02244 Logic for Security
Information Flow
Week 9: Myers' Approach

Sebastian Mödersheim

April 8, 2024

Challenges

1. We add arrays and procedures to our language
   - $D ::= \ldots \mid$ **integer array** $C\ A[n] \mid \ldots$
     $\ldots \mid$ **proc** $p($ **in** $T_1\ C_1\ var_{in}$, **out** $T_2\ C_2\ var_{out})$ is S_{body}
   - $E ::= \ldots \mid A[E_1]$
   - $S ::= \ldots \mid A[E_1] := E_2 \mid$ **call** $p(E,var)$

   Define the new constraints that we need to impose.

2. Draw the AST of Example1 and generate its constraints.

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

D	**Security Class**
integer C A[n]	$\underline{A} =$
proc p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}	$\underline{p} =$ $\underline{var_{in}} =$ $\underline{var_{out}} =$

E	**Security Class**
A[E_1]	$\underline{E} =$

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

D	**Security Class**
integer C A[n]	$\underline{A} = C$
proc p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}	$\underline{p} =$ $\underline{var_{in}} =$ $\underline{var_{out}} =$

E	**Security Class**
A[E_1]	$\underline{E} =$

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

D	**Security Class**
integer C A[n]	$\underline{A} = C$
proc p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}	$\underline{p} = \underline{S_{body}}$ $\underline{var_{in}} =$ $\underline{var_{out}} =$

E	**Security Class**
A[E_1]	$\underline{E} =$

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

D	**Security Class**
integer C A[n]	$\underline{A} = C$
proc p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}	$\underline{p} = \underline{S_{body}}$ $\underline{var_{in}} = C_1$ $\underline{var_{out}} = C_2$

E	**Security Class**
A[E_1]	$\underline{E} =$

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

D	**Security Class**
integer C A[n]	$\underline{A} = C$
proc p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}	$\underline{p} = \underline{S_{body}}$ $\underline{var_{in}} = C_1$ $\underline{var_{out}} = C_2$

E	**Security Class**
A[E_1]	$\underline{E} = \underline{A} \sqcup \underline{E_1}$

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

S	**security class** of S	**constraint**
A[E_1] := E_2	$\underline{S} =$	
call p(E,var)	$\underline{S} =$	

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

S	**security class** of S	**constraint**
A[E_1] := E_2	$\underline{S} =$	$\underline{E_1} \sqcup \underline{E_2} \sqsubseteq \underline{A}$
call p(E,var)	$\underline{S} =$	

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

S	**security class** of S	**constraint**
A[E_1] := E_2	$\underline{S} = \underline{A}$	$\underline{E_1} \sqcup \underline{E_2} \sqsubseteq \underline{A}$
call p(E,var)	$\underline{S} =$	

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

S	**security class** of S	**constraint**
A[E_1] := E_2	$\underline{S} = \underline{A}$	$\underline{E_1} \sqcup \underline{E_2} \sqsubseteq \underline{A}$
call p(E,var)	$\underline{S} = \underline{p}$	

Challenges

We add arrays and procedures to our language

- $\mathsf{D} ::= \ldots \mid \textbf{integer array}\ \mathsf{C}\ \mathsf{A}[n] \mid \ldots$
  $\ldots \mid \textbf{proc}\ \mathsf{p}(\ \textbf{in}\ \mathsf{T}_1\ \mathsf{C}_1\ \mathsf{var}_{\mathsf{in}},\ \textbf{out}\ \mathsf{T}_2\ \mathsf{C}_2\ \mathsf{var}_{\mathsf{out}})\ \mathsf{is}\ \mathsf{S}_{\mathsf{body}}$
- $\mathsf{E} ::= \ldots \mid \mathsf{A}[\mathsf{E}_1]$
- $\mathsf{S} ::= \ldots \mid \mathsf{A}[\mathsf{E}_1] := \mathsf{E}_2 \mid \textbf{call}\ \mathsf{p}(\mathsf{E},\mathsf{var})$

Rules

S	**security class** of S	**constraint**
$\mathsf{A}[\mathsf{E}_1] := \mathsf{E}_2$	$\underline{\mathsf{S}} = \underline{\mathsf{A}}$	$\underline{\mathsf{E}_1} \sqcup \underline{\mathsf{E}_2} \sqsubseteq \underline{\mathsf{A}}$
call p(E,var)	$\underline{\mathsf{S}} = \underline{\mathsf{p}}$	$\underline{\mathsf{E}} \sqsubseteq \underline{\mathsf{var}_{\mathsf{in}}}$

Challenges

We add arrays and procedures to our language

- D ::= ... | **integer array** C A[n] | ...
  ... | **proc** p(**in** T_1 C_1 var_{in}, **out** T_2 C_2 var_{out}) is S_{body}
- E ::= ... | A[E_1]
- S ::= ... | A[E_1] := E_2 | **call** p(E,var)

Rules

S	**security class** of S	**constraint**
A[E_1] := E_2	$\underline{S} = \underline{A}$	$\underline{E_1} \sqcup \underline{E_2} \sqsubseteq \underline{A}$
call p(E,var)	$\underline{S} = \underline{p}$	$\underline{E} \sqsubseteq \underline{\text{var}_{\text{in}}}$
		$\underline{\text{var}_{\text{out}}} \sqsubseteq \underline{\text{var}}$

Declassification

Example: Login-Program

```
pinfo = record [name,password:string{H}]
check_pw(db:array[pinfo]{H},
         name:string{L}, password:string{H})
  returns ret:bool{L}
  i: int{L} :=0;
  match: bool{???} :=false;
  while (i<db.length) do
    if db[i].name=name && db[i].password=password
    then match:=true
    i:=i+1
  ret:=match
```

What label should `match` have?

Declassification

What we cannot do with classical information flow: release some information that is depending on something classified.

- Log in: the password data-base is secret, but the information whether you have entered the right password is not.
- Result of an election: the votes are secret/private, but the result is public.
- Medical database: the medical records are secret, but they may be released to a researcher after personal information is removed.
- Electronic Auction: the max bids of customers is secret at first, but then during bidding they are partially revealed.

Declassification

What we cannot do with classical information flow: release some information that is depending on something classified.

- Log in: the password data-base is secret, but the information whether you have entered the right password is not.
- Result of an election: the votes are secret/private, but the result is public.
- Medical database: the medical records are secret, but they may be released to a researcher after personal information is removed.
- Electronic Auction: the max bids of customers is secret at first, but then during bidding they are partially revealed.

We thus want a mechanism to explicitly declassify information in a fine-grained way.

Declassification

Information Flow gives you a strong guarantee:

An intruder who can only observe the low variables, cannot learn anything about the high variables.

We will logically formalize this guarantee in the next lecture.

Declassification means that you will lose this guarantee.

- Log in: an intruder can do a guessing attack
- Result of an election: you learn a bit about the votes
- Medical database: an intruder may be able to reconstruct some information about the patients.
- Electronic Auction: an intruder learns a bit about the bids

Giving up Control?

If you give an intruder (dishonest person) some information, you lose all control over it. But the world is more complicated.

Giving up Control?

If you give an intruder (dishonest person) some information, you lose all control over it. But the world is more complicated.

Consider a large organization like a hospital:

- Even though the hospital itself is honest, it may run some systems that are not secure.
- Systems that are designed by honest people could have bugs.
- When declassifying information, you may not want to give permission to use the data arbitrarily.
  - ★ There may be a usage policy about using the declassified data, and compliance may be required by law, e.g. GDPR.
  - ★ Similarly, release of data may be subject to usage policy by a contract, e.g., the researchers must make a contract with the hospital to get access to patient data.
- How to formally specify such policies and automatically prove compliance?

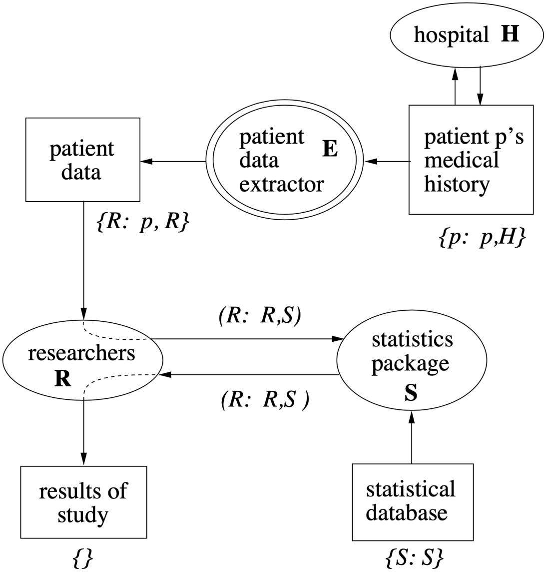
The Decentralized Label Model

Andrew C. Myers and Barbara Liskov: *A Decentralized Model for Information Flow Control*, ACM Symposium on Operating System Principles, 1997 [1].

1. Security lattice: instead of high and low we have more complicated security labels:
   - ★ A set of owners: participants or roles who own the respective data
   - ★ An owner can say who can read the data.
   - ★ You can only read data if all owners have allowed it.
2. We will see later how to compare security labels.
   Except for declassification this is standard information flow à la Denning from the last lecture.
3. Declassify limited: an owner can only relax their own constraint.
4. Programs can act on behalf of an owner and thus declassify, but this forces programmer to make every declassification explicit, so one does not accidentally forget about the rights of some owner.

It is mandatory in the assignment to use the decentralized label model.

Hospital Example



Overview

1. Security lattice:
   - A set of owners: participants or roles who own the respective data
   - An owner can say who can read the data.
   - You can only read data if all owners have allowed it
2. Defining $\sqsubseteq$, $\sqcup$, $\sqcap$.
   Except for declassification this is standard information flow à la Denning from the last lecture.
3. Declassify: an owner can relax their own constraint.
4. Programs can act on behalf of an owner and thus declassify.

1. The Security Lattice

We have the security framework ($P \hookrightarrow \mathrm{PowerSet}(P)$, $\sqsubseteq$, $\sqcup$, $\sqcap$) where

- $P \hookrightarrow \mathrm{PowerSet}(P)$ is the set of all **partial mappings** from P to PowerSet(P).
  - ★ $s_1 = \{\mathtt{A} : \{\mathtt{A}, \mathtt{B}\}\}$
  - ★ $s_2 = \{\mathtt{B} : \{\mathtt{A}\}\}$
  - ★ $s_3 = \{\mathtt{B} : \{\mathtt{A}\}, \mathtt{A} : \{\}\}$
- For a label s we define $\mathrm{Owners}(s) = \mathrm{Domain}(s)$
  - ★ $\mathrm{Owners}(s_1) = \{\mathtt{A}\}$
  - ★ $\mathrm{Owners}(s_2) = \{\mathtt{B}\}$
  - ★ $\mathrm{Owners}(s_3) = \{\mathtt{A}, \mathtt{B}\}$

1. The Security Lattice

We have the security framework ($P \hookrightarrow \text{PowerSet}(P)$, $\sqsubseteq$, $\sqcup$, $\sqcap$) where

- $P \hookrightarrow \text{PowerSet}(P)$ is the set of all **partial mappings** from P to PowerSet(P).
  - ★ $s_1 = \{\texttt{A} : \{\texttt{A}, \texttt{B}\}\}$
  - ★ $s_2 = \{\texttt{B} : \{\texttt{A}\}\}$
  - ★ $s_3 = \{\texttt{B} : \{\texttt{A}\}, \texttt{A} : \{\}\}$
- For a label s we define $\text{Owners}(s) = \text{Domain}(s)$
- For a security label s and principal p define

$$\text{Readers}(s, p) = \begin{cases} s(p) & \text{if } p \in \text{Owners}(s) \\ P & \text{if } p \notin \text{Owners}(s) \end{cases}$$

  - ★ $\text{Readers}(s_1, \texttt{A}) = \{\texttt{A}, \texttt{B}\}$
  - ★ $\text{Readers}(s_2, \texttt{B}) = \{A\}$
  - ★ $\text{Readers}(s_3, \texttt{B}) = \{A\}$
  - ★ $\text{Readers}(s_3, \texttt{A}) = \{\}$
  - ★ $\text{Readers}(s_3, \texttt{C}) = P$ (everybody)

1. The Security Lattice

We have the security framework $(P \hookrightarrow \text{PowerSet}(P), \sqsubseteq, \sqcup, \sqcap)$ where

- $P \hookrightarrow \text{PowerSet}(P)$ is the set of all **partial mappings** from P to PowerSet(P).
  - ★ $s_1 = \{\mathtt{A} : \{\mathtt{A}, \mathtt{B}\}\}$
  - ★ $s_2 = \{\mathtt{B} : \{\mathtt{A}\}\}$
  - ★ $s_3 = \{\mathtt{B} : \{\mathtt{A}\}, \mathtt{A} : \{\}\}$
- For a label s we define $\text{Owners}(s) = \text{Domain}(s)$
- For a security label s and principal p define
  $$\text{Readers}(s, p) = \begin{cases} s(p) & \text{if } p \in \text{Owners}(s) \\ P & \text{if } p \notin \text{Owners}(s) \end{cases}$$
- Alternative notation (bit easier to read):

$$\{(A : A, C), (B : B, C)\} \text{ for } \{A : \{A, C\}, B : \{B, C\}\}$$

2. Ordering

- for two security labels s_1, s_2 we have
  - ★ $s_1 \sqsubseteq s_2$ iff
    Owners$(s_1) \subseteq$ Owners(s_2) and
    Readers$(s_1, o) \supseteq$ Readers(s_2, o) for every $o \in$ Owners(s_1)
  - ★ $s_1 \sqcup s_2$ such that
    Owners$(s_1 \sqcup s_2) =$ Owners$(s_1) \cup$ Owners(s_2)
    Readers$(s_1 \sqcup s_2, o) =$ Readers$(s_1, o) \cap$ Readers(s_2, o)
    for every $o \in$ Owners$(s_1 \sqcup s_2)$
  - ★ $s_1 \sqcap s_2$ such that
    Owners$(s_1 \sqcap s_2) =$ Owners$(s_1) \cap$ Owners(s_2)
    Readers$(s_1 \sqcap s_2, o) =$ Readers$(s_1, o) \cup$ Readers(s_2, o)
    for every owner $o \in$ Owners$(s_1 \sqcap s_2)$

Examples:

- $\{(A : A, B)\} \sqsubseteq \{(A : A), (B : A, B)\}$
- $\{(A : A, B), (C : A, C)\} \sqcup \{(A : A, C), (B : A, B)\}$
  $= \{(A : A), (B : A, B), (C : A, C)\}$
- Write $\{\perp\}$ for the bottom element ($\sqcap$ of all labels), which is: no owners, **everybody** can read!

Label Interpretation

Useful definition that gives an intuitive explanation to our labels:

In the case of data with a **confidentiality** label s

$$\mathrm{EffectiveReaders(s)} = \bigcap_{\mathrm{o \in Owners(s)}} \mathrm{Readers}(s, o)$$

Only principals in the effective readers set can read the data.

Example $\mathrm{EffectiveReaders}(\{(B : A, B), (A : A)\}) = \{A\}$

3. Declassification rule

Rule (confidentiality) An owner o can declassify their data only in the following ways:

- **add readers** for owner o
- or **remove the owner** o.

$L_1 = \{(A : A, B), (B : B, C, D), (C : A, B, C)\}$
Effective readers: $\{B\}$
Owner A can

- Add readers, e.g.,
  $L_2 = \{(A : A, B, C, D), (B : B, C, D), (C : A, B, C)\}$
  - ★ this makes C an effective reader, but not D because C does not support that.
- Remove itself as owner: $L_3 = \{(B : B, C, D), (C : A, B, C)\}$, thus making C an effective reader by removing the A's constraint that only A and B can read.
  - ★ Removing yourself is equivalent to adding everybody as a reader.

4. The act as relation

Declassification is only allowed to an entity who has the right to declassify.

- We can define for each process that it can act on behalf of principals, e.g., "process X can act on behalf of hospital and patient P"
- One think of this as a form of delegation, e.g., a patient gives the hospital the authority to use some data for some purposes.
- By default, all processes run without any authority.
- The special construct `if_acts_for(X,Y) then Z`
  - ★ checks if the current process `X` is allowed to assume authority `Y`
  - ★ and if so, executes command `Z` with that authority;
- Declassification can only happen in the `Z` block of an `if_acts_for`

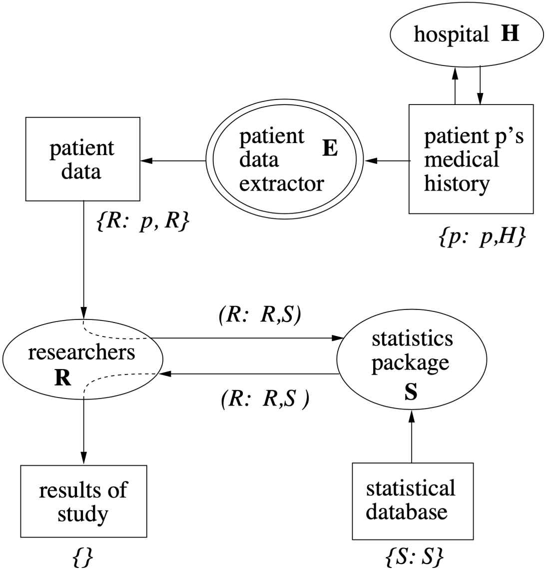
Declassification

Example: Login-Program

```
pinfo = record [name,password:string{chkr:chkr}]
check_pw(db:array[pinfo{⊥}]{⊥},
         name:string{⊥}, password:string{client:chkr})
  returns ret:bool{client:chkr}
  i: int{chkr:chkr} :=0;
  match: bool{client:chkr,chkr:chkr} :=false;
  while (i<db.length) do
    if db[i].name=name && db[i].password=password
    then match:=true
    i:=i+1
  ret:=false
  if_acts_for(check_pw,chkr) then
    ret:=declassify(match,{client:chkr})
```

- `chkr`: special authority that owns the password database.
- The `check_pw` tries to assume the `chkr` authority, and, if successful, declassifies `match`.

Hospital Example



Challenge

```
coronatest = record [ subject : cpr  {shs:shs},
                      testdate: date {shs:shs},
                      positive: bool {shs:shs} ]
publish_stat(db:array[coronatest:{⊥}]{⊥},
             day:date{shs:shs})
returns infections:int{⊥}

lookup_result(db:array[coronatest:{⊥}]{⊥},
               today:date{⊥},
               client:cpr{⊥})
returns result:bool{client:client}
```

`shs` is for sundhedsstyrelsen

Implement the functions `publish_stat` and `lookup_result`

- What authority/declassifications do the functions need?
- Prove that this is indeed safe.

References I

A. C. Myers and B. Liskov.
A decentralized model for information flow control.
In M. Banâtre, H. M. Levy, and W. M. Waite, editors, *ACM Symposium on Operating System Principles*, pages 129–142. ACM, 1997.