Software Security

Information Flow

Erik Poll

Digital Security group

Radboud University Nijmegen

Example

- Imagine using a mobile phone app to
  1. locate nearest hotel using google
  2. book a room with your credit card
- Sensitive information?
  - location information
  - credit card no
- (Un)wanted information flows?
  - location may be leaked to google *only*
  - credit card info may be leaked to hotel *only*

Can we prevent this by access control on the mobile phone app?

Information Flow

- An interesting category of security requirements is about information flow. Eg
  - no confidential information should leak over network
  - no untrusted input from network should leak into database
- Information flow properties can be about confidentiality or integrity
- Note the difference with access control:
  - access control is about *access only*
    (eg for mobile phone app, access to the location API)
  - information flow is *also* about *what you do with data after you accessed it* (eg location obtained from this API)

Example Information Flow - Confidentiality

```
String hi; // security label secret
String lo; // security label public
```

Which program fragments (may) cause problems if `hi` has to be kept confidential?

```
1. hi = lo;
2. lo = hi;
3. lo = "1234";
4. hi = "1234";
```

```
5. println(lo);
6. println(hi);
7. readln(lo);
8. readln(hi);
```

Example Information Flow - Confidentiality

```
String hi; // security label secret
String lo; // security label public
```

Which program fragments (may) cause problems if `hi` has to be kept confidential?

✓ `1. hi = lo;`
✗ `2. lo = hi;`
✓ `3. lo = "1234";`
? `4. hi = "1234";` ?

✓ `5. println(lo)`
✗ `6. println(hi);`
✓ `7. readln(lo);`
? `8. readln(hi);`

Example Information Flow - *Integrity*

```
String hi; // high integrity (trusted) data
String lo; // low integrity (untrusted) data
```

Which program fragments (may) cause problems if integrity of hi is important ?

```
1. hi = lo;              5. println(lo);
2. lo = hi;              6. println(hi);
3. lo = "1234";          7. readln(lo);
4. hi = "1234";          8. readln(hi);
```

Example Information Flow - *Integrity*

```
String hi; // high integrity (trusted) data
String lo; // low integrity (untrusted) data
```

Which program fragments (may) cause problems if integrity of hi is important ?

```
✗ 1. hi = lo;
✓ 2. lo = hi;
✓ 3. lo = "1234";
✓ 4. hi = "1234";
```

```
✓ 5. println(lo);
✓ 6. println(hi);
✓ 7. readln(lo);
✗ 8. readln(hi);
```

Duality

- Integrity and confidentiality are ***DUALS***

  if we "flip" everything in some property or an example for confidentiality, we get a corresponding property or example for integrity

  eg inputs are dangerous for integrity, outputs are dangerous for confidentiality

Some points to note

- Information flow properties are about ruling out unwanted influences/dependencies/interference/observations
- Note the difference between data flow properties and visibility modifiers (eg public, private) or, more generally, access control
  - it's not (just) about accessing data, but also about what you do with it

Questions

- what is information flow? (informally)
- how can we specify information flow policies?
- how can we enforce or check them?
  - dynamically (runtime)
  - statically (compile time) – by type systems
- what is the semantics (ie. meaning) of information flow formally?

Trickier examples for confidentiality

```
int hi; // security label secret
int lo; // security label public
```

Which program fragments (may) cause problems for confidentiality?

```
1. if (hi > 0) { lo = 99; }
2. if (lo > 0) { hi = 66; }
3. if (hi > 0) { print(lo);}
4. if (lo > 0) { print(hi);}
```

Trickier examples for confidentiality

```
int hi; // security label secret
int lo; // security label public
```

Which program fragments (may) cause problems for confidentiality?

```
1. if (hi > 0) { lo = 99; }
2. if (lo > 0) { hi = 66; }
3. if (hi > 0) { print(lo);}
4. if (lo > 0) { print(hi);}
```

indirect vs direct flows

- direct or explicit flows

  by "direct" assignment or leak

  eg `lo=hi;` or `println(hi);`

- indirect or implicit flows

  by indirect "influence"

  eg `if (hi > 0} { lo = 99; }`

Implicit flows can be partial, in the sense that not all info is revealed

Trickier examples for confidentiality

Example

```
int hi; // security label secret
int lo; // security label public
```

Which program fragments (may) cause problems for confidentiality?

```
1. while (hi>99) do {....};
2. while (lo>99) do {....};
3. a[hi] = 23; // where a is high/secret
4. a[hi] = 23; // where a is low/public
5. a[lo] = 23; // where a is high/secret
6. a[lo] = 23; // where a is low/public
```

Hidden channels

More subtle forms of indirect information flows can arise via hidden or covert channels, eg

- (non)termination

  eg `while (hi>99) do {....};`

  or `if (hi=99) then {"loop"} else {"terminate"}`

- execution time

  eg `for (i=0; i<hi; i++) {...};`

  or `if (hi=1234) then {...} else {...}`

- exceptions

  eg `a[i] = 23` may reveal length of a (if i is known),
  or leak info about i (if length of a is known),
  or reveal if a is null..

How can we *statically* enforce information flow policies by means of a type system?

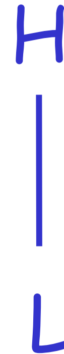
Type-based information flow

- Type systems have been proposed as way to restrict information flow.
  - most of the theoretical work considers confidentiality, but the same works for integrity

- Practical problem: often very (too) restrictive, because of difficulty in ruling out implicit flows

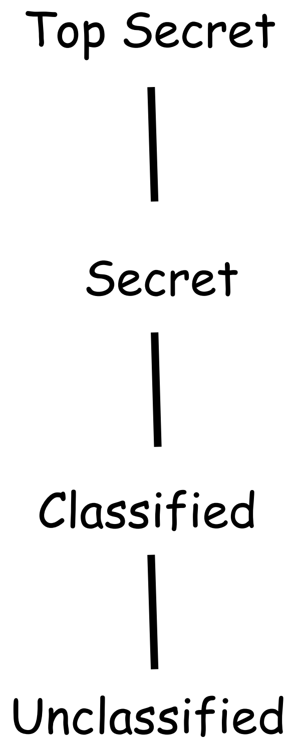
Types for information flow (confidentiality)

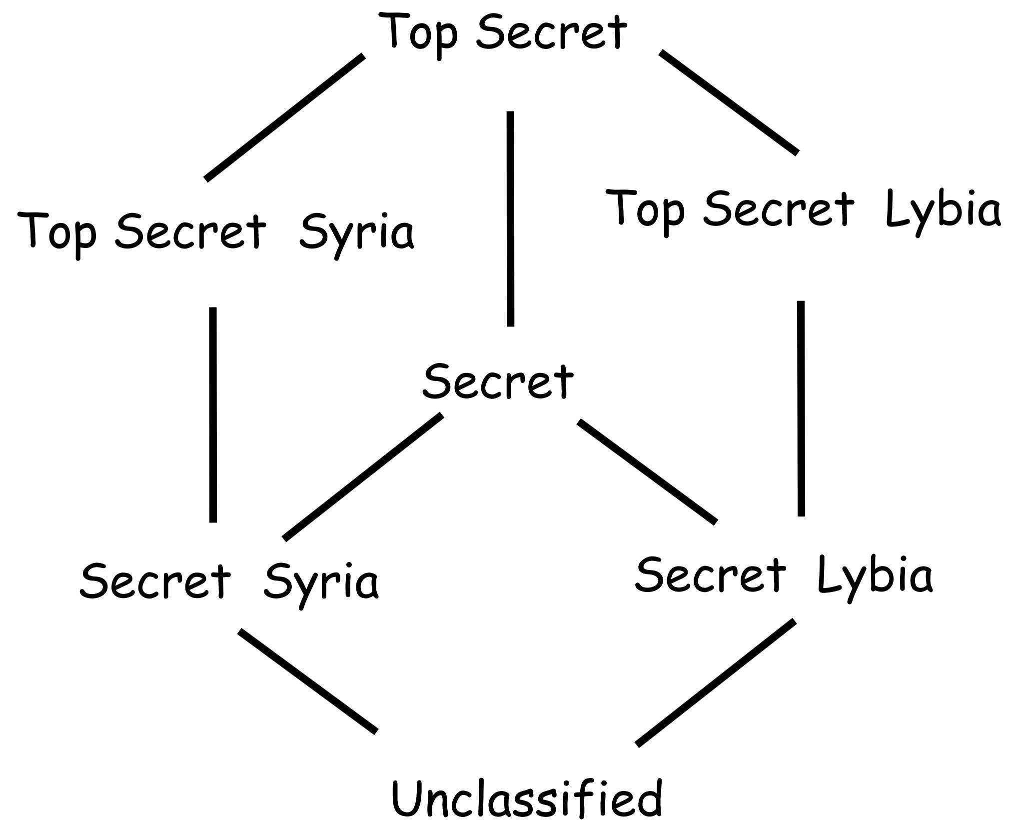
- We consider a lattice (Dutch: tralie) of different security levels
- For simplicity, just two levels
  - H(igh) or confidential, secret
  - L(ow) or public



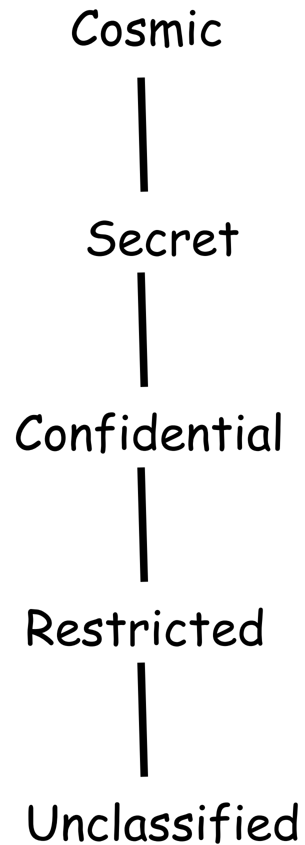
- Typing judgements $e{:}t$
  meaning e has type
- implicitly with respect to a context $x_1{:}t_1, \ldots x_n{:}t_n$ that gives levels of program variables

More complex lattices





NATO classification



Rules for expressions

$e : t$ means e contains information of level t or *lower*

- variable $x{:}t$ if x is a variable of type t
- operations
$$\frac{e{:}t \qquad e'{:}t}{e+e' : t}$$
for binary operation +
similar for n-ary
- subtyping
$$\frac{e{:}t \qquad t \leq t'}{e{:}t'}$$

where $L \leq H$

Rules for commands

s : ok t means *s only writes to level t or higher*

- assignment
$$\frac{e : t \qquad x : t}{x{:=}e \; : ok\ t}$$

- if-then-else
$$\frac{e : t \qquad c1 : ok\ t \qquad c2 : ok\ t}{\text{if } e \text{ then } c1 \text{ else } c2 : ok\ t}$$

subtyping
$$\frac{c : ok\ t \qquad t \geq t'}{c : ok\ t'}$$

ie. $ok\ t \leq ok\ t'$ iff $t \geq t'$ (anti-monotonicity)

Rules for commands

s : ok t means s only writes to level t or *higher*

- composition

$$\frac{c1 : ok\ t \qquad c2 : ok\ t}{c1;c2 : ok\ t}$$

- while

$$\frac{e : t \qquad c : ok\ t}{\text{while } e \text{ do } c : ok\ t}$$

Beware

Beware of the confusing difference in directions

e : t means e contains information of level t or *lower*

s : ok t means s only writes to level t or *higher*

For people familiar will Bell – LaPadula access control : there you have the same confusion, in the "no read up" & "no write down" rules

How can we be sure that such type systems are “correct”?

Soundness vs Completeness

- soundness of the type system:

  programs that are well-typed do no leak
- completeness of the type system:

  programs that do not leak can be typed

Is the type system on preceding slides

- *sound?*
- *complete?*

How can we determine this?

Counterexamples for completeness

It is easy to give examples that are not typable but do not leak, eg

- `if (false) then { lo = hi; }`
- `lo = hi + 1 - hi;`
- `lo = hi; lo = 12;`

Soundness

- Is this type system sound?
  - ie does is prevent the information flows that we want to prevent
- How do we define what we want to prevent?
  - Recall the tricky examples of implicit flows
- This is commonly done using notions of non-interference, which try to capture the notion of what can be observed

Non-interference gives a precise semantics for what "information flow" means

Soundness wrt non-interference

Definition For memories (or program states) **μ** and **v**, we write
$\mu \approx_L v$ iff **μ** and **v** agree on low variables.

Definition (Non-interference)

A program *C* does not leak information if, for all $\mu \approx_L v$:

if executing *C* in **μ** terminates and results in **μ**',

and executing *C* in **v** terminates and results in **v**',

then $\mu' \approx_L v'$

Theorem (Soundness)

if *C* : ok t then *C* does not leak information

Termination as covert channel?

termination-*in*sensitive

Definition (Non-interference)

A program C does not leak information if, for all $\mu \approx_L \nu$:

if executing C in μ terminates and results in μ',

and executing C in ν terminates and results in ν',

then $\mu' \approx_L \nu'$

Does this rule out (non) termination as hidden channel (as observation to distinguish two runs)?

Definition (Termination-sensitive non-interference)

A program C does not leak information if, for all $\mu \approx_L \nu$:

if executing C in μ terminates in μ',

then executing C in ν also terminates, and results in some ν' with $\mu' \approx_L \nu'$

While-rule for termination-sensitive non-interference

The while-rule

$$\frac{e : t \qquad c : ok\ t}{\text{while } e \text{ do } c : ok\ t}$$

does *not* rule out non-termination as covert channel

A more restrictive rule

$$\frac{e : L \qquad c : ok\ L}{\text{while } e \text{ do } c : ok\ L}$$

does rule this out.
(How? NB this is very restrictive!)

- A similar change needed for in-then-else rule.

Other notions of secure information flow

Other definitions of what it means to be secure (in the sense of non-leaking) are needed if

- if programs can throw exceptions
  - exceptions are another covert channel, just like non-termination
- if programs are multi-threaded or non-determinisitic
  - because execution of a program can then result in several outcomes
    - multi-threaded programs are non-deterministic, because results can depend on scheduling

Information flow for non-deterministic programs

Definition (Possibilistic NI)
A non-deterministic program C does not leak information if
for all $\mu \approx_L \nu$
if executing C in μ terminates in μ',
then executing C in ν *can* in some ν' with $\mu' \approx_L \nu'$

This still ignores probabilistic information flows, for which one would take the chance that c does terminate in some ν' with $\mu' \approx_L \nu'$ into account

- Running the program multiple times, you might be able to observe something

.

The problem with secure information flow

- *Practical* problem with secure information flow: the extreme restrictions it imposes, esp. when it come to ruling out implicit flows
  - eg no while loop with a high guard
  - note that login program leaks information about the password

- More generally, some way of allowing forms of declassification is needed in practice

Declassification

More *permissive* forms of information flow can allow de-classification, eg

- for confidentiality:
  - output of encryption operation is labelled as public, even though it depends on secret data.
- for integrity:
  - output of input validation routine may be trusted, even though it depends on untrusted data
  - output of routine that checks digital signature may be trusted, even though it depends on untrusted data

Information Flow in "practice"- *static* enforcement

- Many code analysis tools perform some data flow analysis
  - Eg to spot SQL injection problems (as RIPS does)
  - Recall PREfast did this, but only intra-procedural
  - NB typically for integrity, not confidentiality
  - Often unsound/incomplete, as concession to practicality

- JFlow/Jif is a type system for expressing & enforcing information flow properties for Java
  - using it requires a serious effort!

Information Flow in "practice"- *dynamic* enforcement

- Perl has an *runtime monitoring* of information flow properties (again for integrity properties) using tainting

- Detecting exploits at operating system level
  (eg. worms or viruses that use classic buffer overflows)
  Approach:
  1. taint user input,
  2. trace this during execution,
  3. warn if tainted input ends up on
     - the instruction register or program counter of CPU
     - in a function pointer
     - ...

  This can detect zero-day exploits, and be used to prove that something is an exploit. But is kills performance...

Information Flow in "practice"

Pragmatic approaches typically worry less – if at all - about implicit flows.

Indeed, are implicit flows an issue for integrity?

Related work: Bell-La Padula

- Classic Bell-La Padula system access control combines
  - Mandatory Access control (MAC)
  - Multi-Level Security (MLS)

  and protects information flow between files by rules
  - no read up
  - no write down
    - *nb similarity with our typing rules* but for processes accessing files, instead of a programs accessing variables, and enforced at runtime instead of compile time
- Bell-LaPaluda was developed in the 70s for access control in military applications

Summary

- What is information flow (informally)?
  - explicit flows , implicit flows, covert channels
- How can we *statically* control information flow, using type systems?
- How can we formally define what information flow is?

  non-interference

  termination sensitive vs termination insensitive

You can read all this in Chapter 5 of the lecture notes