</td>
<td></td>
</tr>
</tbody>
</table>
State Transitions

- Represent changes to the protection state of the system
- $\vdash$ represents transition
  - $X_i \vdash_\tau X_{i+1}$: command $\tau$ moves system from state $X_i$ to state $X_{i+1}$
  - $X_i \vdash^* X_{i+1}$: a sequence of commands moves system from state $X_i$ to state $X_{i+1}$
- Commands sometimes called *transformation procedures*
Primitive Operations

- **create subject** $s$; **create object** $o$
  - Creates new row, column in ACM; creates new column in ACM

- **destroy subject** $s$; **destroy object** $o$
  - Deletes row, column from ACM; deletes column from ACM

- **enter** $r$ **into** $A[s, o]$
  - Adds $r$ rights for subject $s$ over object $o$

- **delete** $r$ **from** $A[s, o]$
  - Removes $r$ rights from subject $s$ over object $o$
create subject

- Precondition: \( s \not\in S \)
- Primitive command: \textbf{create subject} \( s \)
- Postconditions:
  - \( S' = S \cup \{s\}, \ O' = O \cup \{s\} \)
  - \( (\forall y \in O')[A'[s, y] = \emptyset], \ (\forall x \in S')[A'[x, s] = \emptyset] \)
  - \( (\forall x \in S)(\forall y \in O)[A'[x, y] = A[x, y]] \)
Primitive operations

create object

- Precondition: \( o \notin O \)
- Primitive command: \textbf{create object} \( o \)
- Postconditions:
  - \( S' = S, \; O' = O \cup \{ o \} \)
  - \( (\forall x \in S')[A'[x, o] = \emptyset] \)
  - \( (\forall x \in S)(\forall y \in O)[A'[x, y] = A[x, y]] \)
Primitive operations

enter

- Precondition: \( s \in S, \ o \in O \)
- Primitive command: enter \( r \) into \( A[s, o] \)
- Postconditions:
  - \( S' = S, \ O' = O \)
  - \( A'[s, o] = A[s, o] \cup \{ r \} \)
  - \( (\forall x \in S)(\forall y \in O' - \{ o \})[A'[x, y] = A[x, y]] \)
  - \( (\forall x \in S - \{ s \})(\forall y \in O')[A'[x, y] = A[x, y]] \)
Primitive operations

**delete**

- **Precondition:** $s \in S$, $o \in O$
- **Primitive command:** *delete r from A*[s, o]*
- **Postconditions:**
  - $S' = S$, $O' = O$
  - $A'[s, o] = A[s, o] - \{r\}$
  - $(\forall x \in S)(\forall y \in O' - \{o\})[A'[x, y] = A[x, y]]$
  - $(\forall x \in S - \{s\})(\forall y \in O')[A'[x, y] = A[x, y]]$
Primitive operations

**destroy subject**

- **Precondition:** $s \in S$
- **Primitive command:** `destroy subject s`
- **Postconditions:**
  - $S' = S - \{s\}, \quad O' = O - \{s\}$
  - $(\forall y \in O')[A'[s, y] = \emptyset], \quad (\forall x \in S')[A'[x, s] = \emptyset]$
  - $(\forall x \in S')(\forall y \in O')[A'[x, y] = A[x, y]]$
Primitive operations

**destroy object**

- **Precondition:** $o \in O$
- **Primitive command:** **destroy object** $s$
- **Postconditions:**
  - $S' = S$, $O' = O - \{o\}$
  - $(\forall x \in S')[A'[x, o] = \emptyset]$
  - $(\forall x \in S)(\forall y \in O)[A'[x, y] = A[x, y]]$
Example: Creating File

Process $p$ creates file $f$ with $r$ and $w$ permissions

```
command make_file(p, f)
    create object f;
    enter own into a[p, f];
    enter r into a[p, f];
    enter w into a[p, f];
end
```
Mono-Operational Commands

- Make process \( p \) the owner of file \( f \)

  \[
  \text{command} \quad \text{make}\cdot\text{owner}(p, f) \\
  \text{enter} \quad \text{own} \quad \text{into} \quad a[p, f]; \\
  \text{end}
  \]

- Single primitive operation in this command
  - So it’s \textit{mono-operational}
Conditional Commands

- If $p$ owns $f$, let $p$ give $q$ $r$ rights over $f$

```plaintext
command grant rights(p, f)
    if own in A[p, f]
    then
        enter r into A[q, f];
    end
```

- Single condition in this command
  - So it's *mono-conditional*
Multiple Conditions

- If \( p \) has both \( r \) and \( c \) rights over \( f \), let \( p \) give \( q \) \( r \) and \( w \) rights over \( f \)

```plaintext
command grant.read.file.ifrc\((p, f)\)
  if \( r \) in \( A[p, f] \) and \( c \) in \( [p, q] \)
  then
    enter \( r \) into \( A[q, f] \);
    enter \( w \) into \( A[q, f] \);
  end
```

- Two conditions in this command
  - So it's bi-conditional
“Or” Conditions

- If \( p \) has either \( r \) or \( c \) rights over \( f \), let \( p \) give \( q \ r \) and \( w \) rights over \( f \)
  - No “or” operator, so we write command for each possibility
  - Then execute them sequentially
  - Note: if multiple conditions hold, actions may be taken more than once (usually to no effect)
Types of commands

**$r, c$ Commands**

```plaintext
command grant \cdot read \cdot file \cdot ifr(p, f)
   if $r \text{ in } A[p, f]$
   then
      enter $r$ into $A[q, f]$;
      enter $w$ into $A[q, f]$;
   end
command grant \cdot read \cdot file \cdot ifc(p, f)
   if $c \text{ in } A[p, f]$
   then
      enter $r$ into $A[q, f]$;
      enter $w$ into $A[q, f]$;
   end
```
Types of commands

**r or c Command**

```
command grant.read.file.ifrorc(p, f)
  grant.read.file.iffr(p, f)
  grant.read.file.ifrc(p, f)
end
```
Copy

- Allows possessor to give rights to another
- Often attached to a right, so only applies to that right
  - $r$ is read right that cannot be copied
  - $rc$ or $r:c$ is read right that can be copied
  - In this case, called a *copy flag*
- Is copy flag copied with copying the associated right?
  - Depends on rules of model, or instantiation of model
- Usually allows possessor to change entries in ACM column
  - Owner of object can add, delete rights over that object for others
- What can be done is system (instantiation) dependent
  - Some disallow giving rights to specific (set of) users
  - Some disallow passing of copy flag to specific (set of) users
Principle of Attenuation of Privilege

- You increase your rights
- You cannot give rights that you do not possess
  - Restricts addition of rights within a system
- Usually *ignored* for owner
  - Why? Owner gives herself rights; gives them to others; deletes her rights
Now What?

- **Very simple model, but very powerful**
- **Will use this to examine decidability of security**
- **Will use very simple definition of “secure”:**
  - Adding a generic right $r$ where there was not one is *leaking*
  - If a system $S$ begins in initial state $s_0$ and it cannot leak right $r$, we consider it secure with respect to the right $r$

We will formalize this and study it
What is “Secure”?

**Leaking**

Adding a generic right $r$ where there was not one is *leaking*

**Safe**

If a system $S$, beginning in initial state $s_0$, cannot leak right $r$, it is *safe* with respect to the right $r$.

Here, “safe” = “secure” for an abstract model
What is Does “Decidable” Mean?

Safety Question

Does there exist an algorithm for determining whether a protection system $S$ with initial state $s_0$ is safe with respect to a generic right $r$?
Mono-Operational Commands

Answer:
Yes!

Proof sketch:
Consider minimal sequence of commands $c_1, \ldots, c_k$ to leak the right

- Can omit **delete, destroy**
- Can merge all **creates** into one

Worst case: insert every right into every entry; with $s$ subjects, $o$ objects, and $n$ rights initially, upper bound is $k \leq n(s + 1)(o + 1)$
Proof (1)

- Consider minimal sequences of commands (of length $m$) needed to leak $r$ from system with initial state $s_0$
  - Identify each command by the type of primitive operation it invokes
- Cannot test for absence of rights, so delete, destroy not relevant
  - Ignore them
- Reorder sequences of commands so all creates come first
  - Can be done because enters require subject, object to exist
- Commands after these creates check only for existence of right
It can be shown (see homework):

- Suppose \( s_1, s_2 \) are created, and commands test rights in \( A[s_1, o_1], A[s_2, o_2] \)
- Doing the same tests on \( A[s_1, o_1] \) and \( A[s_1, o_2] = A[s_1, o_2] \cup A[s_2, o_2] \) gives same result
- Thus all \textit{creates} unnecessary
  - Unless \( s_0 \) is empty; then you need to create it (1 \textit{create})

In \( s_0 \):

- \( |S_0| \) number of subjects, \( |O_0| \) number of objects, \( n \) number of (generic) rights

In worst case, 1 create

- So a total of at most \( (|S_0| + 1)(|O_0| + 1) \) elements

So \( m \leq n(|S_0| + 1)(|O_0| + 1) \)
General Case

Answer:

No

Proof sketch:

1. Show arbitrary Turing machine can be reduced to safety problem
2. Then deciding safety problem means deciding the halting problem
Turing Machine Review

- Infinite tape in one direction
- States $K$, symbols $M$, distinguished blank $\#$
- State transition function $\delta(k, m) = (k', m', L)$ in state $k$ with symbol $m$ under the TM head replace $m$ with $m'$, move head left one square, enter state $k'$
- Halting state is $q_f$
Mapping

Turing machine

access control matrix representation

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>...</td>
</tr>
</tbody>
</table>

↑

\[ k \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ s_4 \]

\[ s_1 \]

\[ s_2 \]

\[ s_3 \]

\[ s_4 \]

\[ \ldots \]

Turing machine with head over square 3 on tape, in state \( k \) and its representation as an access control matrix

\( o \) is own right

\( e \) is end right
Mapping

Turing machine

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>X</td>
<td>D</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \uparrow \]

\[ k_1 \]

access control matrix representation

<table>
<thead>
<tr>
<th></th>
<th>( s_1 )</th>
<th>( s_2 )</th>
<th>( s_3 )</th>
<th>( s_4 )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>A</td>
<td>o</td>
<td></td>
<td></td>
<td>( s_1 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td></td>
<td>B</td>
<td>o</td>
<td></td>
<td>( s_2 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td></td>
<td></td>
<td>X</td>
<td>o</td>
<td>( s_3 )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td>( k_1 )</td>
</tr>
</tbody>
</table>

After \( \delta(k, C) = (k_1, X, R) \), where \( k \) is the previous state and \( k_1 \) the current state.
General case

Command Mapping

\[ \delta(k, C) = (k_1, X, R) \] at intermediate becomes:

\[
\text{command } c_{k,C}(s_i, s_{i+1}) \]
\[
\text{if } o \in A[s_i, s_{i+1}] \text{ and } k \in A[s_i, s_i] \text{ and } C \in A[s_i, s_i] \text{ then}
\]
\[
\text{delete } k \text{ from } A[s_i, s_i];
\]
\[
\text{delete } C \text{ from } A[s_i, s_i];
\]
\[
\text{enter } X \text{ into } A[s_i, s_i];
\]
\[
\text{enter } k_1 \text{ into } A[s_{i+1}, s_{i+1}];
\]
\[
\text{end}
\]
Mapping

Turing machine

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>X</td>
<td>Y</td>
<td>ə</td>
</tr>
</tbody>
</table>

$k_2$  

access control matrix representation

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>A</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>B</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td></td>
<td>X</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_4$</td>
<td></td>
<td></td>
<td>Y</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>$s_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$k_2$ e</td>
</tr>
</tbody>
</table>

After $\delta(k_1, D) = (k_2, Y, R)$, where $k_1$ is the previous state and $k_2$ the current state