Software Security

Information Flow for Android Apps

Erik Poll

Digital Security group

Radboud University Nijmegen

Last week

- A type system for information flow
  - for a toy language
- Non-interference as formal definition of information flow
  - to prove soundness of such a type system

This week

- Enforcing information flow policies for Android apps
  - using such a type system for information flow

Type system for information flow in Java apps

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

by

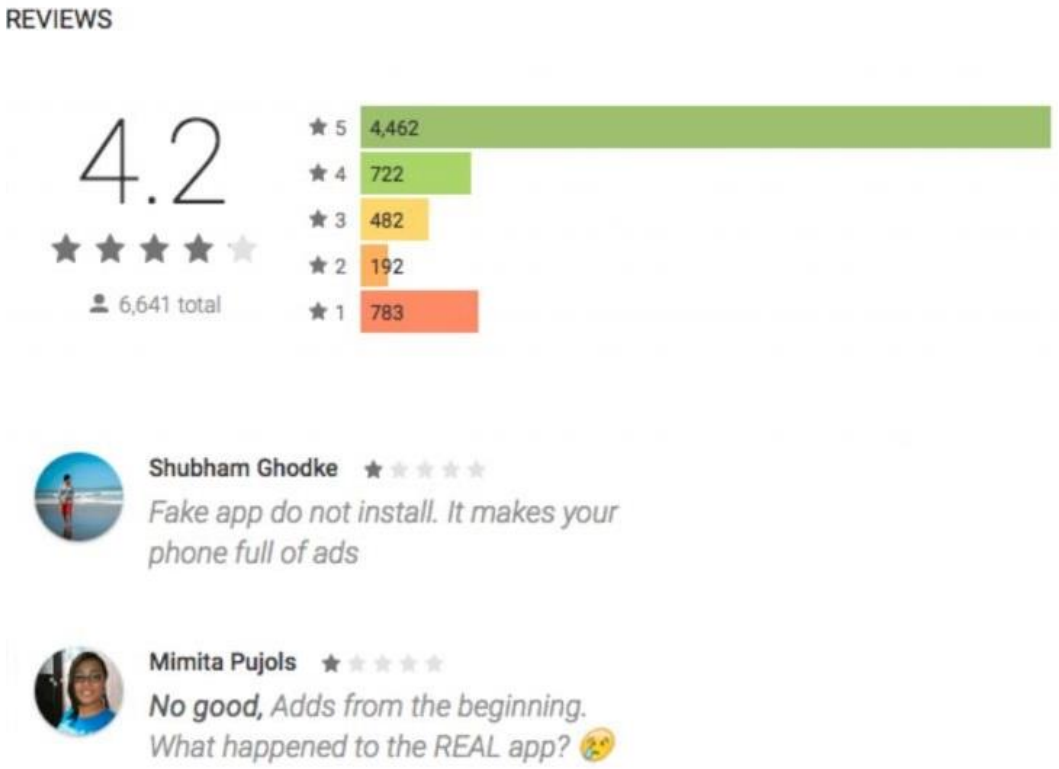
Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu

CCS 2014

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

Context: 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 practices:
  - run static analysis tool to spot suspicious apps
    - which should not have many (any?) false positives
  - remove malware when it is reported



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

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
   - forms of SEO (Search Engine Optimisation) that search engines disapprove of
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

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:*

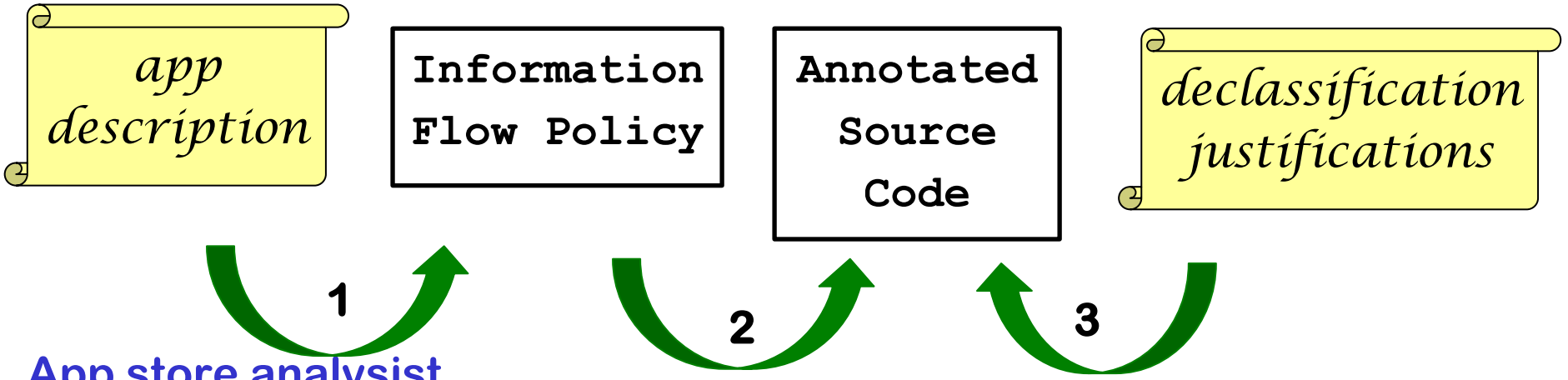
  1. *what to check ?*
  2. *how to check?*
  3. *can this be done at reasonable cost (time & effort)?*

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 two weeks ago, 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?

Does the policy

```
CAMERA      -> FILESYSTEM

FILESYSTEM  -> INTERNET
```

also allow

```
CAMERA      -> INTERNET     ?
```

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) String 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)`

Subtyping

There is a natural subtyping relation on types.

For example,

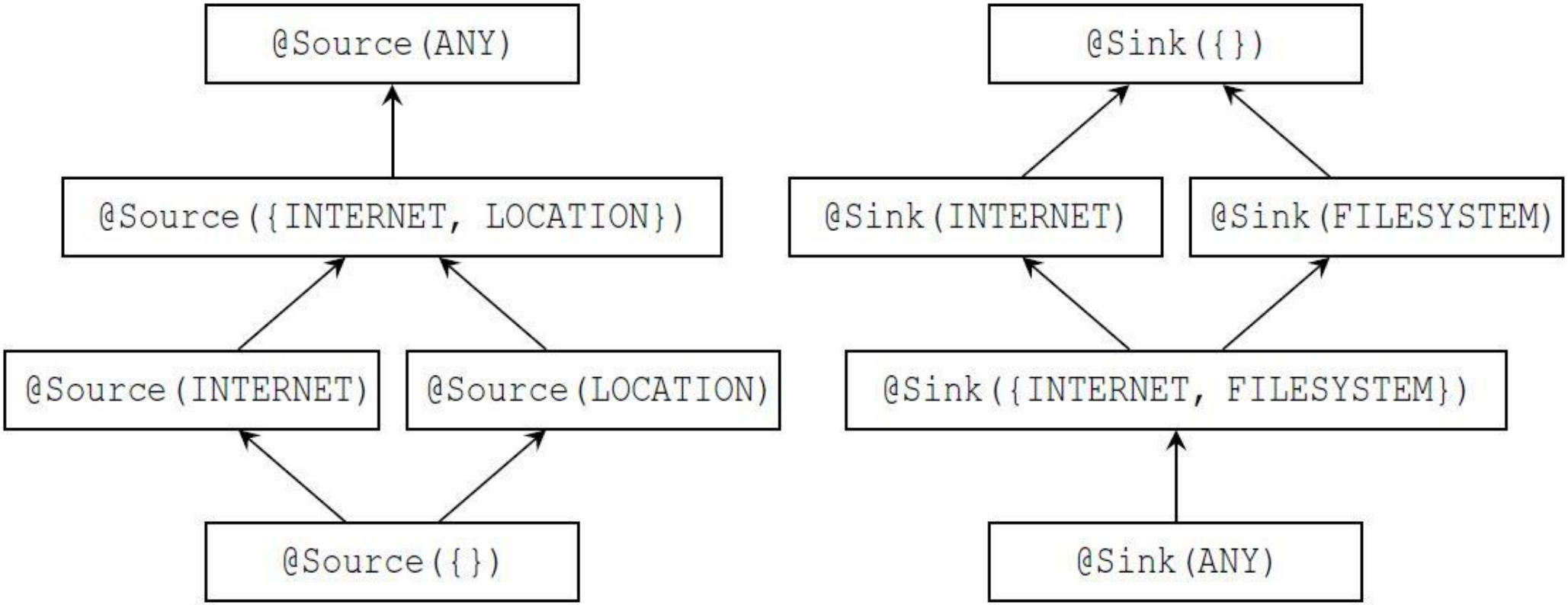
`@Source(SMS)` is a subtype of `@Source(SMS,INTERNET)`

`@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 two weeks ago and assiocated 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 

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:

LOCATION -> SMS 

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

App code:

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

Policy:

`LOCATION -> INTERNET` 

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

References to mutable data structures are tricky

```
@Sink(SMS,INTERNET) char[] a;

        @Sink(SMS) char[] b;
```

Is statement below ok?

```
a = b;
```

```
        // Obviously this should disallowed,
        // because now b can be sent over the internet
```

Is statement below ok?

```
b = a;
```

```
        // Less obviously incorrect!
        // Code elsewhere could sticks data in b and expect
        //      that this info can only be sent to SMS.
        // But that info can only be sent over the internet
        //      using reference a.
```

The trickier cases…

Problematic cases

- The SPARTA type system is overly restrictive

  Some 'legal' programs (which do not violate the policy) will be rejected, as show in next slides

  - 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 `@Source` *annotation of the code above, using the parameters* `LOCATION` *and* `SMS`*?*

Problem 1: heterogeneous arrays

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

a[0] = readGPS();

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

What would be a good `@Source` *annotation of the code above, using the parameters* `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 = a[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!

Problem 2: 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 an if-statement we can only send stuff over the internet
   if all variables used in the guard can be sent over the internet
   This becomes *very* restrictive!

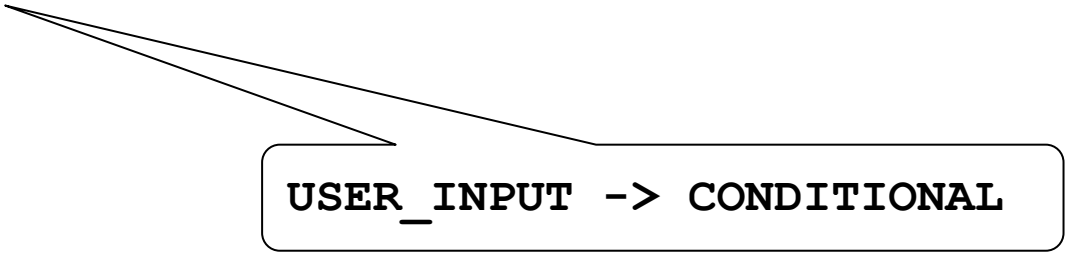
3. Solution used in SPARTA: introduce a new sink `CONDITIONAL`

Problem 2: implicit flows

```
@Source(USER_INPUT) long pin = getPINCode();

long i=0;

while (true) { if (i == pin) { sendToInternet(i); }
                 i++;
               }
```



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

Problem 3: generic/polymorphic functions

How to annotate a function such as

```
static String convertToLowerCase(String s) ?
```

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

So

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

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

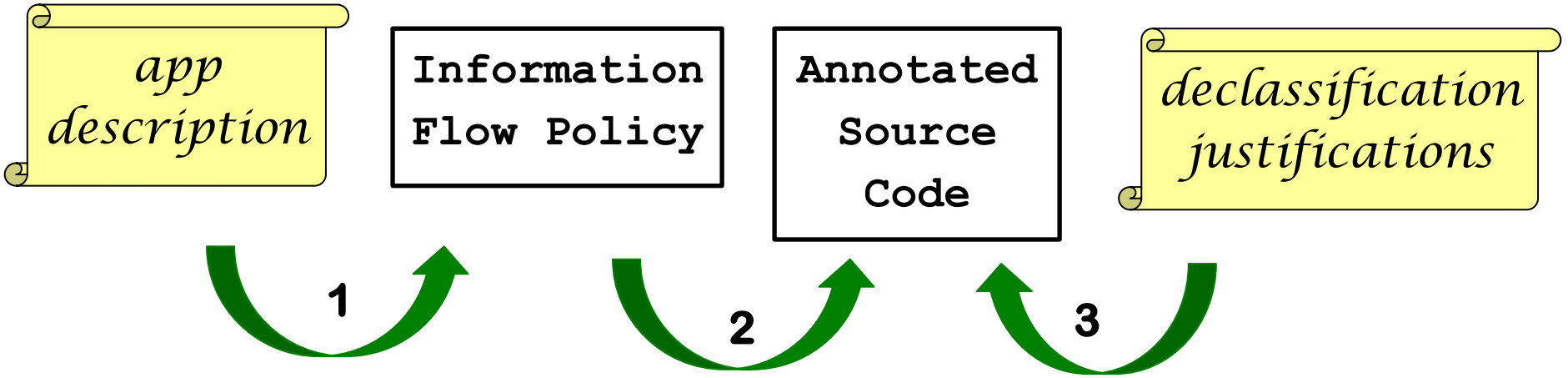
Solution: in annotations we can quantify over all type annotations, using a special type annotation variable `@PolySource`

```
@PolySource String convertToLowerCase(@PolySource String s)
```

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

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

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

Trusted Computing Base (TCB)

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

1. The Android OS, incl. the Java Virtual Machine
2. The type checker for annotations
3. The Java compiler
4. The annotations provided for the APIs
5. The annotations provided by the app developer
6. The human verifier

Trusted Computing Base (TCB)

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

1. **The Android OS, incl. the Java Virtual Machine** **YES**
2. **The type checker for annotations** **YES**
3. **The Java compiler** **YES**
4. **The annotations provided for the APIs** **YES**
5. ***the annotations provided by the app developer*** ***NO***
6. **The human verifier** **YES**

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.

information flow type systems for cryptographic software

Information flow analysis for side channels

In cryptographic software there is a risk of timing leaks

```
for i = 0 to KEY_LENGTH
{   ...
    if (key[i] = 1) then s1 else s2 ;
    ...
}
```

This has a timing leak if executing s_1 and s_2 take different times

In general, there is a risk of timing leaks of a guard depends on secret information.

So information flows from key material to conditionals are problematic; exactly the cases for which SPARTA has the loophole to ignore them.

Information flow analysis for side channels

This program snippet with (potentially) a timing leak

```
if ( key_bit = 0) then x = A else x = B;
```

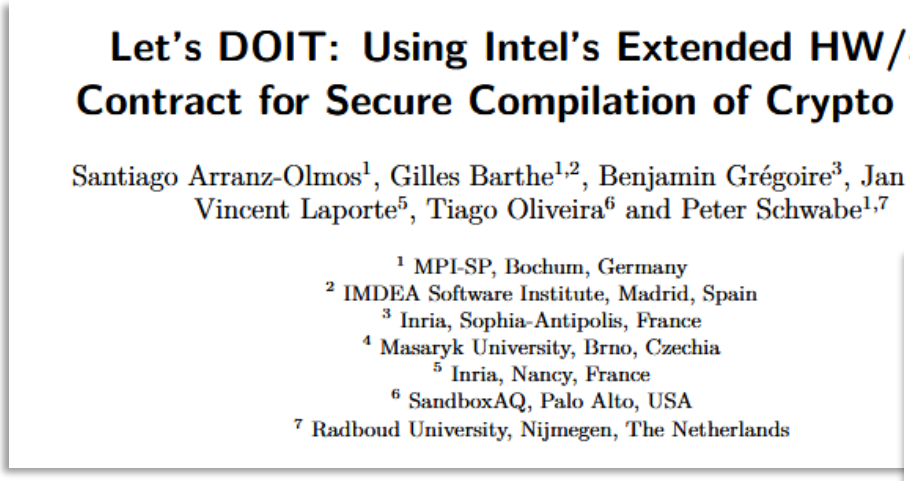
can be turned into (slower) time-insensitive code

```
a[0] = A;
a[1] = B;
x = a[key_bit];
```

But a good compiler will optimise this code to re-introduce the leak

Information flow typing for cryptographic software

- Intel's DOIT (Data Operand Independent Timing) architecture guarantees constant time execution for some instructions
  - DOIT requires disabling some hardware optimisations
- An information flow type system can be used to ensure that key material does not leak into guards or non-DOIT instructions

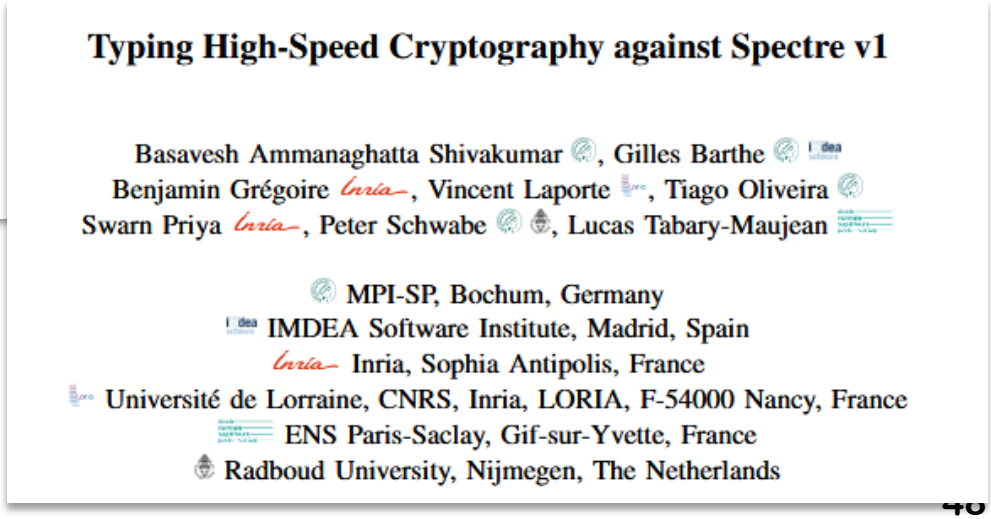


Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code

Santiago Arranz-Olmos[1], Gilles Barthe[1,2], Benjamin Grégoire[3], Jan Jancar[4], Vincent Laporte[5], Tiago Oliveira[6] and Peter Schwabe[1,7]

[1] MPI-SP, Bochum, Germany
[2] IMDEA Software Institute, Madrid, Spain
[3] Inria, Sophia-Antipolis, France
[4] Masaryk University, Brno, Czechia
[5] Inria, Nancy, France
[6] SandboxAQ, Palo Alto, USA
[7] Radboud University, Nijmegen, The Netherlands

CHES 2025



Typing High-Speed Cryptography against Spectre v1

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, Lucas Tabary-Maujean

MPI-SP, Bochum, Germany
IMDEA Software Institute, Madrid, Spain
Inria, Sophia Antipolis, France
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
ENS Paris-Saclay, Gif-sur-Yvette, France
Radboud University, Nijmegen, The Netherlands

S&P 2023

Zooming out

Types for security: this week vs three weeks ago

SPARTA

Eg `@Source(LOCATION)`

`@Sink(SMS)`

- Types (type annotations) for *confidentiality* in Java

- *Could we use a SPARTA-like approach for integrity too?*

  YES!

Google's Trusted Types

aka API hardening

Eg `SafeScript` `SafeUrl`

`TrustedResourceUrl` `SafeHtml`

- Types for *integrity* aka tainting in JavaScript

Zooming out

Security properties can be about CIA (Confidentiality, Integrity, Availability)

Both confidentiality & integrity are about information flow

This week and last week:

type systems for confidentiality

Earlier weeks:

static analysis tools & type systems for integrity (aka tainting)

- PreFAST `[SA_Post(Tainted=SA_Yes)]`
- semgrep

```
id: request-subprocess-taint-tracking
mode: taint
pattern-sources: - pattern: flask.request.$ATTRIBUTE
pattern-sinks:   - pattern: subprocess.$METHOD(...)
```

- Safe Builders approach using typing to prevent injection attacks
- Trusted Types using typing to prevent DOM-based XSS