Goto Statements

• No assignments
  • Hence no explicit flows
• Need to detect implicit flows
• *Basic block* is sequence of statements that have one entry point and one exit point
  • Control in block *always* flows from entry point to exit point
Example Program

\[
\text{proc } \text{tm}(x: \text{array}[1..10][1..10] \text{ of integer class } \{x\};
\]
\[
\quad \text{var } y: \text{array}[1..10][1..10] \text{ of integer class } \{y\});
\]
\[
\text{var } i, j: \text{integer class } \{i\};
\]
\[
\text{begin}
\]
\[
b_1 \quad i := 1;
\]
\[
b_2 \text{ L2: if } i > 10 \text{ goto L7;}
\]
\[
b_3 \quad j := 1;
\]
\[
b_4 \text{ L4: if } j > 10 \text{ then goto L6;}
\]
\[
b_5 \quad y[j][i] := x[i][j]; \quad j := j + 1; \text{ goto L4;}
\]
\[
b_6 \text{ L6: } i := i + 1; \text{ goto L2;}
\]
\[
b_7 \text{ L7:}
\]
\[
\text{end;}
\]
Flow of Control

b_1 \rightarrow b_2 \quad i > n \quad b_2 \rightarrow b_7

b_6 \quad i \leq n \quad b_3

j > n \quad b_4

j \leq n \quad b_5 \rightarrow b_4

b_7
Immediate Forward Dominators

- Idea: when two paths out of basic block, implicit flow occurs
  - Because information says *which* path to take
- When paths converge, either:
  - Implicit flow becomes irrelevant; or
  - Implicit flow becomes explicit
- *Immediate forward dominator* of basic block \( b \) (written IFD(\( b \))) is first basic block lying on all paths of execution passing through \( b \)
IFD Example

- In previous procedure:
  - $\text{IFD}(b_1) = b_2$ one path
  - $\text{IFD}(b_2) = b_7$ $b_2 \rightarrow b_7$ or $b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_7$
  - $\text{IFD}(b_3) = b_4$ one path
  - $\text{IFD}(b_4) = b_6$ $b_4 \rightarrow b_6$ or $b_4 \rightarrow b_5 \rightarrow b_6$
  - $\text{IFD}(b_5) = b_4$ one path
  - $\text{IFD}(b_6) = b_2$ one path
Requirements

- $B_i$ is set of basic blocks along an execution path from $b_i$ to IFD($b_i$)
  - Analogous to statements in conditional statement
- $x_{i1}, \ldots, x_{in}$ variables in expression selecting which execution path containing basic blocks in $B_i$ used
  - Analogous to conditional expression
- Requirements for secure:
  - All statements in each basic blocks are secure
  - $\text{lub}\{x_{i1}, \ldots, x_{in}\} \leq \text{glb}\{y \mid y \text{ target of assignment in } B_i\}$
Example of Requirements

\[ \text{lub}\{ \text{Low, } i \} \leq i \]

\[ \text{lub}\{ x[j][i], i, j \} \leq y[j][i] \}; \text{lub}\{ \text{Low, } j \} \leq j \]

\[ b_1 \quad i := 1; \]
\[ b_2 \quad \text{L2: if } i > 10 \text{ goto L7;} \]
\[ b_3 \quad j := 1; \]
\[ b_4 \quad \text{L4: if } j > 10 \text{ then goto L6;} \]
\[ b_5 \quad y[j][i] := x[i][j]; \]
\[ \quad j := j + 1; \text{ goto L4;} \]
\[ b_6 \quad \text{L6: i := i + 1; goto L2;} \]
\[ b_7 \quad \text{L7:} \]
Example of Requirements

• Within each basic block:
  \[ b_1: \text{Low} \leq i \quad b_3: \text{Low} \leq j \quad b_6: \text{lub}\{ \text{Low}, i \} \leq i \]
  \[ b_5: \text{lub}\{ x[i][j], i, j \} \leq y[j][i] \}; \text{lub}\{ \text{Low}, i \} \leq i \]
  • Combining, \( \text{lub}\{ x[i][j], i, j \} \leq y[j][i] \}
  • From declarations, true when \( \text{lub}\{ x, i \} \leq y \)

• \( B_2 = \{ b_3, b_4, b_5, b_6 \} \)
  • Assignments to \( i, j, y[j][i] \); conditional is \( i \leq 10 \)
  • Requires \( i \leq \text{glb}\{ i, j, y[j][i] \} \)
  • From declarations, true when \( i \leq y \)
Example (continued)

• $B_4 = \{ b_5 \}$
  • Assignments to $j$, $y[j][i]$; conditional is $j \leq 10$
  • Requires $j \leq \text{glb}\{ j, y[j][i] \}$
  • From declarations, means $i \leq y$

• Result:
  • Combine $\text{lub}\{ x, i \} \leq y; i \leq y; i \leq y$
  • Requirement is $\text{lub}\{ x, i \} \leq y$
Procedure Calls

\[\text{tm}(a, b)\];

From previous slides, to be secure, \(\text{lub}\{x, i\} \leq y\) must hold

- In call, \(x\) corresponds to \(a\), \(y\) to \(b\)
- Means that \(\text{lub}\{a, i\} \leq b\), or \(a \leq b\)

More generally:

\[
\text{proc } \text{pn}(i_1, \ldots, i_m: \text{int}; \text{var } o_1, \ldots, o_n: \text{int})\; \text{begin } S \text{ end;}
\]

- \(S\) must be secure
- For all \(j\) and \(k\), if \(i_j \leq o_k\), then \(x_j \leq y_k\)
- For all \(j\) and \(k\), if \(o_j \leq o_k\), then \(y_j \leq y_k\)
Exceptions

```plaintext
proc copy(x: integer class { x });
    var y: integer class Low);
var sum: integer class { x };
    z: int class Low;
begin
    y := z := sum := 0;
    while z = 0 do begin
        sum := sum + x;
        y := y + 1;
    end
end
```
Exceptions (cont)

• When sum overflows, integer overflow trap
  • Procedure exits
  • Value of \( sum \) is MAXINT/\( y \)
  • Information flows from \( y \) to \( sum \), but \( sum \leq y \) never checked

• Need to handle exceptions explicitly
  • Idea: on integer overflow, terminate loop
    \[
    \text{on integer_overflow_exception } \quad \begin{array}{l}
    \text{sum do } \quad \begin{array}{l}
    z := 1;
    \end{array}
    \end{array}
    \]
  • Now information flows from \( sum \) to \( z \), meaning \( sum \leq z \)
  • This is false (\( sum = \{ x \} \) dominates \( z = \text{Low} \))
Infinite Loops

```plaintext
proc copy(x: integer 0..1 class { x });
    var y: integer 0..1 class Low);
begin
    y := 0;
    while x = 0 do
        (* nothing *);
    y := 1;
end
```

- If \( x = 0 \) initially, infinite loop
- If \( x = 1 \) initially, terminates with \( y \) set to 1
- No explicit flows, but implicit flow from \( x \) to \( y \)
Semaphores

Use these constructs:

\[
\text{wait}(x): \quad \text{if } x = 0 \quad \text{then block until } x > 0; \quad x := x - 1;
\]

\[
\text{signal}(x): \quad x := x + 1;
\]

• \(x\) is semaphore, a shared variable
• Both executed atomically

Consider statement

\[
\text{wait}(\text{sem}); \quad x := x + 1;
\]

• Implicit flow from \(\text{sem}\) to \(x\)
  • Certification must take this into account!
Flow Requirements

• Semaphores in *signal* irrelevant
  • Don’t affect information flow in that process

• Statement S is a *wait*
  • shared(S): set of shared variables read
    • Idea: information flows out of variables in shared(S)
  • fglb(S): glb of assignment targets *following* S
  • So, requirement is shared(S) ≤ fglb(S)

• begin S₁; ... Sₙ end
  • All Sᵢ must be secure
  • For all i, shared(Sᵢ) ≤ fglb(Sᵢ)
Example

begin
  \[ x := y + z; \quad (* S_1 *) \]
  \[ \text{wait}(sem); \quad (* S_2 *) \]
  \[ a := b \times c - x; \quad (* S_3 *) \]
end

• Requirements:
  • \( \text{lub}\{y, z\} \leq x \)
  • \( \text{lub}\{b, c, x\} \leq a \)
  • \( \text{sem} \leq a \)
    • Because \( \text{fglb}(S_2) = a \) and \( \text{shared}(S_2) = \text{sem} \)
Concurrent Loops

• Similar, but wait in loop affects *all* statements in loop
  • Because if flow of control loops, statements in loop before wait may be executed after wait

• Requirements
  • Loop terminates
  • All statements $S_1, \ldots, S_n$ in loop secure
  • $\text{lub}\{\text{shared}(S_1), \ldots, \text{shared}(S_n)\} \leq \text{glb}(t_1, \ldots, t_m)$
    • Where $t_1, \ldots, t_m$ are variables assigned to in loop
Loop Example

while $i < n$ do begin
  $a[i] := item$; (* $S_1$ *)
  wait$(sem)$; (* $S_2$ *)
  $i := i + 1$; (* $S_3$ *)
end

• Conditions for this to be secure:
  • Loop terminates, so this condition met
  • $S_1$ secure if lub{$i, item$} $\leq a[i]$
  • $S_2$ secure if $sem \leq i$ and $sem \leq a[i]$
  • $S_3$ trivially secure
cobegin/coend

cobegin

\[ x := y + z; \quad (\ast S_1 \ast) \]
\[ a := b \ast c - y; \quad (\ast S_2 \ast) \]

coen
d

• No information flow among statements
  • For \( S_1 \), \( \mathrm{lub}\{y, z\} \leq x \)
  • For \( S_2 \), \( \mathrm{lub}\{b, c, y\} \leq a \)

• Security requirement is both must hold
  • So this is secure if \( \mathrm{lub}\{y, z\} \leq x \land \mathrm{lub}\{b, c, y\} \leq a \)
Soundness

• Above exposition intuitive
• Can be made rigorous:
  • Express flows as types
  • Equate certification to correct use of types
  • Checking for valid information flows same as checking types conform to semantics imposed by security policy
Execution-Based Mechanisms

• Detect and stop flows of information that violate policy
  • Done at run time, not compile time

• Obvious approach: check explicit flows
  • Problem: assume for security, $x \leq y$
    
    ```
    if x = 1 then y := a;
    ```
  • When $x \neq 1$, $\chi = \text{High}$, $y = \text{Low}$, $a = \text{Low}$, appears okay—but implicit flow violates condition!
Fenton’s Data Mark Machine

• Each variable has an associated class
• Program counter (PC) has one too
• Idea: branches are assignments to PC, so you can treat implicit flows as explicit flows
• Stack-based machine, so everything done in terms of pushing onto and popping from a program stack
Instruction Description

• \textit{skip}: instruction not executed
• \textit{push}(x, \_x\_): push variable $x$ and its security class $\_x\_$ onto program stack
• \textit{pop}(x, \_x\_): pop top value and security class from program stack, assign them to variable $x$ and its security class $\_x\_$ respectively
Instructions

• $x := x + 1$ (increment)
  • Same as:
    \[
    \text{if } PC \leq x \text{ then } x := x + 1 \text{ else } \text{skip}
    \]

• if $x = 0$ then goto $n$ else $x := x - 1$ (branch and save PC on stack)
  • Same as:
    \[
    \text{if } x = 0 \text{ then begin}
    \quad \text{push}(PC, \ PC); \ PC := \text{lub}\{PC, \ x\}; \ PC := n;
    \quad \text{end else if } PC \leq x \text{ then}
    \quad x := x - 1
    \quad \text{else}
    \quad \text{skip};
    \]
More Instructions

- if’ $x = 0$ then goto $n$ else $x := x - 1$ (branch without saving PC on stack)
  - Same as:
    
    ```
    if $x = 0$ then
      if $x \leq PC$ then $PC := n$ else skip
    else
      if $PC \leq x$ then $x := x - 1$ else skip
    ```
More Instructions

- `return` (go to just after last `if`)
  - Same as:
    - `pop(PC, PC);`

- `halt` (stop)
  - Same as:
    - `if program stack empty then halt`
  - Note stack empty to prevent user obtaining information from it after halting
Example Program

1 if $x = 0$ then goto 4 else $x := x - 1$
2 if $z = 0$ then goto 6 else $z := z - 1$
3 halt
4 $z := z - 1$
5 return
6 $y := y - 1$
7 return

Initially $x = 0$ or $x = 1$, $y = 0$, $z = 0$

Program copies value of $x$ to $y$
### Example Execution

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
<th>PC</th>
<th>PC</th>
<th>stack</th>
<th>check</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Low</td>
<td>—</td>
<td>Low ≤ x</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>z</td>
<td>(3, Low)</td>
<td>PC ≤ y</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>z</td>
<td>(3, Low)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>Low</td>
<td>—</td>
<td></td>
</tr>
</tbody>
</table>
Handling Errors

• Ignore statement that causes error, but continue execution
  • If aborted or a visible exception taken, user could deduce information
  • Means errors cannot be reported unless user has clearance at least equal to that of the information causing the error
Variable Classes

• Up to now, classes fixed
  • Check relationships on assignment, etc.

• Consider variable classes
  • Fenton’s Data Mark Machine does this for $PC$
  • On assignment of form $y := f(x_1, ..., x_n)$, $y$ changed to $\text{lub}\{x_1, ..., x_n\}$
  • Need to consider implicit flows, also
Example Program

(* Copy value from \( x \) to \( y \). Initially, \( x \) is 0 or 1 *)

```plaintext
proc copy(x: integer class { x });
    var y: integer class { y }
var z: integer class variable { Low };
begin
    y := 0;
    z := 0;
    if x = 0 then z := 1;
    if z = 0 then y := 1;
end;
```

• \( z \) changes when \( z \) assigned to

• Assume \( y < x \) (that is, \( x \) strictly dominates \( y \); they are not equal)
Analysis of Example

• $x = 0$
  • $z := 0$ sets $z$ to Low
  • if $x = 0$ then $z := 1$ sets $z$ to 1 and $z$ to $x$
  • So on exit, $y = 0$

• $x = 1$
  • $z := 0$ sets $z$ to Low
  • if $z = 0$ then $y := 1$ sets $y$ to 1 and checks that lub{Low, $z$} ≤ $y$
  • So on exit, $y = 1$

• Information flowed from $x$ to $y$ even though $y < x$
Handling This (1)

• Fenton’s Data Mark Machine detects implicit flows violating certification rules
Handling This (2)

• Raise class of variables assigned to in conditionals even when branch not taken
• Also, verify information flow requirements even when branch not taken
• Example:
  • In \texttt{if } x = 0 \texttt{ then } z := 1, z \text{ raised to } x \text{ whether or not } x = 0
  • Certification check in next statement, that \( z \leq y \), fails, as \( z = x \) from previous statement, and \( y < x \)
Handling This (3)

• Change classes only when explicit flows occur, but *all* flows (implicit as well as explicit) force certification checks

• Example
  • When $x = 0$, first if sets $z$ to Low, then checks $x \leq z$
  • When $x = 1$, first if checks $x \leq z$
  • This holds if and only if $x = \text{Low}$
    • Not possible as $y < x = \text{Low}$ by assumption and there is no class that Low strictly dominates
Quiz

Should a statement that causes an error be ignored, and execution continue?

1. Yes; if the program is aborted or a visible exception is taken, the user could deduce information about values in the program

2. Yes; such a statement cannot be certified and so it must be ignored

3. No; the user must be informed lest they draw an incorrect conclusion about values in the program

4. No; the user’s clearance may allow them to see that an error occurred
Integrity Mechanisms

• The above also works with Biba, as it is mathematical dual of Bell-LaPadula

• All constraints are simply duals of confidentiality-based ones presented above
Example 1

For information flow of assignment statement:

\[ y := f(x_1, \ldots, x_n) \]

the relation \( \text{glb}\{ x_1, \ldots, x_n \} \geq y \) must hold

- Why? Because information flows from \( x_1, \ldots, x_n \) to \( y \), and under Biba, information must flow from a higher (or equal) class to a lower one.
Example 2

For information flow of conditional statement:

\[ \text{if } f(x_1, \ldots, x_n) \text{ then } S_1; \text{ else } S_2; \text{ end; } \]

then the following must hold:

• \( S_1, S_2 \) must satisfy integrity constraints
• \( \text{glb}\{ x_1, \ldots, x_n \} \geq \text{lub}\{ y | y \text{ target of assignment in } S_1, S_2 \} \)
Example Information Flow Control Systems

• Privacy and Android Cell Phones
  • Analyzes data being sent from the phone
• Firewalls
Privacy and Android Cell Phones

• Many commercial apps use advertising libraries to monitor clicks, fetch ads, display them
  • So they send information, ostensibly to help tailor advertising to you

• Many apps ask to have full access to phone, data
  • This is because of complexity of permission structure of Android system

• Ads displayed with privileges of app
  • And if they use Javascript, that executes with those privileges
  • So if it has full access privilege, it can send contact lists, other information to others

• Information flow problem as information is flowing from phone to external party
Analyzing Android Flows

• Android based on Linux
  • App executables in bytecode format (Dalvik executables, or DEX) and run in Dalvik VM
  • Apps event driven
  • Apps use system libraries to do many of their functions
  • Binder subsystem controls interprocess communication

• Analysis uses 2 security levels, *untainted* and *tainted*
  • No categories, and *tainted* < *untainted*
TaintDroid: Checking Information Flows

• All objects tagged \textit{tainted} or \textit{untainted}
  • Interpreters, Binder augmented to handle tags

• Android native libraries trusted
  • Those communicating externally are \textit{taint sinks}

• When untrusted app invokes a taint sink library, taint tag of data is recorded

• Taint tags assigned to external variables, library return values
  • These are assigned based on knowledge of what native code does

• Files have single taint tag, updated when file is written

• Database queries retrieve information, so tag determined by database query responder
TaintDroid: Checking Information Flows

• Information from phone sensor may be sensitive; if so, \textit{tainted}
  • TaintDroid determines this from characteristics of information
• Experiment 1 (2010): selected 30 popular apps out of a set of 358 that required permission to access Internet, phone location, camera, or microphone; also could access cell phone information
  • 105 network connections accessed \textit{tainted} data
  • 2 sent phone identification information to a server
  • 9 sent device identifiers to third parties, and 2 didn’t tell user
  • 15 sent location information to third parties, none told user
  • No false positives
TaintDroid: Checking Information Flows

• Experiment 2 (2012): revisited 18 out of the 30 apps (others did not run on current version of Android)
  • 3 still sent location information to third parties
  • 8 sent device identification information to third parties without consent
    • 3 of these did so in 2010 experiment
    • 5 were new
  • 2 new flows that could reveal tainted data
  • No false positives
Firewalls

• Host that mediates access to a network
  • Allows, disallows accesses based on configuration and type of access

• Example: block Conficker worm
  • Conficker connects to botnet, which can use system for many purposes
    • Spreads through a vulnerability in a particular network service
  • Firewall analyze packets using that service remotely, and look for Conficker and its variants
    • If found, packets discarded, and other actions may be taken
  • Conficker also generates list of host names, tried to contact botnets at those hosts
    • As set of domains known, firewall can also block outbound traffic to those hosts
Filtering Firewalls

- Access control based on attributes of packets and packet headers
  - Such as destination address, port numbers, options, etc.
  - Also called a *packet filtering firewall*
- Does not control access based on content
- Examples: routers, other infrastructure systems
Proxy

• Intermediate agent or server acting on behalf of endpoint without allowing a direct connection between the two endpoints
  • So each endpoint talks to proxy, thinking it is talking to other endpoint
  • Proxy decides whether to forward messages, and whether to alter them
Proxy Firewall

• Access control done with proxies
  • Usually bases access control on content as well as source, destination addresses, etc.
  • Also called an *applications level* or *application level firewall*
• Example: virus checking in electronic mail
  • Incoming mail goes to proxy firewall
  • Proxy firewall receives mail, scans it
  • If no virus, mail forwarded to destination
  • If virus, mail rejected or disinfected before forwarding
Example

- Want to scan incoming email for malware
- Firewall acts as recipient, gets packets making up message and reassembles the message
  - It then scans the message for malware
  - If none, message forwarded
  - If some found, mail is discarded (or some other appropriate action)
- As email reassembled at firewall by a mail agent acting on behalf of mail agent at destination, it’s a proxy firewall (application layer firewall)
Stateful Firewall

• Keeps track of the state of each connection

• Similar to a proxy firewall
  • No proxies involved, but this can examine contents of connections
  • Analyzes each packet, keeps track of state
  • When state indicates an attack, connection blocked or some other appropriate action taken
Network Organization: DMZ

• DMZ is portion of network separating a purely internal network from external network
• Usually put systems that need to connect to the Internet here
• Firewall separates DMZ from purely internal network
• Firewall controls what information is allowed to flow through it
  • Control is bidirectional; it control flow in both directions
One Setup of DMZ

One dual-homed firewall that routes messages to internal network or DMZ as appropriate.
Another Setup of DMZ

Two firewalls, one (outer firewall) connected to the Internet, the other (inner firewall) connected to internal network, and the DMZ is between the firewalls.