Software Security

Information Flow for Android Apps

Erik Poll

Digital Security group

Radboud University Nijmegen

Today

1. Possibilities to make the type system of Java richer, ie. more expressive
   - using Java annotations on types

2. Using this possibility to add types for information flow
   - for Android apps

Recap: typing

Recall from earlier lectures & lecture notes (Chapter 3):

- Typing is a great way to prevent certain kinds of bugs
  - by making certain kinds of bugs *less likely*
- Type-safety (aka type-soundness) makes this even better
  - by making make certain kinds of bugs *impossible*
  - by eliminating some weird behaviour that attackers may exploit
- Sound type system can provide the basis for more language-level safety guarantees, such as
  - visibility restrictions (using eg `private` fields)
  - sand-boxing (ie. code-based access control inside program)
  - immutable objects
  - ...

Recap: typing for information flow

Recall from last week & lecture notes (Chapter 5):

- Typing can be used to track information flows between several levels
  - using a lattice of security levels
  - for integrity or for confidentiality
- Such type systems can be overly restrictive
  - Preventing implicit flows is tricky

```
if (... hi ...) { lo = ... } ; // not allowed
```

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

  - Preventing termination-leaks or timing-leaks even more so

```
if (... hi ...) { ... } ; // not allowed
```

Java Annotations on Types

&

Java’s type system is too weak?

Java’s type system can catch certain errors at compile-time

```
boolean b = 2.0 + "hello";
```

but not all

```
System.console().readLine();

int i = a[4];

rs = stmt.executeQuery(query);
```

possible NullPointer

possible NullPointer or IndexOutOfBounds

possible SQL injection?

Java types not expressive enough?

```
int doubleLength(String s)
      {return s.length * 2; }
...
int j = doubleLength(null);
```

Java's type system can't spot the null-pointer problem in this code

A human reader of the code can spot this problem.

Whose fault is this NullpointerException?

- **programmer who wrote the code for `doubleLength` ?**
- **or: programmer who calls `doubleLength` with a `null` argument ?**

Type annotations

Programmer can enrich Java's type system with annotations of the form

```
@SomeAnnotation
@SomeAnnotationWithElements(colour = "blue", age = "18")
@SomeAnnotationWithOneElement("blue")
```

that can be added to source code

- on declarations, eg of classes, fields, methods, ...

```
public @Colour("green") class Grass {...}
private @Colour("red") int i;
```

- on uses of types (since Java 8)

```
new @Colour("green") List(...);
```

Annotation example: @NonNull and @Nullable

```
int doubleLength(@NonNull String s)
   {return s.length * 2; }
 ...
 int j = doubleLength(null);  // this is wrong!
```

Annotation example: @NonNull and @Nullable

```
int doubleLength(@Nullable String s)
   {return s.length * 2; }    // this is wrong!
...
int j = doubleLength(null);
```

Moral of the story: even *without any tool* support, custom annotations can help to

- **clarify assumptions & guarantees (and hence) assign blame**
- **help humans to spot bugs**

Tool support (by type checker) could automate this, of course.

Fancier examples

You can combine type annotations with generics, eg

```
@NonEmpty List<@NonNull String>
```

Warning: annotations on array types can be hard to read, eg

```
@NonNull String []
```

What could this mean?

a non-null array of strings

or: an array of non-nulls strings?

These are the sort of things you have to look up with the language manual.

Type annotations & pluggable type systems

Why use annotations?

1. Annotations can simply be informal documentation to help the programmer

2. Annotations can be used to help static analysis tools

3. Annotations on types can be used as 'real' types, to improve (or *refine*) Java's type system, if we have a type checker for them.

   - Effectively, this is a special form of 2

Annotation example: ensuring encrypted information

```
void send(@Encrypted String msg) {...}
      // So send() expects an @Encrypted string
@Encrypted String encrypt(String s, Key k) {...}
      // So encrypt() produces an @Encrypted string
...
@Encrypted String msg1 = ...;
send(msg1); // OK?
String msg2 = ....;
send(msg2); // OK?
send(encrypt(msg2,key)); // OK?
```

Annotation example: ensuring encrypted information

```
void send(@Encrypted String msg) {...}
     // So send() expects an @Encrypted string
@Encrypted String encrypt(String s, Key k) {...}
     // So encrypt() produces an @Encrypted string
...
@Encrypted String msg1 = ...;
send(msg1); // OK!
String msg2 = ....;
send(msg2); // Warning!
send(encrypt(msg2,key)); // OK!
```

Moral of the story: we can use custom annotations to help prevent certain categories of flaws

Annotation example: ensuring encrypted information

The (one line!) definition of a typechecker for `@Encrypted` annotations using the Checker framework

```
@Target(ElementType.TYPE_USE)
  @SubtypeOf(Unqualified.class)
    public @interface Encrypted {}
```

SPARTA:
Static Program Analysis
for Reliable Trusted Apps

Type system for information flow in Java apps

Collaborative Verification of Information Flow for a High-Assurance App Store

by

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu

CCS 2014

This paper presents SPARTA
(Static Program Analysis for Reliable Trusted Apps)

More info at http://www.cs.washington.edu/sparta

Context: current generation of app stores

- App stores have some approval process
- They have to approve hundreds of apps per day
- Problem: all app stores have approved malware
- Current best practice: remove malware when it is reported

Security worries in apps

Malware can

1. steal user information (location, installed apps, ..)
2. steal user credentials (passwords, ...)
3. make premium calls or send expensive SMS
4. send SMS advertising
5. improve website rankings in search engine results
6. do some purposeless destruction
7. ransomware

SPARTA can prevent 1-4

- but not phishing as way to steal credentials, which is also a form of 2

Shubham Ghodke
Fake app do not install. It makes your phone full of ads

Mimita Pujols
No good, *Adds from the beginning. What happened to the REAL app?*

Fake WhatsApp app in Google Playstore in Nov. 2017, with > 1 million installs

Better app stores offering higher assurance level

- Could a specialised app store provide higher level of assurance?

  Eg for special categories of apps or users, such as

  - financial or medical apps,
  - corporate or government users

- Could there be a business model in this?

  To make extra effort commercially viable or even profitable.

- *Bottlenecks:*

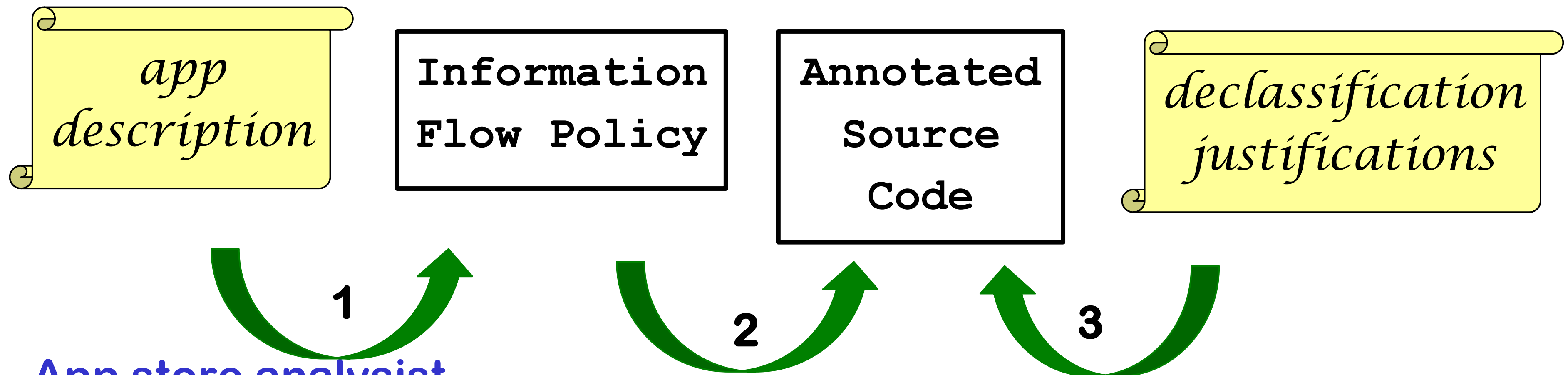
  1. *what to check ?*
  2. *how to check?*
  3. *can this be done at reasonable cost (time & effort)?*

SPARTA

- Security type system for Android apps
  - to guarantee information flow policies,
    that rule out unwanted information leaks
- Java annotations used to annotate code
- Checker framework is used to type check these
  - *but* some manual checks for declassification,
    incl. manual checks for implicit flows in conditional statements
- Collaborative verification, where
  1. code developer does some work by adding annotations
  2. human verifier runs checker & performs manual checks

Collaborative verification model

- **Developer** provides



- **App store analysist**

  1. checks if information flow policy is acceptable (manually)
  2. runs the type checker
  3. checks the declassifications (manually)

What to check? Information flow policies!

The target: preventing malware with unwanted information flows

Information flow policies specified using sources and sinks,
where information comes from or goes to

Example sources

- camera
- location information
- SMS reading
- the file system

Example sinks

- the display
- the internet
- SMS sending
- the file system

- Many sources and sinks already occur as Android permissions
- Some things can be both source and sink, eg. the file system

Android permissions vs information flow policies

- An app can have Android permission to access
  - location information
  - internet connection
  - camera
  - file system
- As an information flow policy, the app could *only* have permission to
  - save camera image to disk (ie not to send it over the internet)
  - save location to disk (eg, to save location with a photo)
  - download updates over the internet connection

  This is much more fine-grained! But maybe still not perfect…

As discussed last week, information flow policies are more expressive than conventional access control policies.

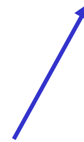
Example information flow policies

```
READ_SMS     -> DISPLAY

USER_INPUT   -> CALL_PHONE

CAMERA       -> DISPLAY, DATA

LOCATION     -> INTERNET(maps.google.nl)
```



Sources and sinks may be parameterised.

Notation: -> is also written as ! in the paper

Transitivity & white-washing

Transitive flows must be explicitly specified.

So the policy

```
CAMERA       -> FILESYSTEM
FILESYSTEM   -> INTERNET
```

must also include

```
CAMERA       -> INTERNET
```

if photes are allowed to be sent over the internet.

Idea: make sure an app cannot whitewash data via file system if this was not explicitly intended.

Parameters could also rule out such issues, eg

```
               CAMERA      -> FILESYSTEM("images/*")
   FILESYSTEM("config/*")  -> INTERNET
```

Information flow types: sources and sinks

`@Source` Where might this info come from?

`@Sink` Where might this info go to?

These type annotations take a parameter (or *element*, in Java terminology) and are then applied to variables or parameters.

For example

`@Source(CAMERA)` – this info comes, or *might* come, from the camera

`@Source(LOCATION)` – this info may be location information

`@Sink(INTERNET)` – this info may be sent over the internet

`@Source(INTERNET,CAMERA)` – this info may come from camera or internet

Example type annotations

Suppose the Android API includes methods

```
public static void sendToInternet(String s);

public static String readGPS();
```

What would be good annotations of these methods ?

```
public static void sendToInternet(@Sink(INTERNET) s);

public static @Source(LOCATION)String readGPS();
```

Example typings

Given the API methods

```
void sendToInternet(@Sink(INTERNET) String message);
@Source(LOCATION)String readGPS();
```

What would be a correct annotation of the app code below?

```
String loc = readGPS();
sendToInternet(loc);
```

Example typings

API annotations are given and trusted

Given the API methods

```
void sendToInternet(@Sink(INTERNET) String message);
@Source(LOCATION)String readGPS();
```

What would be a correct annotation of the app code below?

```
@Source(LOCATION)@Sink(INTERNET) String loc = readGPS();
// loc may receive LOCATION info
//      and may be sent over internet
sendToInternet(loc);
```

app annotations are to be provided by app developer and are not trusted

This code is only acceptable if the policy includes `LOCATION->INTERNET`

Example typings

Which of these annotations would be rejected by the type checker?

1.
```
@Source(LOCATION) String loc = readGPS();
sendToInternet(loc);
```
- Not type correct, because in 2nd line `loc` cannot be sent over internet

2.
```
@Sink(INTERNET) String loc = readGPS();
sendToInternet(loc);
```
- Not type correct, because in 1st line `loc` can't store location information

3.
```
String loc = readGPS();
sendToInternet(loc);
```
- Not type correct, because of same reasons as 1 and 2

4.
```
@Source(LOCATION)@Sink(INTERNET) String loc = readGPS();
sendToInternet(loc);
```

- Type correct, but does require policy includes `LOCATION->INTERNET`

Moral of the story: programmers have to get annotations right to make their code typecheck, and cannot cheat!

Is this code ok?

```
@Source(SMS) String s = ... ;
@Source(SMS,INTERNET) String t = s;
```

Yes, as `@Source(SMS)` is a subset of `@Source(SMS,INTERNET)`

```
@Source(SMS) @Sink(SMS,INTERNET) String msg1 = ...;
@Source(SMS,INTERNET) @Sink(SMS) String msg2 = msg1;
```

Yes, as `@Sink(SMS)` is a subset of `@Sink(SMS,INTERNET)`

Beware: with aliasing between mutable data structures you have to be careful!

Eg having two references to the same `char[]` *with different annotations, say* `@Sink(SMS,INTERNET)` *and* `@Sink(SMS)`*, would cause unsoundness*

Subtyping

There is a natural subtyping relation on types.

For example,

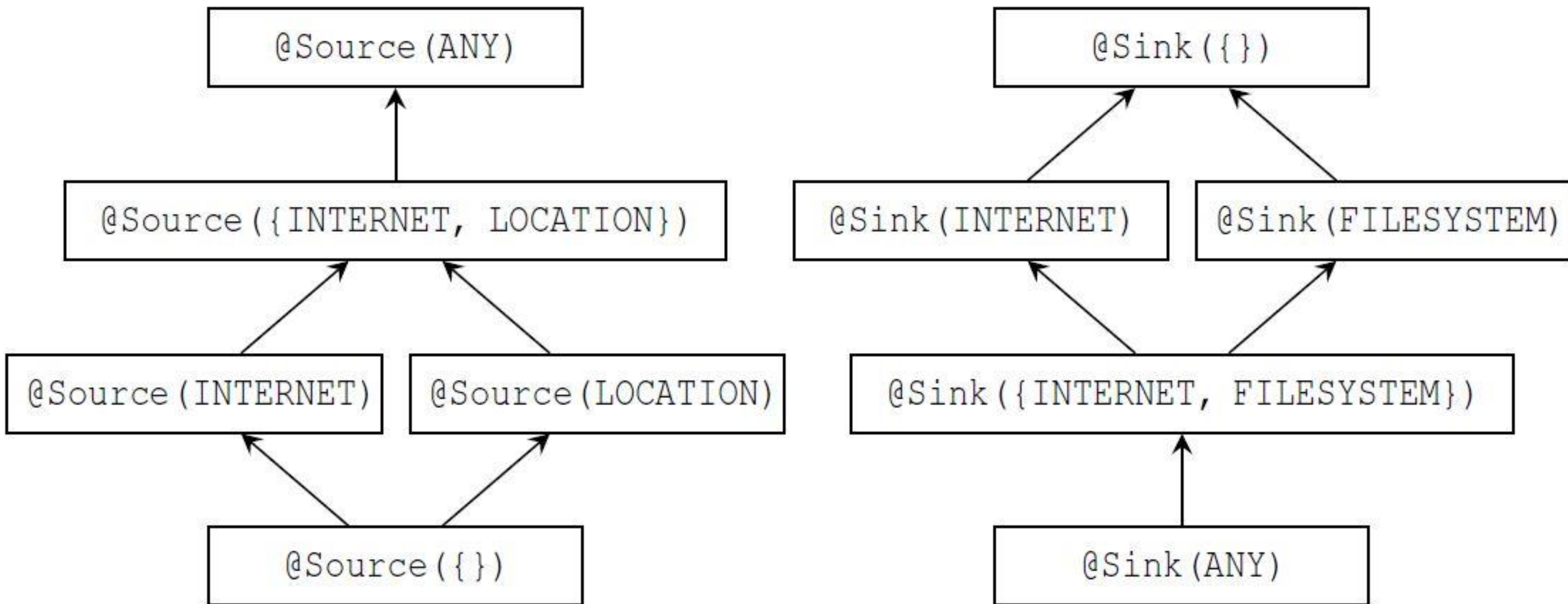
`@Source(SMS)` is a subtype of `@Source(SMS,INTERNET)`

`@Sink(SMS,INTERNET)` is a subtype of `@Sink(SMS)`

Note the opposite direction of the subtype relation for Sources and Sinks.

- Recall: we also saw this duality in type systems for information flow
  for reading information of some level
  vs writing information to a variable of some level

The subtype relation forms a lattice



`@Source(ANY) = @Source{LOCATION, INTERNET, SMS, CAMERA, ...}`

Subtyping

The subtype relation gives rise to a subtyping rule in the type system.

- **Eg, if**

```
@Source(SMS) String s;
```

then `s` also has type `@Source(SMS,INTERNET)`

Recall subtyping rule (aka subsumption) from last week & lecture notes

$$\frac{e : t \qquad t \leq t'}{e : t'}$$

Quiz: is this app code ok & does it meet policy?

App code:

```
@Source(LOCATION)@Sink(INTERNET)String loc = readGPS();

sendToInternet(loc);
```

Policy:

```
LOCATION -> INTERNET
```

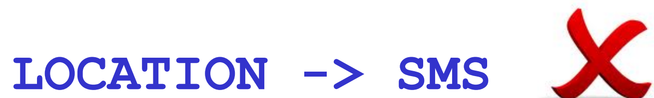
Quiz: is this app code ok & does it meet policy?

App code:

```
@Source(LOCATION)@Sink(INTERNET)String loc = readGPS();

sendToInternet(loc);
```

Policy:



LOCATION -> SMS,INTERNET

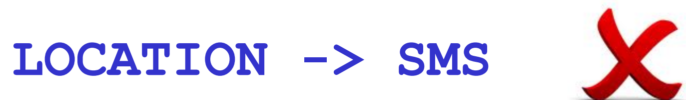
Quiz: is this app code ok & does it meet policy?

App code:

```
@Source(LOCATION)@Sink(SMS)String loc = readGPS();

sendToInternet(loc);
```

Policy:

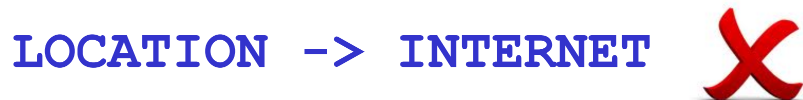


Quiz: is this app code ok & does it meet policy?

App code:

```
@Source(LOCATION)@Sink(SMS)String loc = readGPS();
sendToInternet(loc);
```

Policy:



The code does meet the policy, but the app developer screwed up the annotations, so the type checker will complain!

The tricky cases...

Problematic cases

- The SPARTA type system is overly restrictive
  - Some ‘legal’ programs (which do not violate the policy) will be rejected
    - ie. there are false positives

- Solution to this:
  - The type system provides explicit loopholes for this
  - Any use of these loopholes will have to be manually verified

Problem 1: heterogeneous arrays

```
String[] a;

a[0] = readGPS();

a[1] = readSMS();
```

What would be a good annotation of the code above, using the parameter `LOCATION` and `SMS`?

Problem 1: heterogeneous arrays

```
@Source({LOCATION, SMS}) String[] a;

a[0] = readGPS();

a[1] = readSMS();
```

What would be a good annotation of the code above, using the parameter `LOCATION` and `SMS`?

This is not the most accurate description, we 'lose' some information, namely that the two array elements have different types.

The annotation system is not expressive enough to talk about such heterogenous arrays.

Problem 1: heterogeneous arrays

```
@Source({LOCATION, SMS}) String[] a;

a[0] = readGPS();

a[1] = readSMS();

String loc = array[0];
```

What would be a good annotation of the code above?

Problem 1: heterogeneous arrays

```
@Source({LOCATION, SMS}) String[] a;

a[0] = readGPS();

a[1] = readSMS();

@Source({LOCATION, SMS})String loc = array[0];
```

We would like to be more precise and write

```
@Source(LOCATION) String loc = array[0];
```

but then the type checker will complain, even though this complaint is a false positive

- *as this declassification is ok*

Problem 1: heterogeneous arrays

```
@Source({LOCATION, SMS}) String[] a;

a[0] = readGPS();

a[1] = readSMS();

@SuppressWarnings("flow") // Always returns location data

@Source({LOCATION}) String loc = array[0];
```

App developer can use this to surpress false positives.

But the human verifier will have to manually verify these! The

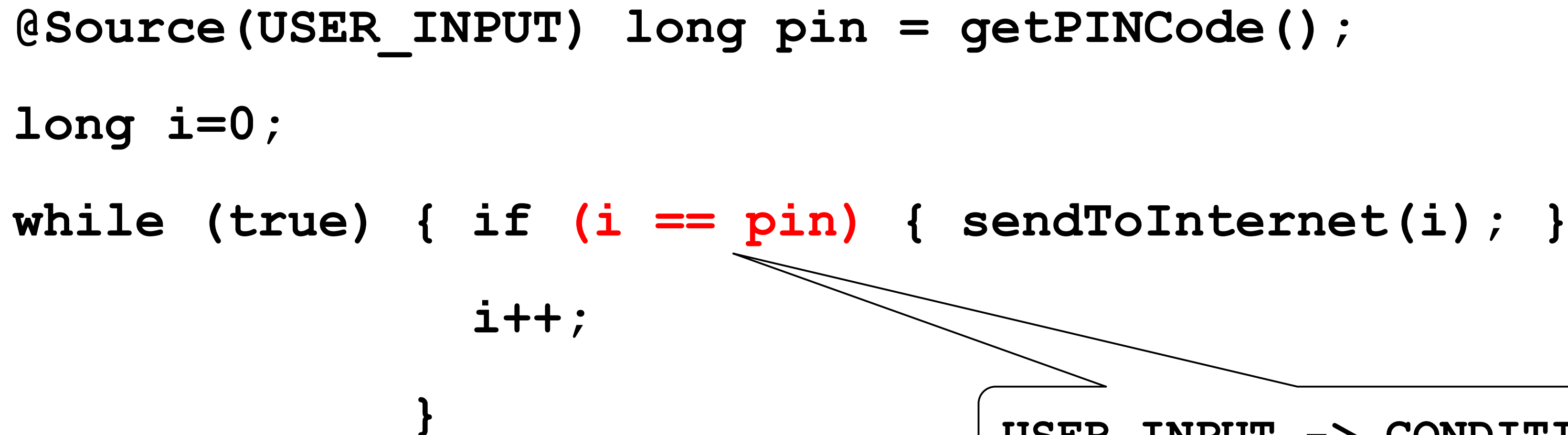
Problem 2: Dealing with implicit flows

```
@Source(USER_INPUT) long pin = getPINCode();
long i=0;
while (true) { if (i == pin) { sendToInternet(i); }
               i++;
             }
```

Possible approaches

1. **Ignoring implicit flows: this would be unsound, and allow leaking of the PIN code**
2. **Classic, sound approach, as in lecture notes: inside if-statement you can only send stuff over the internet if all variables used in the guard can be sent over the internet. This becomes *very* restrictive!**
3. **Solution used in SPARTA: introduce a new sink `CONDITIONAL`**

Problem 2: Dealing with implicit flows

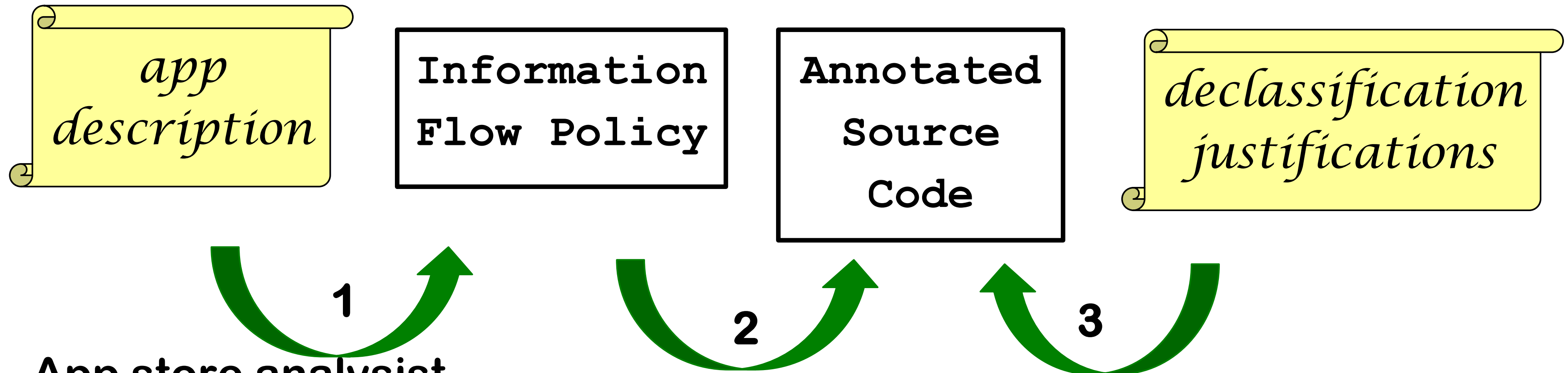


SPARTA’s approach to implicit flows

- New sink CONDITIONAL
- Flows to CONDITIONAL if classified information is used in a condition
- Type checker will warn about these
- Human verifier will have to check these

Overview: SPARTA's collaborative verification model

- Developer provides



- App store analysist

1. checks if information flow policy is acceptable
2. runs the type checker
3. manually checks the declassifications

Trusted Computing Base (TCB)

What is in the Trusted Computing Base? And what not?

- the Android OS, incl the Java Virtual Machine
- the type checker for annotations
- the Java compiler & byte code verifier
- the annotations provided for the APIs
- the annotations provided by the app developer
- the human verifier

Trusted Computing Base (TCB)

What is in the Trusted Computing Base? And what not?

- **the Android OS, incl the Java Virtual Machine YES**
- **the type checker for annotations YES**
- **the Java compiler & byte code verifier YES**
- **the annotations provided for the APIs YES**
- ***the annotations provided by the app developer NO***
- **the human verifier YES**

Trickier case: generics

- Annotations interact with generics in the obvious way, eg

```
List<@Source(USER_INPUT) @Sink(SMS) String> myList;
```

Reading these can get tricky…

Trickier case: polymorphic functions

How to annotate a function such as

```
static char[] stringToCharArray(String s) ?
```

This function is polymorphic: it preserves any type annotation on the input.

So

1. if `x` is `@Sink(SMS)` then `stringToCharArray(x)` is `@Sink(SMS)`
2. if `x` is `@Sink(INTERNET)` then `stringToCharArray(x)` is `@Sink(INTERNET)`

But we cannot a single annotation that covers both 1 and 2...

Solution: in annotations you can quantify over all type annotations, using a special type annotation variable `@PolySource`

```
@PolySource char[] stringToCharArray(@PolySource String s)
```

Note we had the same problem with tainting annotation for `array_copy` with PREfast

Relation with information flow by source code scanners

The more ad-hoc data flow analysis done by RIPS (for tainted input data, ie for integrity instead of confidentiality) runs into similar issues as here, eg

- **can information be white-washed by storing and then reading it from the file system?**
- **how can we track information flows through arrays?**
- **how do we deal with implicit flows?**
- **how do we deal with the API, and operations such as `stringToCharArray, subString, toUpperCase, ...` ?**

These are fundamental complications in *any* information flow analysis!

Case study with SPARTA (see paper)

- Analysis of 72 apps written by Red Team
- (Relatively low) annotation burden: 6 annotations/100 loc
- Auditing (ie human verifier) burden: 30 minutes/ kloc
  - but is this acceptable for several Mbytes of code?
- 96% of information flow related malware found
  (It's hard to find in the paper what the problem with the remaining 4% is, but it is claimed that extensions discussed in 2.10 would fix them)
- This was 82% of all malware in these apps, as some malware behaviour was not about unwanted information flow

To read

Collaborative Verification of Information Flow for a High-Assurance App Store

by

Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu

CCS 2014

See link on course webpage.