Lecture 21: Dynamic Information Flow Control
Information flow policies

Automatic deduction of policies!
Labels represent policies
Noninterference
[Goguen and Meseguer 1982]

An interpretation of noninterference for a program:
- Changes on H inputs should not cause changes on L outputs.
Static type system

Assignment-Rule: \[
\Gamma \vdash e : \ell \quad \ell \sqcup ctx \sqsubseteq \Gamma(x) \\
\Gamma, ctx \vdash x := e
\]

If-Rule: \[
\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup ctx \vdash c_1 \quad \Gamma, \ell \sqcup ctx \vdash c_2 \\
\Gamma, ctx \vdash \text{if } e \text{ then } c_1 \text{ else } c_2
\]

While-Rule: \[
\Gamma \vdash e : \ell \quad \Gamma, \ell \sqcup ctx \vdash c \\
\Gamma, ctx \vdash \text{while } e \text{ do } c
\]

Sequence-Rule: \[
\Gamma, ctx \vdash c_1 \quad \Gamma, ctx \vdash c_2 \\
\Gamma, ctx \vdash c_1; c_2
\]
This type system is not complete.

- \( c \) satisfies noninterference \( \notimplies \Gamma, \text{ctx} \vdash c \)
  - There is a command \( c \), such that noninterference is satisfied, but \( c \) is not type correct.

- **Example 1:**
  - \( \Gamma(x) = H, \Gamma(y) = L \)
  - \( c \) is if \( x > 0 \) then \( y := 1 \) else \( y := 1 \)

- **Example 2:**
  - \( \Gamma(x) = H, \Gamma(y) = L \)
  - \( c \) is if \( 1 = 1 \) then \( y := 1 \) else \( y := x \)

- So, this type system is *conservative*. It has *false negatives*:
  - There are programs that are not type correct, but that satisfy noninterference.
Can we build a complete mechanism?

• Is there an enforcement mechanism for information flow control that has no false negatives?
  • A mechanism that rejects only programs that do not satisfy noninterference?

• No! [Sabelfeld and Myers, 2003]
  • “The general problem of confidentiality for programs is undecidable.”
  • The halting problem can be reduced to the information flow control problem.
  • Example:
    ```c
    if h>1 then c; l:=2 else skip
    ```
  • If we could precisely decide whether this program is secure, we could decide whether `c` terminates!
DYNAMIC ENFORCEMENT
Dynamic Enforcement

- Dynamic mechanisms use run time information to decrease false negatives.
- A dynamic mechanism (monitor) checks/deduces labels along the execution:
  - When an assignment $\texttt{x := e}$ is executed,
    - either check whether $\Gamma(\texttt{e}) \cup \texttt{ctx} \sqsupseteq \Gamma(\texttt{x})$ holds (fixed $\Gamma$),
      - The execution of a program is halted when a check fails.
    - or deduce $\Gamma(\texttt{x})$ such that $\Gamma(\texttt{e}) \cup \texttt{ctx} \sqsubseteq \Gamma(\texttt{x})$ holds (flow-sensitive $\Gamma$).
  - Monitor maintains a context label $\texttt{ctx}$. When execution enters a conditional command, the mechanism augments $\texttt{ctx}$ with the label of the guard.
Dynamic Enforcement

- Example 2:
  - $\Gamma(x) = H$, $\Gamma(y) = L$
  - $c$ is $\text{if } 1=1 \text{ then } y:=1 \text{ else } y:=x$
  - $c$ satisfies noninterference, because $x$ does not leak to $y$.
  - dynamic check $\Gamma(1) \sqcup \Gamma(1=1) \sqsubseteq \Gamma(y)$ always succeeds, because branch $y:=x$ is never taken.
  - Remember: the static type system rejects this program before execution, even though the program is secure!
But, there is a caveat…

- A dynamic mechanism may leak information
  - when deciding to halt an execution due to a failed check (fixed $\Gamma$), or
  - when deducing labels during execution (flow-sensitive $\Gamma$).
Leaking through halting (fixed $\Gamma$)

- Consider fixed $\Gamma$: $\Gamma(h) = L$ and $\Gamma(1) = H$.
- Consider program:
  ```
  p:=0;
  if h>0 then l:=1 else h:=1;
  l:=2
  ```
- If $h>0$ is true, then execution is halted.
  - No public output.
- If $h>0$ is false, then execution terminates normally.
  - One public output.
- Problem: $h>0$ is leaked to public outputs.
But, there is a caveat…

- A dynamic mechanism may leak information
  - when deciding to halt an execution due to a failed check (fixed $\Gamma$), or
  - when deducing labels during execution (flow-sensitive $\Gamma$).
Leaking through labels (flow-sensitive $\Gamma$)

- Initially: $\Gamma(x) = L$, $\Gamma(y) = L$, $\Gamma(h) = H$
  
  ```
  x := 0;
  if h > 0 then x := 1 else skip
  y := x
  ```

- At termination, when $h \not> 0$: $\Gamma(y) = \Gamma(x) = L$.
  - Two public outputs.
- At termination, when $h > 0$: $\Gamma(y) = \Gamma(x) = H$.
  - No public output.
- Problem: Even though $h$ flows to $x$, $x$ is tagged with $H$ only when $h > 0$. So, $h > 0$ is leaked to public outputs.
The Problem with Dynamic Mechanisms

• Purely dynamic mechanisms are usually unsound.
• Purely dynamic mechanism with additional restrictions can become sound:
  • Restriction: Stop execution whenever the guard expression of a conditional command is high.
  • But, the resulting mechanism is more conservative than desired.
• Alternatively…
Use on-the-fly static analysis

- Use on-the-fly static analysis to update the labels of target variables in untaken branch.
- The resulting mechanism is sound and less conservative.
Use on-the-fly static analysis

Problem: $x$ was tagged with $H$ only when $h>0$ was true, even though $h$ always flow to $x$.
Goal: $x$ should be tagged with $H$ at every execution.

```plaintext
x:=0;
if h>0 then x:=1 else skip
```

$h>0$ is evaluated to $false$. Execute taken branch.
Use on-the-fly static analysis

Problem: \( x \) was tagged with \( H \) only when \( h>0 \) was true, even though \( h \) always flow to \( x \).
Goal: \( x \) should be tagged with \( H \) at every execution.

\[
x := 0;
\]
\[
\text{if } h>0 \text{ then } x := 1 \text{ else skip}
\]

On-the-fly static analysis:
\[
\Gamma(x) = \Gamma(1) \sqcup \Gamma(h>0) = H
\]

Apply on-the-fly static analysis to the untaken branch.
Use on-the-fly static analysis

Problem: $x$ was tagged with $H$ only when $h>0$ was true, even though $h$ always flow to $x$.
Goal: $x$ should be tagged with $H$ at every execution.

```
x:=0;
if h>0 then x:=1 else skip
```

$\Gamma(x) = H$
Static versus Dynamic

• Static:
  • Low run time overhead.
  • No new covert channels.
  • More conservative.

• Dynamic
  • Increased run time overhead.
  • Possible new covert channels.
  • Less conservative.

• Ongoing research for both static and dynamic.
  • Different expressiveness of policies, different NI versions, different mechanisms.
INFORMATION FLOW CONTROL IN PRACTICE(ISH)
Past and current research on dynamic analysis

- RIFLE (ISA) [Vachharajani et al. 2004]
- HiStar (OS) [Zeldovich et al. 2006]
- Trishul (JVM) [Nair et al. 2008]
- TaintDroid (Android) [Enck et al. 2010]
- LIO (Haskell) [Stefan et al. 2011]
- ...
TaintDroid

- Smartphones run apps developed by (potentially untrusted) third parties
- Apps can access sensitive information (location, contacts, etc.)
- In Android, users grant apps particular permissions on download
- End-user license agreement (EULA) states how information will be used
- How can you tell whether app behavior follows its permissions?
TaintDroid Labels

Sensitive, \{GPS, camera\}  
Sensitive, \{GPS, contacts\}  
Sensitive, \{camera, contacts\}  
Sensitive, \{GPS\}  
Sensitive, \{camera\}  
Sensitive, \{contacts\}  
Public
Android Background Info

- Linux-based, open source, mobile-phone platform
- Middleware written in Java and C/C++.
- Functionality implemented by (3rd party) applications.
- Apps run on top of middleware.

- Applications written in Java.
- Compiled into Dalvik Executable (DEX) byte-code format.
  - custom byte-code
  - Register-based as opposed to stack-based.

- Executes within Dalvik VM interpreter instance.
  - Runs isolated on the platform.
  - Has unique UNIX user ids.
  - Communicate via binder IPC mechanism.
TaintTracking

- Instrument VM interpreter to provide variable-level taint tracking
- Use message-level tracking between apps
- Use method-level tracking in native libraries
- Use file-level tracking for persistent data
Limitations

- Dynamic IFC mechanisms incur run-time overhead
  - 14% for CPU bound microbenchmark
  - Negligible for interactive applications
- Doesn't capture implicit flows
Experimental Findings

- Researchers studied real-world apps with TaintDroid
- Of 30 apps, found:

<table>
<thead>
<tr>
<th>Observed Behavior (# of apps)</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone Information to Content Servers (2)</td>
<td>2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app’s content server.</td>
</tr>
<tr>
<td>Device ID to Content Servers (7)*</td>
<td>2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app’s content server.</td>
</tr>
<tr>
<td>Location to Advertisement Servers (15)</td>
<td>5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location↑ to data.flurry.com.</td>
</tr>
</tbody>
</table>

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

↑ To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).
Flume

- Extends linux with process-level information flow control
- User-level implementation
- No new OS, can use existing communication abstractions
Flume Labels

• Lattice of labels
  • Label summarizes which categories of data a process is assumed to have seen.
  • Examples:
    • { “Financial Reports” }
    • { “HR Documents” }
    • { “Financial Reports” and “HR Documents” }

• Processes have an integrity label and a confidentiality label
  • Processes can upgrade their labels
  • Processes can create new tags, can declassify tags they created
  • Inter-process communication mediated by Flume to enforce IFC
Information Flow Control in Flume

- Linux processes communicate via a variety of channels: sockets, pipes, shared memory
- Endpoint abstraction: process can specify which privileges can be used when communicating through each endpoint
Information Flow Control in Flume

- Linux processes communicate via a variety of channels: sockets, pipes, shared memory
- Endpoint abstraction: process can specify which privileges can be used when communicating through each endpoint
- Flume mediates all communications between endpoints (system call delegation)
Information Flow Control in Flume

- Linux processes communicate via a variety of channels: sockets, pipes, shared memory
- Endpoint abstraction: process can specify which privileges can be used when communicating through each endpoint
- Flume mediates all communications between endpoints (system call delegation)

- Flume enforces IFC

\[
\begin{align*}
S_p &= \{\} \\
D_p &= \{\text{HR}\} \\
S_e &= \{\text{HR}\} \\
S_f &= \{\text{HR}\} \\
S_q &= \{\text{HR}\}
\end{align*}
\]
Limitations

- Dynamic IFC mechanisms incur run-time overhead
  - 30-40% reduction in throughput for file I/O
  - Increased latency
- Large trusted computing base
- Coarse granularity
- Alternative solutions:
  - Dedicated OS (e.g., Asbestos, HiStar)
  - PL-level techniques (e.g., DLM, TaintDroid)
Past and current research on static analysis

- [Denning and Denning 1977]
- VSI type system [Volpano, Smith, and Irvine 1996]
- Jif [Myers 1999] Java + Information Flow (originally JFlow)
- FlowCaml [Simonet 2003] OCaml + Information Flow
- Aura, PCML5, Fine, ...
Jif

class passwordFile authority(root) {
    public boolean
        check (String user, String password)
    where authority(root) {
        // Return whether password is correct
        boolean match = false;
        try {
            for (int i = 0; i < names.length; i++) {
                if (names[i] == user &&
                    passwords[i] == password) {
                    match = true;
                    break;
                }
            }
        }
        catch (NullPointerException e) {}
        catch (IndexOutOfBoundsException e) {}
        return declassify(match, {user; password});
    }
    private String [] names;
    private String { root: } [] passwords;
}
Jif

class passwordFile authority(root) {
  public boolean
    check (String user, String password)
  where authority(root) {
    // Return whether password is correct
    boolean match = false;
    try {
      for (int i = 0; i < names.length; i++) {
        if (names[i] == user &&
            passwords[i] == password) {
          match = true;
          break;
        }
      }
    }
    catch (NullPointerException e) {}
    catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }

  private String [] names;
  private String { root: } [] passwords;
}
Jif

class passwordFile authority(root) {
    public boolean
    check (String user, String password)

    where authority(root) {
      // Return whether password is correct
      boolean match = false;
      try {
        for (int i = 0; i < names.length; i++) {
          if (names[i] == user &&
              passwords[i] == password) {
            match = true;
            break;
          }
        }
        catch (NullPointerException e) {}
        catch (IndexOutOfBoundsException e) {}
        return declassify(match, {user; password});
      }
    }

    private String [] names;
    private String { root: } [] passwords;
}
Jif type checking

• Variables (fields, methods, etc.) may have additional label as part of their type, e.g., `int {lbl} x;`

• Label constrains information flow to and from variable
  • **reader label**: `alice -> bob, charlie`
    • Alice owns this constraint; her permission required to violate it
    • Alice permits the information to flow to Bob and Charlie
    • On previous slide: `root:` is short for `root -> root`
  • **writer label**: `alice <- bob, charlie`
    • Alice owns this constraint; her permission required to violate it
    • Alice permits the information to flow from Bob and Charlie
  • can have multiple such constraints as part of label
  • can read these arrows as the may flow relation \(\rightarrow\)
  • Decentralized label model (DLM) [Myers and Liskov 1997]