Software Security
Information Flow
(Chapter 5 of the lecture notes)

Erik Poll
Digital Security group
Radboud University Nijmegen

Motivating 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 should be leaked to google *only*
- credit card info should be leaked to hotel *only*

Such information flow policies are an interesting class of security policies

Motivating example

Suppose that for our mobile phone app we want to enforce

- location should be leaked to google *only*
- credit card info should be leaked to hotel *only*

- *Can OS access control on the app prevent these flows?*

  NO! Access control can give or deny an app access to some information or service, but cannot restrict what the app does with it.

- *More generally, could we enforce this at runtime by monitoring the inputs & outputs of the application?*

  NO! Unless track the information *inside* *the app with dynamic taint tracking.*

  - Recall PREfast supported static taint tracking – clumsily – also inside the code

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 data)
  - information flow is *also* about *what you do with data after you accessed it*

    (eg how you process & forward location data)

- **Warning: possible exam questions coming up!**

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 - 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. lo = some_function_call(hi);
2. lo = encrypt(hi,AESkey);
```

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);
```

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 = some_function_call(lo);
2. hi = convertToUpperCase(lo);
3. hi = HTMLencode(lo);
4. hi = checkAndStripMAC(lo);
   // where MAC is MessageAuthenticationCode
```

Duality between integrity & confidentiality

Integrity and confidentiality are *duals* :

if you "flip" everything in a property or example for confidentiality,

you get a corresponding property or example for integrity

For example

inputs are dangerous for integrity,
outputs are dangerous for confidentiality

Information flow

- **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 do we mean by 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);}
```

implicit
aka
indirect flows

indirect vs direct flows

There are (at least) two kinds of information flows

- direct aka explicit flows

  by "direct" assignment or leak

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

- indirect aka implicit flows

  by indirect "influence"

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

Implicit flows can be partial, ie leak *some* but not *all* info

Eg the example above only leaks the sign of `hi`, not its value.

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
```

Trickier examples for confidentiality

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

✗ 1. `while (hi>99) do {....};`
// timing or termination may reveal if hi > 99

✓ 2. `while (lo>99) do {....};` // no problem

✗ 3. `a[hi] = 23; // where a is high/secret`
// exception may reveal if hi is negative

✗ 4. `a[hi] = 23; // where a is low/public`
// contents of a may reveal value of hi and, again,
// exception may reveal if hi is negative

✗ 5. `a[lo] = 23; // where a is high/secret`
// exception may reveal the length of a, which may be secret

✓ 6. `a[lo] = 23; // where a is low/public - no` problem

Hidden channels

More subtle forms of indirect information flows can arise via
hidden channel aka covert channels aka side channels

- (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),

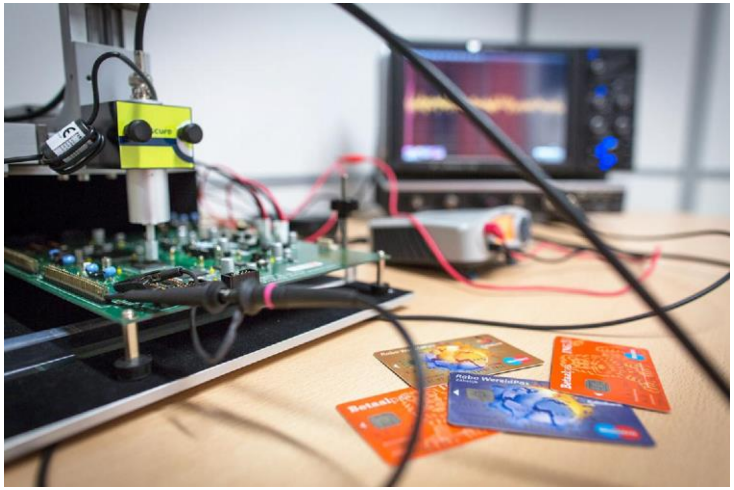
  or reveal if a is null..

Hidden channels

- Apart from timing & terminations, there are many more side-channels:
  - noise
  - power consumption
  - EM radiation – aka TEMPEST attacks

- In the courses Hardware Security and Cryptographic Engineering you can find out more about hidden channels

- In our lab we have set-ups for power analysis & EM radiation



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.

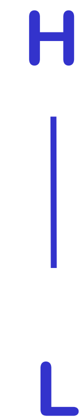
- most of the theoretical work considers confidentiality, but the same works for integrity

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

Types for information flow (confidentiality)

- We consider a lattice (in Dutch: tralie) of different security levels

- For simplicity, just two levels
  - H(igh) or confidential, secret
  - L(ow) or public

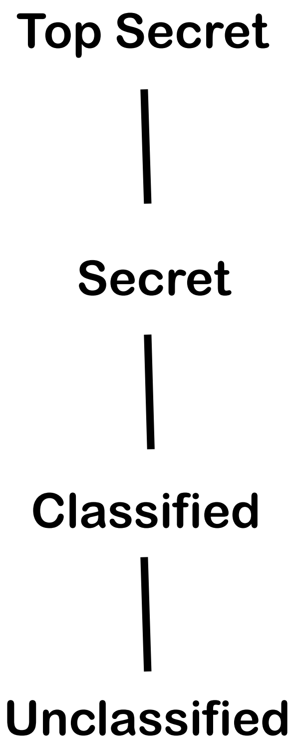


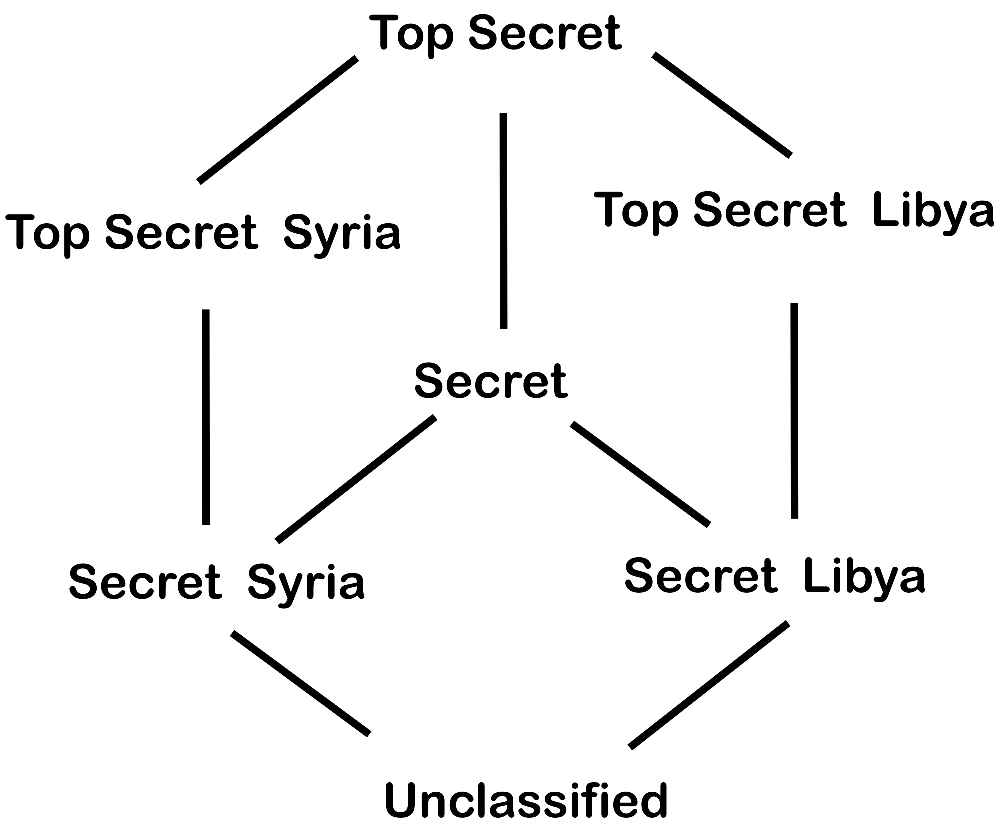
- Typing judgements e:t

  meaning e has type t

- implicitly with respect to a context x_1:t_1, ... x_n:t_n that gives levels of program variables

More complex lattices





NATO classification

Cosmic

Secret

Confidential

Restricted

Unclassified

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 $\dfrac{e{:}t \quad e'{:}t}{e+e' : t}$ for some binary operation + (similar for n-ary)
- subtyping $\dfrac{e{:}t \quad t \leq t'}{e{:}t'}$

Rules for commands

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

- assignment
$$\frac{e : t \qquad x \text{ is a variable of type } t}{x := e \; : \text{ok } t}$$

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

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

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

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

- while

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

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 and 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 ν,
we write $\mu \approx_L \nu$ iff μ and ν agree on low variables.

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 $\mu' \approx_L \nu'$

Theorem (Soundness)
if C : ok t then C does not leak information

Termination as covert channel?

<u>Definition</u> (Non-interference) termination-*in*sensitive

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)?

<u>Definition</u> (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

<u>Definition</u> (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* terminate in some ν' with $\mu' \approx_L \nu'$

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

- **At attacker that can run the program multiple times, 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 inevitably leaks information about the password

- For most practical applications, we need a looser notion of information flow than non-interference
  - Some controlled form of declassification

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

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

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

Information Flow in practice

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

- **Indeed, are implicit flows an issue for integrity?**
  - **for confidentialy implicit flows can clearly be dangerous, for integrity this is not so clear**

Related work: Bell-La Padula

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

  and protects information flow between files by the rules
  1. no read up
  2. no write down

- Note the similarity with our typing rules, but the rules are for processes accessing files, instead of programs accessing variables, and enforced at runtime instead of compile time
- Bell-LaPaluda was developed in the 70s for access control in military applications
- The dual Biba model has been proposed for integrity

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 or termination-insensitive

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

- Next week: static information flow analysis for Android using extension of Java

Possible exam questions

- **Explaining if there is unwanted information for integrity or confidentiality in example programs**

  (like those on slides 6-11, 15, 17)
- **Giving and/or motivating a typing rule for information flow typing (like on slides 27-29 or 37), for termination-sensitive or insensitive**
- **Giving and/or explaining the definition of non-interference, for integrity or confidentiality**

  (but not the possibilistic & probabilistic versions)