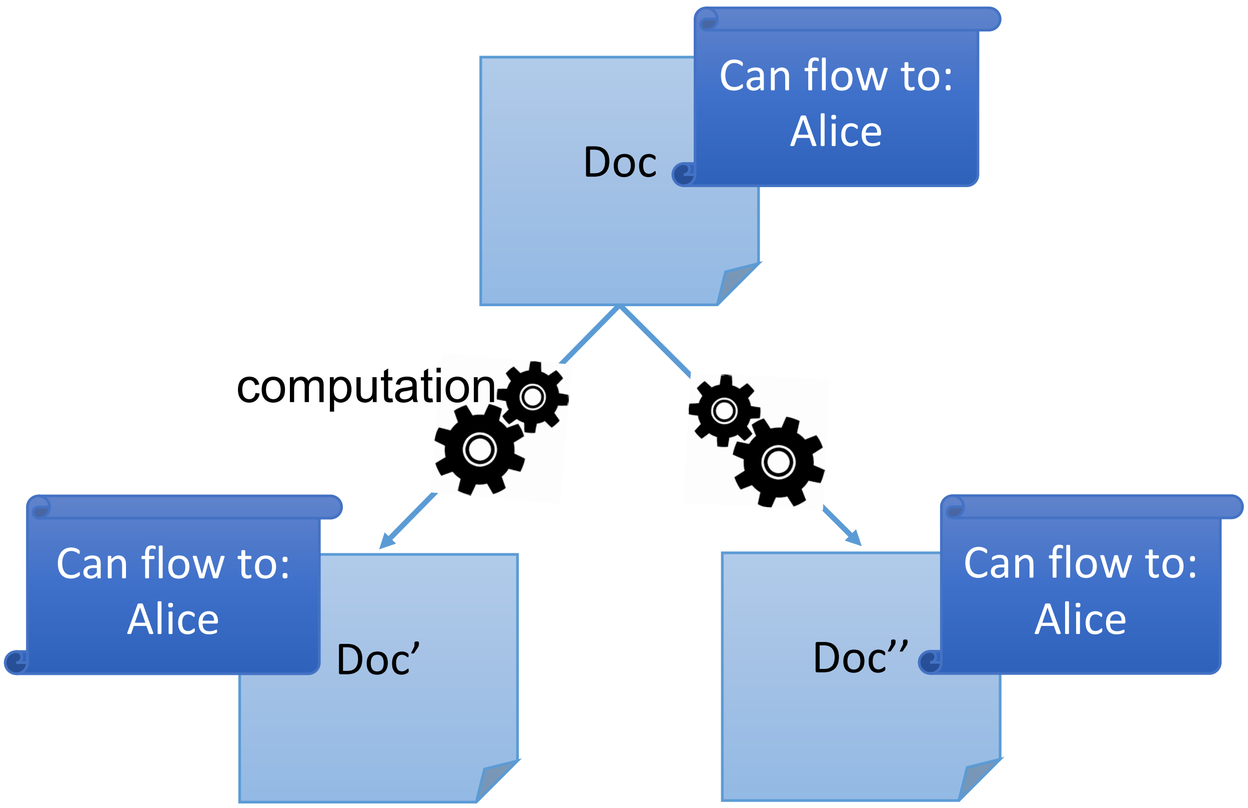
Lecture 21: Dynamic Information Flow Control

CS 181S

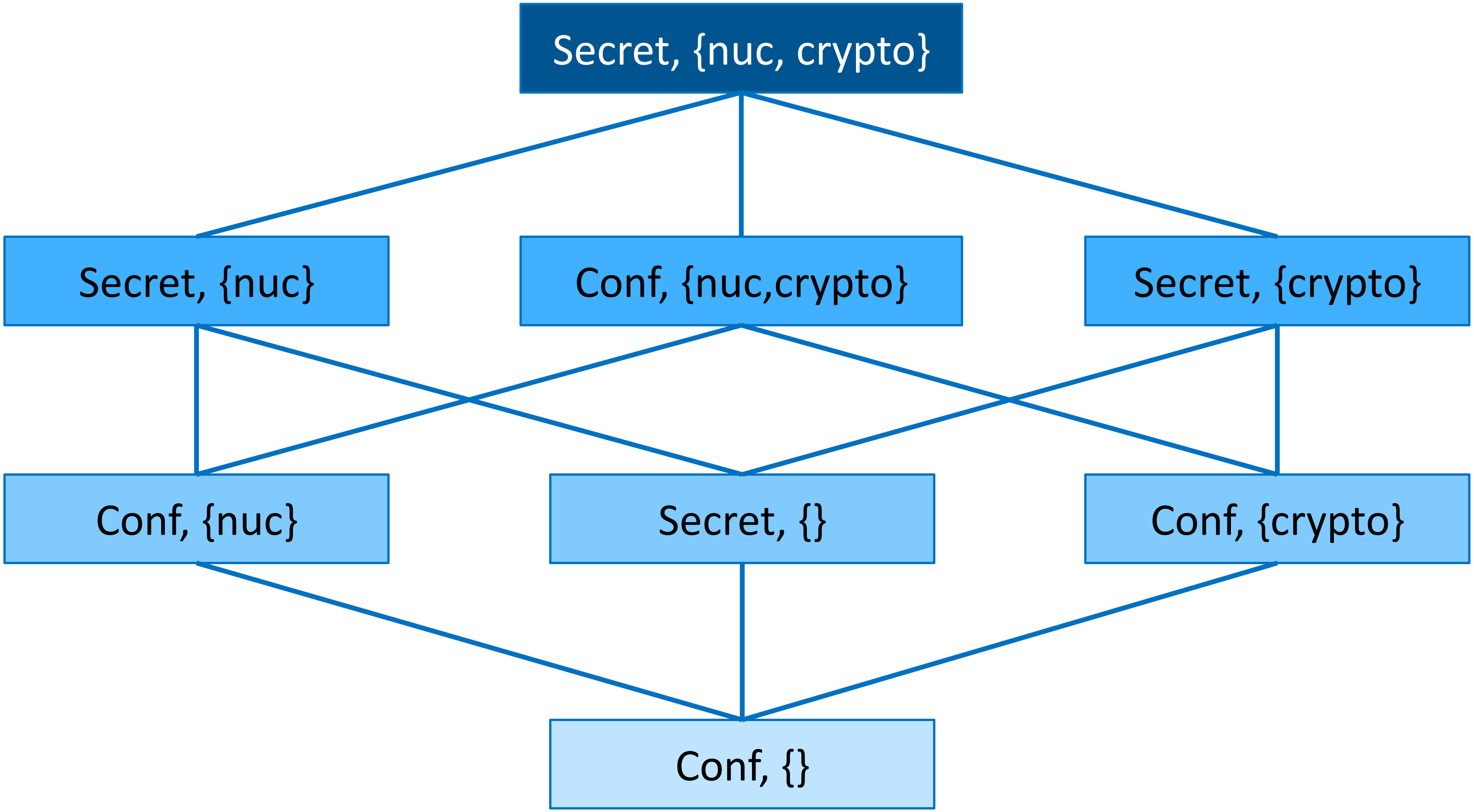
November 12, 2018

Information flow policies



Automatic deduction of policies!

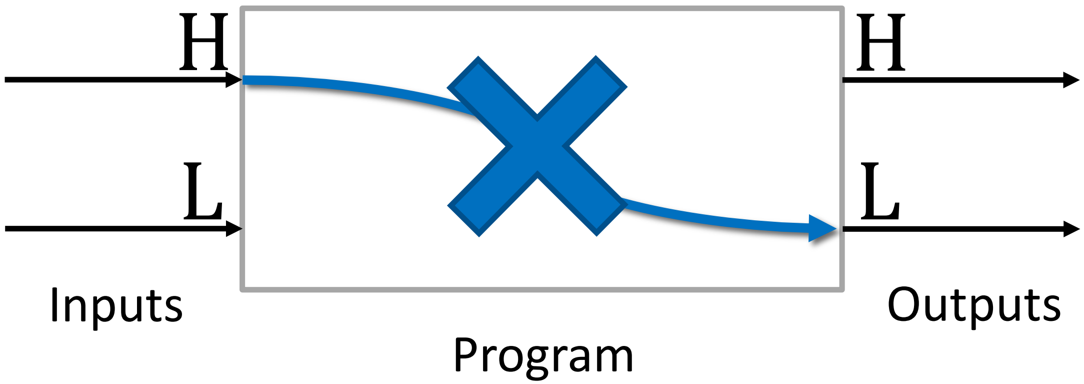
Labels represent policies



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:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \ell \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})}{\Gamma, ctx \vdash \mathbf{x:=e}}$$

If-Rule:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c1} \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c2}}{\Gamma, ctx \vdash \mathbf{if\ e\ then\ c1\ else\ c2}}$$

While-Rule:
$$\frac{\Gamma \vdash \mathbf{e} : \ell \qquad \Gamma, \ell \sqcup ctx \vdash \mathbf{c}}{\Gamma, ctx \vdash \mathbf{while\ e\ do\ c}}$$

Sequence-Rule:
$$\frac{\Gamma, ctx \vdash \mathbf{c1} \qquad \Gamma, ctx \vdash \mathbf{c2}}{\Gamma, ctx \vdash \mathbf{c1;c2}}$$

Soundness of type system

$$\Gamma, ctx \vdash \mathbf{c} \quad \Rightarrow \quad \mathbf{c} \text{ satisfies NI}$$

Limitations of the type system

This type system does not prevent leaks through covert channels.

Example of covert channel:

```
while s != 0 do { //nothing };
p:=1
```

where **s** is a secret variable (i.e., $\Gamma(\mathbf{s})$=H) and **p** is a public variable (i.e., $\Gamma(\mathbf{p})$=L).

A solution

- To prevent covert channels due to infinite loops, strengthen the typing rule for while-statement, to allow only **low** guard expression:

$$\frac{\Gamma \vdash \mathtt{e} : \bot \qquad \Gamma, ctx \vdash \mathtt{c}}{\Gamma, ctx \vdash \mathtt{while\ e\ do\ c}}$$

- Now, type correctness implies termination sensitive NI.
- But, the enforcement mechanism becomes overly conservative.
- Another solution? Research!

This type system is not complete.

- **`c`** satisfies noninterference $\not\Rightarrow$ $\Gamma, ctx \vdash$ **`c`**
  - There is a command **`c`**, such that noninterference is satisfied, but **`c`** is not type correct.
- Example 1:
  - $\Gamma(\mathbf{x}) = \mathrm{H}, \Gamma(\mathbf{y}) = \mathrm{L}$
  - **`c`** is **`if x>0 then y:=1 else y:=1`**
  - **`c`** satisfies noninterference, because **`x`** does not leak to **`y`**.
  - **`c`** is not type correct, because $\Gamma(\mathrm{x}) \not\sqsubseteq \Gamma(\mathrm{y})$.

This type system is not complete.

- Example 2:
  - $\Gamma(x) = H$, $\Gamma(y) = L$
  - **`c`** is **`if 1=1 then y:=1 else y:=x`**
  - **`c`** satisfies noninterference, because **`x`** does not leak to **`y`**.
  - **`c`** is not type correct, because $\Gamma(x) \not\sqsubseteq \Gamma(y)$.
- 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; l:= h
```

    - If we could precisely decide whether this program is secure, we could decide whether `c` terminates!

Can we build a mechanism with fewer false positives?

Switch from static to dynamic mechanisms!

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 **x:=e** is executed,
    - either check whether $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (fixed Γ),
      - The execution of a program is halted when a check fails.
    - or deduce $\Gamma(\mathbf{x})$ such that $\Gamma(\mathbf{e}) \sqcup ctx \sqsubseteq \Gamma(\mathbf{x})$ holds (flow-sensitive Γ).
  - Monitor maintains a context label ctx**.** When execution enters a conditional command, the mechanism augments ctx with the label of the guard.

Dynamic Enforcement

- Example 2:
  - $\Gamma(x) = H, \Gamma(y) = L$
  - **`c`** is **`if 1=1 then y:=1 else y:=x`**
  - **`c`** satisfies noninterference, because **`x`** does not leak to **`y`**.
  - dynamic check $\Gamma(\mathbf{1}) \sqcup \Gamma(\mathbf{1{=}1}) \sqsubseteq \Gamma(\mathbf{y})$ 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 Γ), or
  - when deducing labels during execution (flow-sensitive Γ).

Leaking through halting (fixed Γ)

- Consider fixed Γ: Γ(**h**)=H and Γ(**l**)=L.
- Consider program:

```
l:=0;
if h>0 then l:=1 else h:=1;
l:=2
```

Output

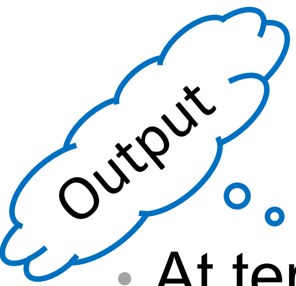
- If **h>0** is *true*, then execution is halted.
  - No public output.
- If **h>0** is *false*, then execution terminates normally.
  - One public output.
- Problem: **h>0** is leaked to public outputs.

But, there is a caveat…

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

Leaking through labels (flow-sensitive Γ)

- Initially: $\Gamma(\mathbf{x}) = \mathrm{L}$, $\Gamma(\mathbf{y}) = \mathrm{L}$, $\Gamma(\mathbf{h}) = \mathrm{H}$




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

- At termination, when **h≯0**: $\Gamma(\mathbf{y}) = \Gamma(\mathbf{x}) = \mathrm{L}$.
  - Two public outputs.
- At termination, when **h>0**: $\Gamma(\mathbf{y}) = \Gamma(\mathbf{x}) = \mathrm{H}$.
  - 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.

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

On-the-fly static analysis:
$\Gamma(\mathbf{x}) = \Gamma(\mathbf{1}) \sqcup \Gamma(\mathbf{h>0}) = \mathrm{H}$

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
```

$\Gamma(\mathbf{x}) = \mathrm{H}$

Exercise

- Consider a dynamic mechanism with on-the-fly static analysis and flow sensitive Γ.
- Assume Γ is initialized as: $\Gamma(\mathbf{x}) = \mathrm{H}$, $\Gamma(\mathbf{y}) = \mathrm{L}$, $\Gamma(\mathbf{z}) = \mathrm{H}$, and consider the following program:

$$\mathbf{x} \coloneqq \mathbf{1};$$

$$\mathbf{y} \coloneqq \mathbf{2};$$

$$\mathbf{if\ z > 0\ then\ y \coloneqq 1\ else\ x \coloneqq 2}$$

- What are the confidentiality labels that tag variables when the program terminates?

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.

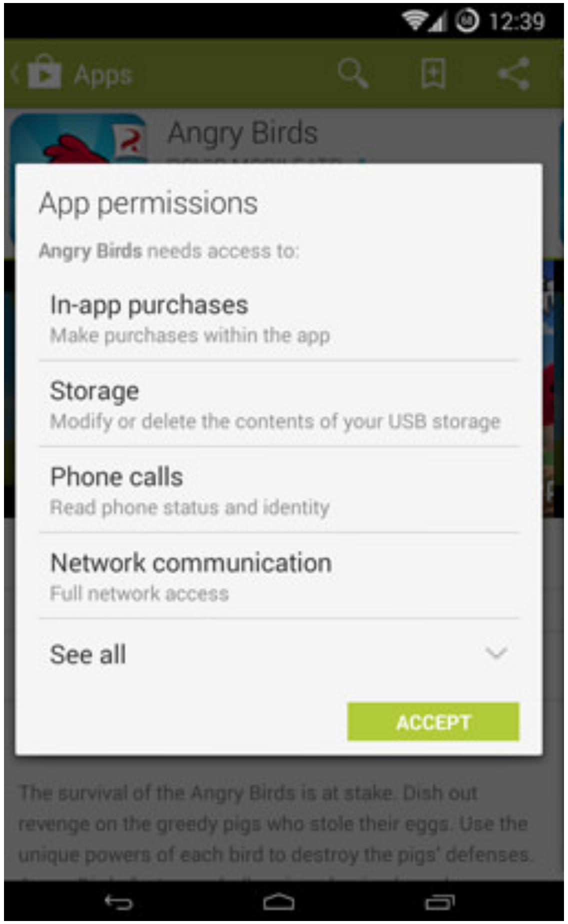
INFORMATION FLOW CONTROL IN PRACTICE(ISH)

Past and current research on dynamic analysis

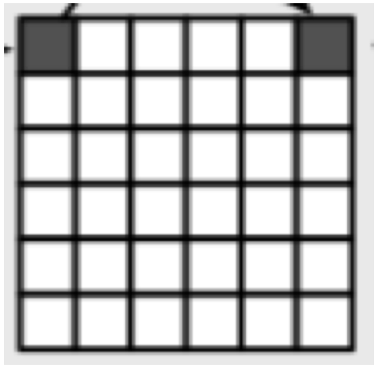
- RIFLE (ISA) [Vachharajani et al. 2004]
- HiStar (OS) [Zeldovich et al. 2006]
- Trishul (JVM) [Nair et al. 2008]
- TaintDroid (Android) [Enck et al. 2010]
- LIO (Haskell) [Stefan et al. 2011]
- ...

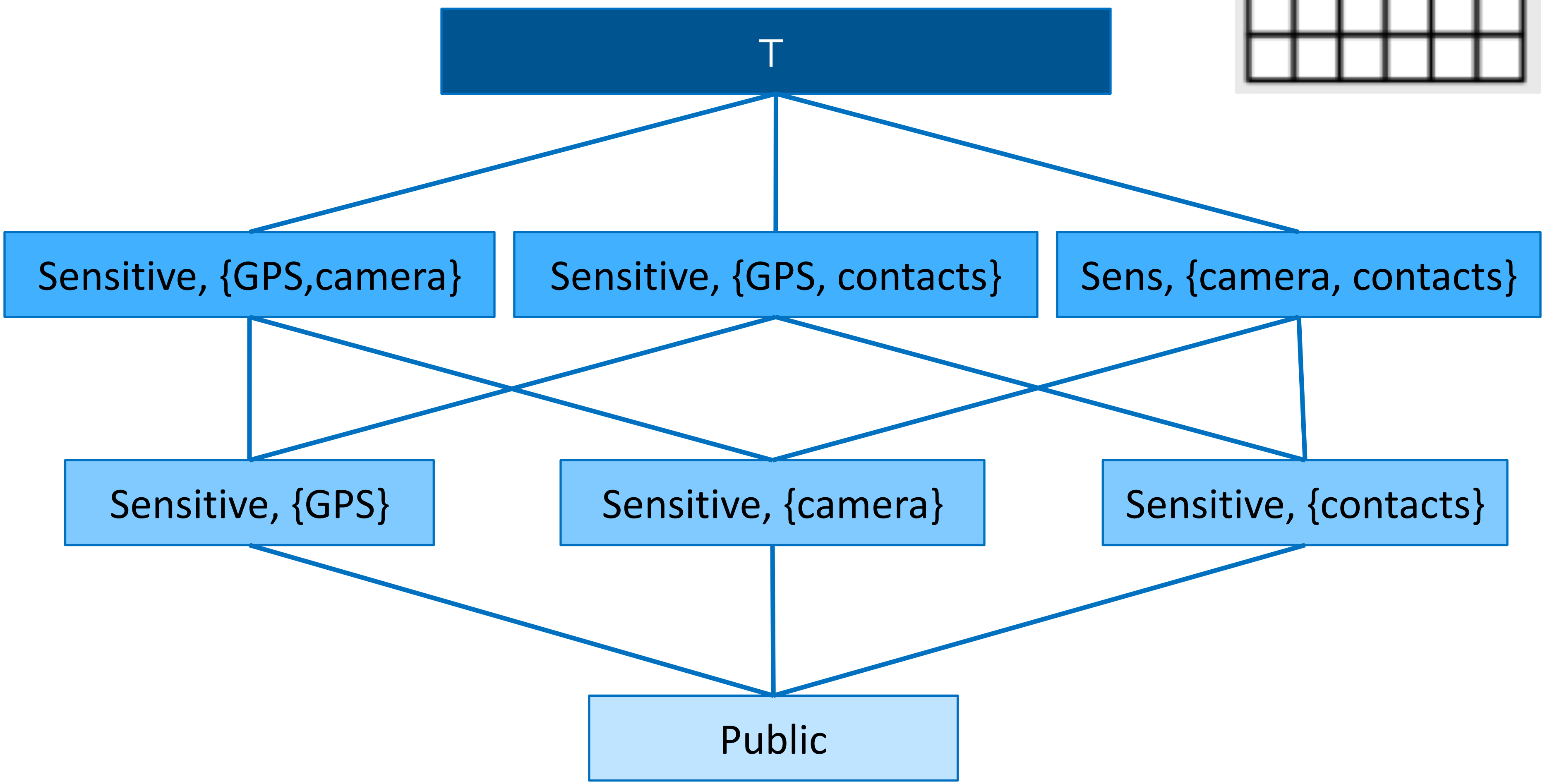
TaintDroid

- 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





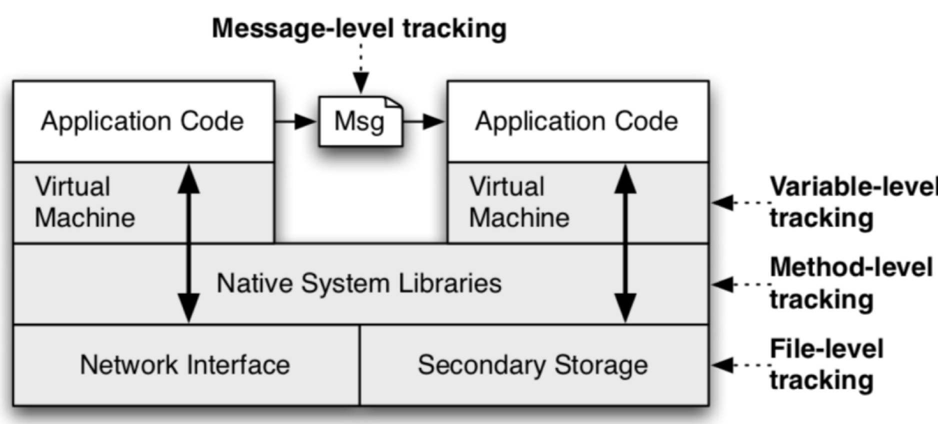
Android Background Info

- 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

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint



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:

Observed Behavior (# of apps)	Details
Phone Information to Content Servers (2)	2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	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.

* 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” }

“tag”

“label”

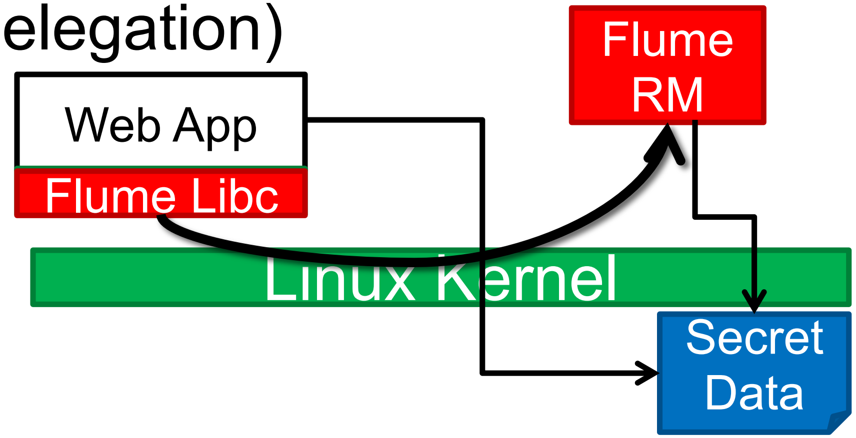
- Processes have an integrity label and a confidentiality label
  - Processes can upgrade their labels
  - Processes can create new tags, can declassify tags they created
  - Inter-process communication mediated by Flume to enforce IFC

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

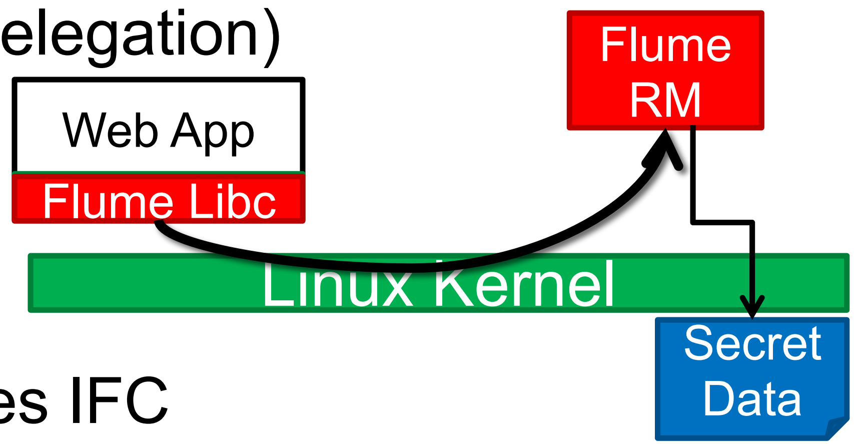
Information Flow Control in Flume

- 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)

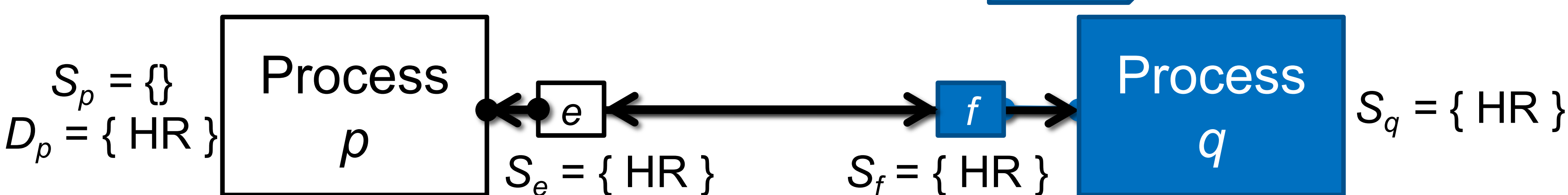


Information Flow Control in Flume

- 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



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)