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

http://cs.washington.edu/sparta

Werner Dietl

University of Waterloo

https://ece.uwaterloo.ca/~wdietl/

Problem: Malware in app stores

A
O
BY

SECURITY | 11/07/2011 @ 2:38PM | 62,847 views

iPhone Security Bug L
Innocent-Looking Apps Go Bad

13 comments, 11 called-out + Comment Now + Follow Comments

Fi
D
BY

Apple's iPhones and iPads have remained malware-free thanks mostly to the company's puritanical attitude toward its App Store: Nothing even vaguely sinful gets in, and nothing from outside the App Store gets downloaded to an iOS gadget. Now serial Mac hacker Charlie Miller has found a way to sneak a fully-evil app onto your phone or tablet, right under Apple's nose.

Seemingly benign "Jekyll" app passes Apple review, then becomes "evil"

flaw

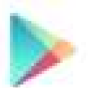
App that sneaks into App store can be used to steal data, take stealth photos.

by **Dan Goodin** - Aug 19 2013, 6:00pm PDT

HACKING IOS APPS 37

Current app approval process

Apple App Store: slow, arbitrary

 Google Play Store: fast, incomplete

- Coarse description of app behavior
- Binary executable is hard to analyze
- Vendor has little incentive to cooperate

Collaborative verification model

- Fine-grained specification of app behavior
  - Type system for information flow
- Analyze source code
  - Enables verification, not just heuristic bug-finding
- Vendor and app store do work that is easy for them
  - Lower cost overall

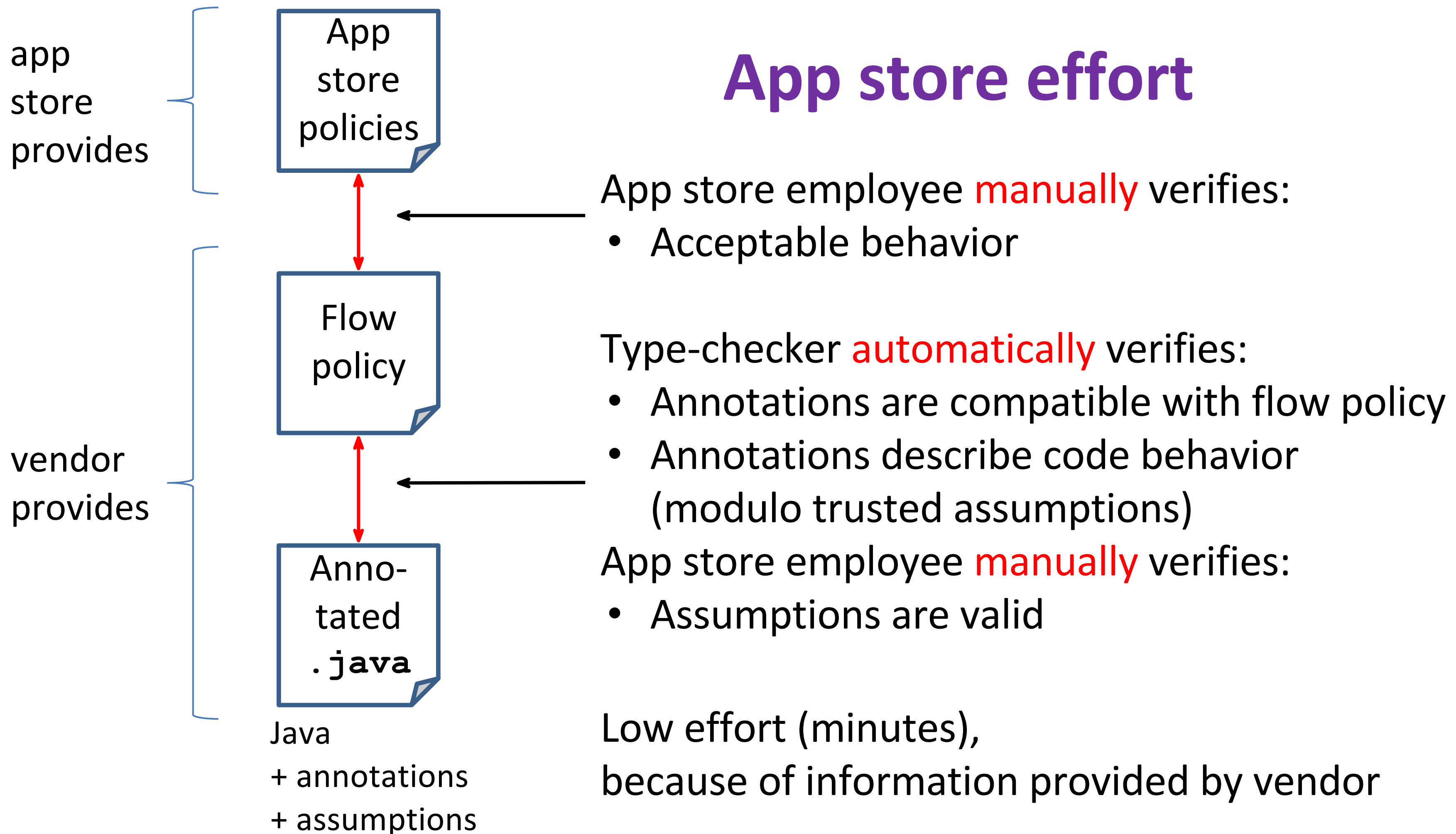
Results

- Information flow type system
  - Flow-sensitive, context-sensitive, reflection, intents
- DARPA hired 5 companies to create malware
  - Had access to our source code and documentation
  - Created 72 apps (576,000 LOC), 57 of them Trojans
- Our system detected 82% of the Trojans
  - 96% of Trojans with malicious information flow
  - Complements other analyses
  - 3 false alarms per 1000 lines of code
  - Outperformed a control team

Further details

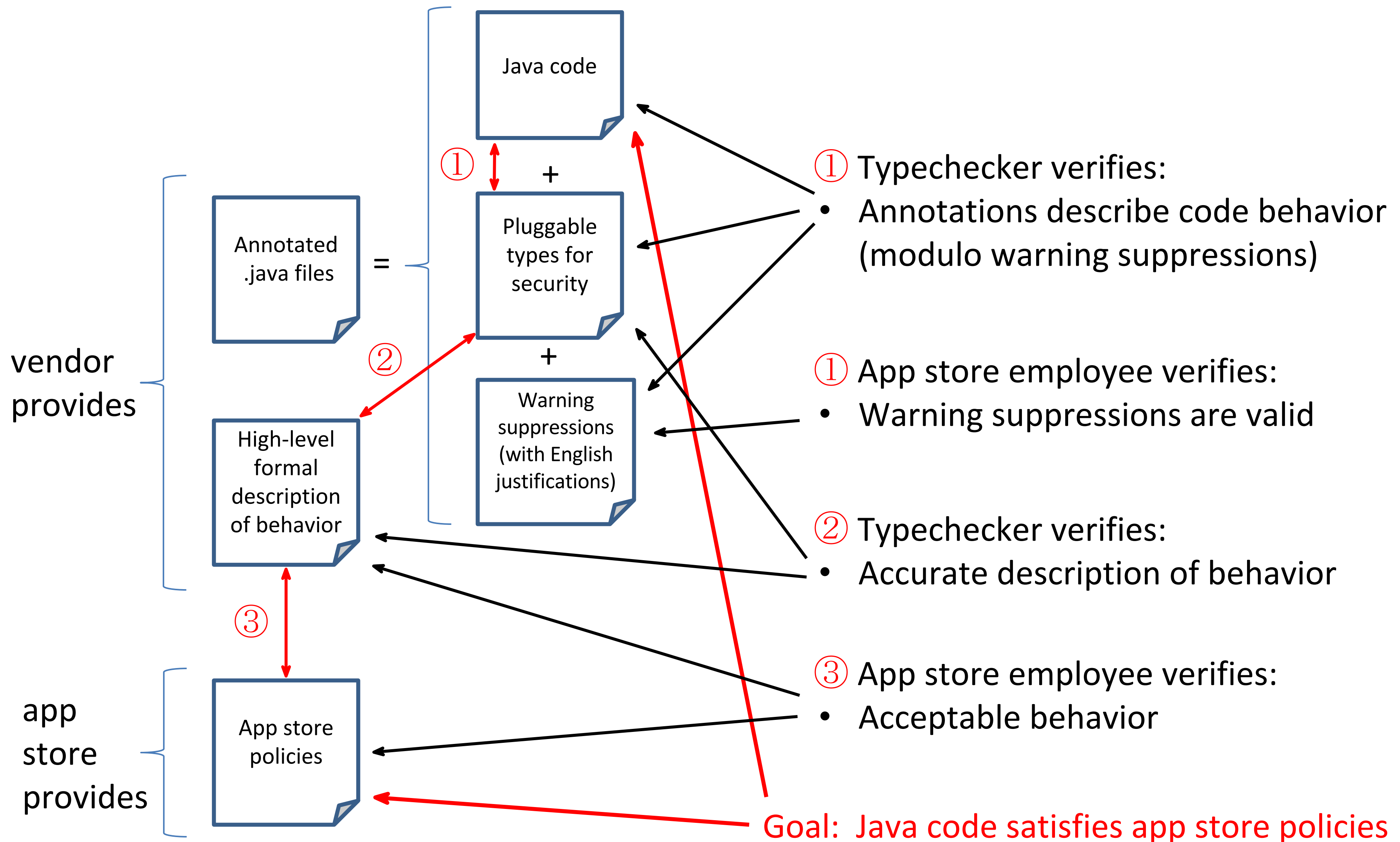
- "Collaborative verification of information flow for a high-assurance app store"
  by M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu;
  in Computer and Communications Security (CCS), 2014.
- "Static analysis of implicit control flow: Resolving Java reflection and Android intents"
  by P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst;
  in Automated Software Engineering (ASE), 2015.

Cooperative verification



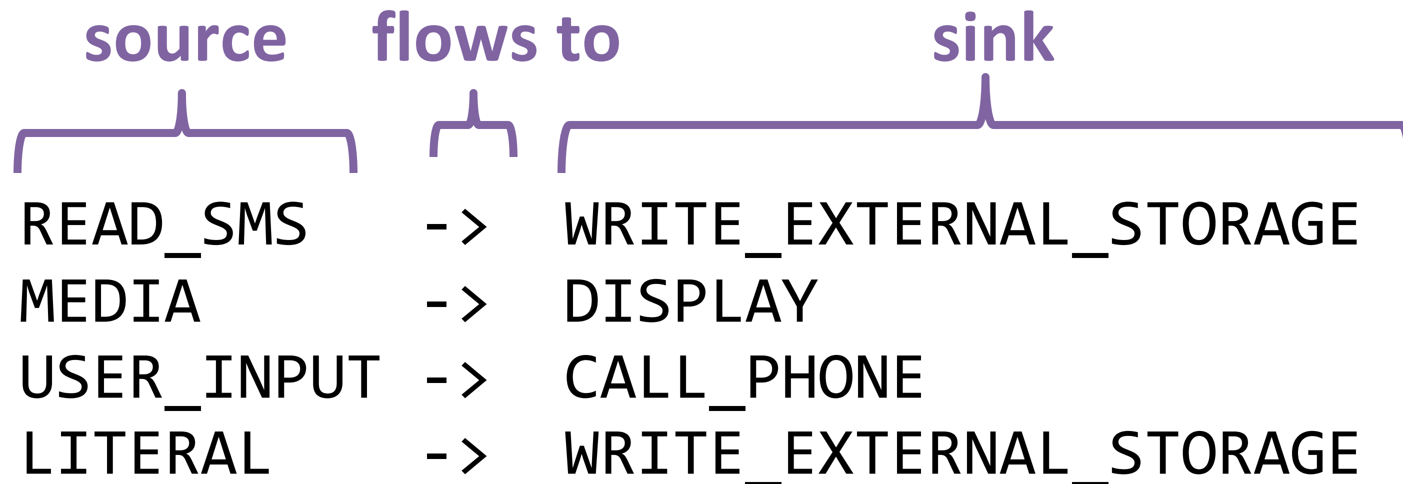
Conclusion: Java code satisfies app store policies

Piecewise verification



Information flow policy

Indicates permitted information flows in the program



Sources and sinks:

- Android permissions, familiar to developers (145)
- Other sensitive inputs or outputs (26 so far)
  Accelerometer, time of day, conditional, display, …

Flow policy for ShareLocation app

Read the description:

ShareLocation is a convenient application that shares the user's current GPS location with a contact via text message, Bluetooth, SMS, etc. Great for when you don't know exactly where you are and need to share your location urgently with someone.

Permissions in Android manifest:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" >
  </uses-permission>
  <uses-permission android:name="android.permission.INTERNET" >
  </uses-permission>
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" >
  </uses-permission>
  <uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" >
  </uses-permission>
  <uses-permission android:name="com.test.mypermission" />
```

Information flow policy:

ACCESS_COARSE_LOCATION -> INTERNET
ACCESS_FINE_LOCATION -> INTERNET

Information flow types

For a variable:

- What inputs might affect its value? **`@Source`**
- What outputs might it affect? **`@Sink`**

```
@Source(SMS) @Sink(FILE) String s;
...
s = getSMS();
...
writeToFile(s);
```

Annotation example

API Annotation

```
class Sender {
   @Source(SMS) Error sendSMS(@Sink(SMS) String message);
}
```

```
error = sender.sendSMS(msg);
```

From SMS

Going to SMS

Example (from Stardroid app)

```
public class LatLong {
   public @Source(LOCATION) float latitude;
   public @Source(LOCATION) float longitude;

   public LatLong(@Source(LOCATION) float latitude,
                  @Source(LOCATION) float longitude) {
      this.latitude = latitude;
      this.longitude = longitude;
   }

   …
}
```

Unannotatable malware in UltraCoolMap

Not marked as @Sink(NETWORK)

```
protected Void doInBackground(URI... uris) {
    ...
    URI uri = uris[0];
    HttpGet httpGet = new HttpGet(uri);
    ...
}
```

Required to be @Sink(NETWORK)

Information flow type-checking

Verifies two properties:

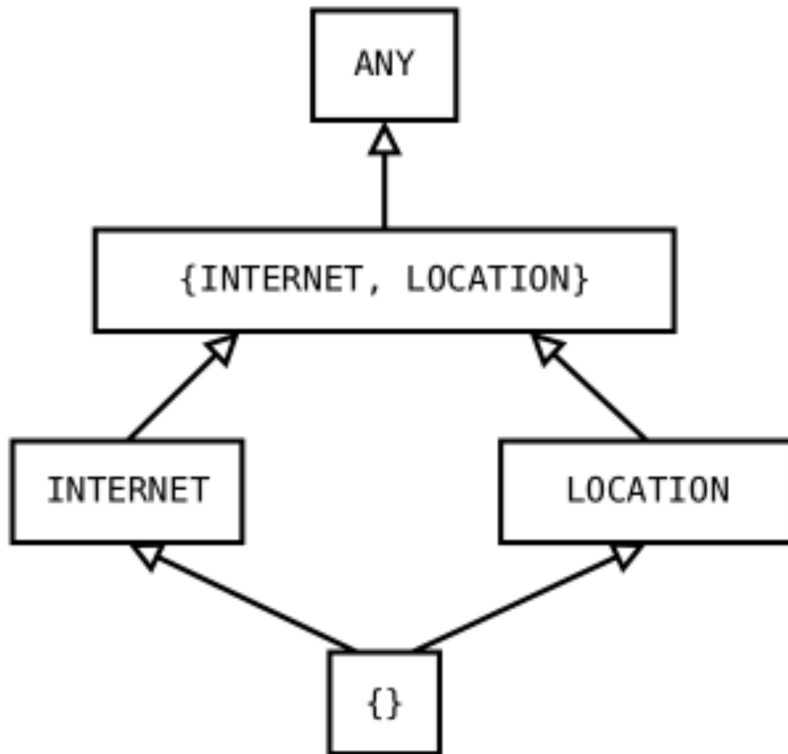
- No information flow beyond the flow policy
- Annotations are consistent (with the code and each other)
  - Annotations, and original developer, are untrusted

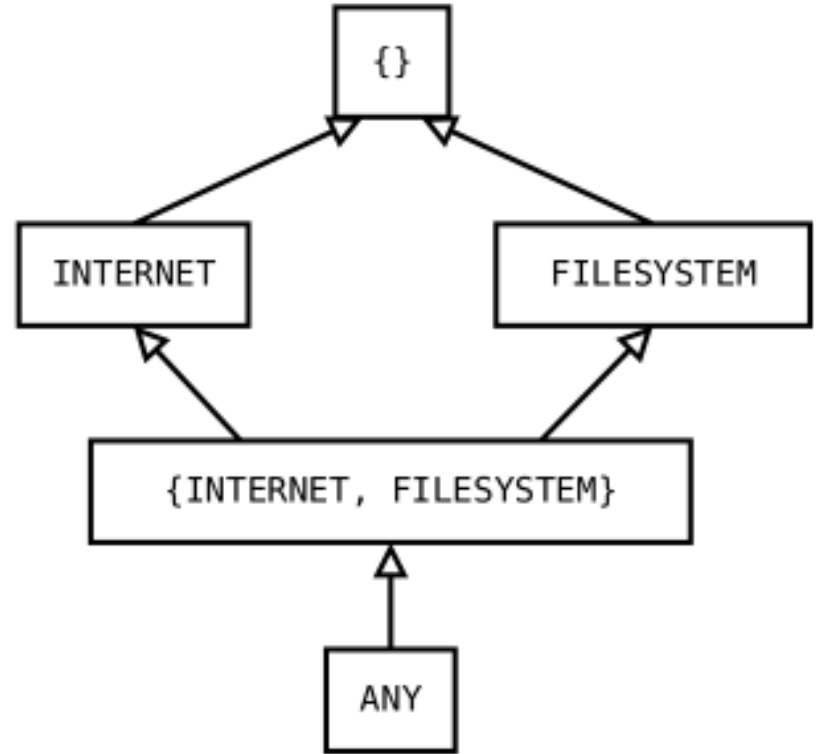
```
@Sink(SEND_SMS) String x;
display.setText(x);
```

Type-checker output:

```
warning: incompatible types in argument.
 display.setText(x);
                 ^
found   : @Sink(SEND_SMS)
required: @Sink(DISPLAY)
```

Type system hierarchy





Integrates cleanly with existing Java type system

Reducing annotation burden

- No annotations within methods
  - Local variable types are inferred
  - Technique: flow-sensitive type refinement
- Trusted annotations on Android and Java APIs
  - Manual annotations for over 10,000 methods
- Carefully-chosen defaults
  - Can be overridden by user
- Permit partial information flow types (next slide)

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})            String s
               @Sink(Y)   String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})              String s
               @Sink(Y) String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})                String s
                 @Sink(Y)     String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})   @Sink(Y) String s
                 @Sink(Y) String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})   @Sink(Y) String s
                 @Sink(Y) String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})   @Sink(Y) String s
                 @Sink(Y) String s
```

Flow policy inference

If the programmer writes only a source or sink:
Complete the type with the most general possibility

Suppose the flow policy is:

```
A -> X,Y
B -> Y
C -> Y
```

Then consider this code:

```
@Source({B,C})   @Sink(Y) String s
@Source({A,B,C}) @Sink(Y) String s
```

Benefits of flow policy inference

- Programmer may think about a computation in terms of only its sources or only its sinks
  - Enables more local reasoning
- Essential for annotating libraries
  - File constructor returns `@Source(FILESYSTEM)`
  - No `@Sink` annotation would be correct for all programs
  - Filled in based on the application's flow policy
- This is an application of type polymorphism

Polymorphism

- Type polymorphism (Java generics)

```
addToList(@PolyFlow Object,
          List<@PolyFlow Object>)
```

- Qualifier polymorphism
  - For generic *and non-generic* classes/methods

```
void <<Q>> append(StringBuffer<<@Q>> buf,
                  String<<@Q>> s);
```

  - Actual syntax uses Java 8 type annotations
- Observation: transitivity + inheritance + mutation = type unsoundness

Value-dependent behavior

Result type depends on the argument ***value***, not its ***type***:

```
@Source(LOCATION) x = instanceState.getDouble(LAT);
@Source(TIME) y = instanceState.getDouble(HOUR);
```

Avoid special-casing methods like **`getDouble`** in the type system:

- Annotate **`AppWidgetsColumns.LAT`** with @Source(LOCATION)
  (This constant, 22, doesn't actually reveal location)
- Annotate **`getDouble`** polymorphically:

```
@PolySource double
getDouble(@PolySource AppWidgetsColumns) {…}
```

Implicit/indirect flow

```
bool in_afghanistan;

in_afghanistan
  = (lat > 29 && lat < 39 && long > 61 && long < 75);

if (lat > 29 && lat < 39 && long > 61 && long < 75) {
  in_afghanistan = true;
} else {
  in_afghanistan = false;
}
```

Currently: Issue warning when secret data is used as a predicate

Future: Taint all assignments within conditional

Parameterized permissions

UltraCoolMap
- Mapping Application
- Location data is sent to `maps.google-com.cc` rather than `maps.google.com`
- Satisfies flow: LOCATION → NETWORK

Refine permissions to indicate specific destinations
- USER_INPUT -> INTERNET(“google.com”)
- FILESYSTEM
- WRITE_CONTACTS: email address vs. phone number vs. notes field

Reflection

Most static analyses ignore reflection*

– Analysis results are unsound if code uses reflection

Why handle reflection?

- Used for backward compatibility
  – Recommended in Android Developer Blog
- Used to access private APIs
  – Not recommended, but often used
- Used to prevent reverse engineering
  – Recommended in Android Developer Docs

* Soundiness Manifesto (http://soundiness.org/)

Reflection Example

```
Class cl = Class.forName("Sender");
Method m = cl.getDeclaredMethod("sendSMS");
Object error = m.invoke(sender, message);
```

could be from anywhere

could be going anywhere

API Annotation

```
class Method {
  @Source(ANY) Object invoke(@Sink(ANY) Object obj,
                             @Sink(ANY) Object... args);
}
```

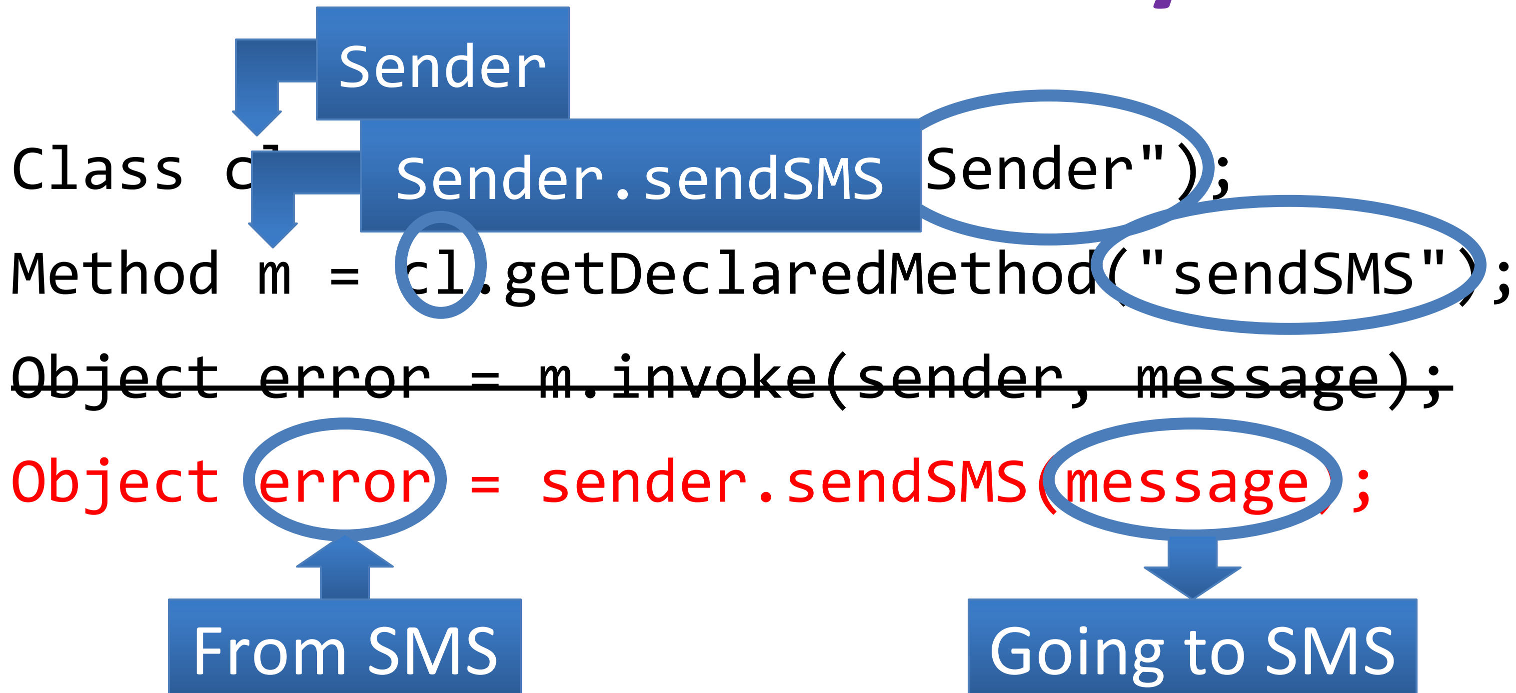
Reflection Analysis

Sender

Sender.sendSMS

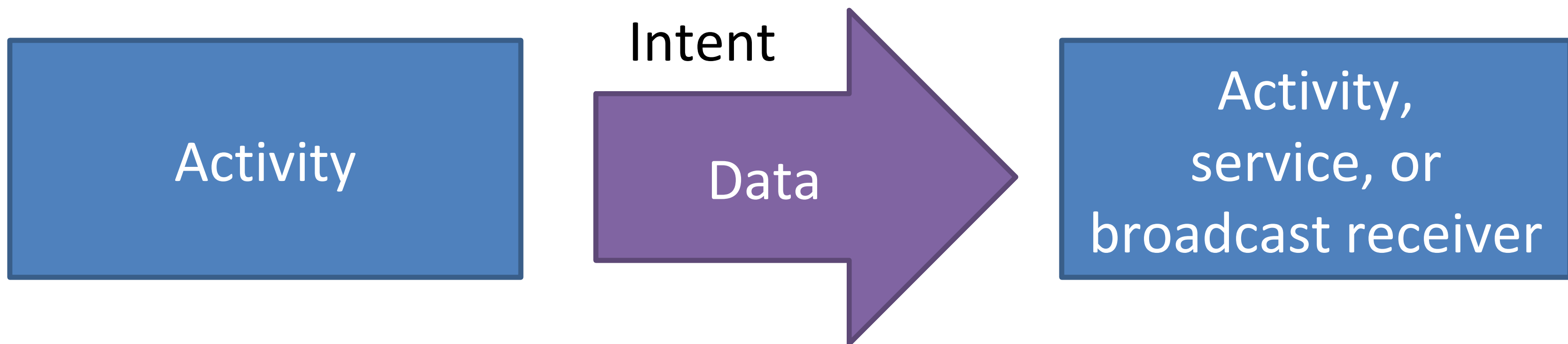
```
Class c                          Sender");
Method m = cl.getDeclaredMethod("sendSMS");
Object error = m.invoke(sender, message);
```

Reflection Analysis



- Key idea: constant propagation
  - Class and Method are constants
  - Invoke API and program methods where needed
- Resolved 96% of reflection in Red Team apps
- Integrates with any downstream analysis

Inter-component communication in Android apps



Flow policy must be respected across components

Inter-component communication in Android apps

```
Intent s
  = new Intent(Receiver);
s.putExtra("k1", valueA);
s.putExtra("k2", valueB);
sendIntent(s);
```

Intent

"k1" -> valueA
"k2" -> valueB

```
void receiveIntent(Intent r) {
   Object a = r.getExtra("k1");
   Object b = r.getExtra("k2");
}
```

1. Where is the intent going? [Octeau 2013]
2. What happens to the data when it gets there?

Intent types

```
@IExtra(key="k1",type=@A)
@IExtra(key="k2",type=@B)
```

```
Intent s
  = new Intent(Receiver);
@A valueA = ...;
@B valueB = ...;
s.putExtra("k1", valueA);
s.putExtra("k2", valueB);
sendIntent(s);
```

Intent

"k1" -> valueA
"k2" -> valueB

```
@IExtra(key="k1",type=@A)
@IExtra(key="k2",type=@B)
```

```
void receiveIntent(Intent r) {
@A Object a = r.getExtra("k1");
@B Object b = r.getExtra("k2");
}
```

Implicit control flow via Intents

Method called by sender	Method invoked on receiver
startActivity(Intent i)	void setIntent(Intent intent)
sendBroadcast(Intent i)	void onReceive (…, Intent intent)
startService(Intent s)	int onStartCommand (Intent intent, …)
bindService(Intent s, …)	IBinder onBind (Intent intent)
and others.	

Implicit control flow via Intents

Method called by sender	Method invoked on receiver
sendIntent(Intent i)	receiveIntent(Intent i)

Implicit control flow via Intents

Method called by sender	Method invoked on receiver
sendIntent(Intent i)	receiveIntent(Intent i)

Methods that add data	Methods that retrieve data
putExtra(String key, String value)	getStringExtra(String key)
putExtra(String key, int value)	getStringExtra(String key, int default)
putExtra(String key, String[] values)	getStringArrayExtra(String key)
and others.	

Implicit control flow via Intents

Method called by sender	Method invoked on receiver
sendIntent(Intent i)	receiveIntent(Intent i)

Methods that add data	Methods that retrieve data
putExtra(String key, Object value)	getExtra(String key)

and others.

Subtyping vs. intent sending

- Subtyping:
  - At least as many keys
  - Invariant typing: identical value types

$$\frac{\forall k \in \text{keys}(\tau_2) : k \in \text{keys}(\tau_1) \wedge \tau_1.\text{get}(k) = \tau_2.\text{get}(k)}{\tau_1 <: \tau_2}$$

- Intent sending:
  - A copy of the intent is sent to the receiver; no aliasing
  - Subtyping permitted among value types

$$\frac{\forall k \in \text{keys}(\tau_2) : k \in \text{keys}(\tau_1) \wedge \tau_1.\text{get}(k) <: \tau_2.\text{get}(k)}{\tau_1 <_{\text{copyable}} \tau_2}$$

Typing judgments for `getExtra` and `putExtra`

$$\frac{k \text{ is a compile-time constant string} \quad e : \tau \quad \sigma = \tau.\text{get}(k)}{e.\text{getExtra}(k) : \sigma}$$

$$\frac{k \text{ is a compile-time constant string} \quad e : \tau \quad v : \tau.\text{get}(k)}{e.\text{putExtra}(k, v) : \tau}$$

How does this catch malware?

Flow policy:
LOCATION -> DISPLAY

```
@IExtra(key="data",
        type=@Source(ANY)
             @Sink(INTERNET))
```

@Source(LOCATION) loc =…
s.putExtra("data", **loc**);
sendInter…);

```
@IExtra(key="data",
        type=@Source(LOCATION)
             @Sink(DISPLAY))
```

Intent

void receiveIntent(Intent r){
Obj data = r.getExtra("data");
sendToInternet(**data**);
}

Red Team evaluation

- DARPA hired 5 companies (Red Teams) to create Trojans
  - Had access to our source code and documentation
- 20 people worked on the 5 teams
  - 18 full-time security analysts with BS or MS degree
  - 2 interns
  - Most have been exposed to information flow theory
- Surveyed real-world malware
- Created 72 apps (576,000 LOC), 57 of them Trojans
  - 2 goals: simulate real-world Trojans and defeat our system
  - Java source code only
  - Largely undocumented
    - less than an app store blurb, no code comments, poor code style
  - No type qualifiers

Analyzing an app

We received the apps in 5 batches

- Limited time to evaluate each batch
  - We spent most of our time doing reverse engineering
  - Would not be necessary in collaborative verification model

Process:

- Run home-grown reverse-engineering tool
- Add type annotations where necessary
- Use human insight
- Did not run the app
  - In practice, should use complementary analyses to raise the bar for attackers

Results:

- Our team correctly classified 88% of the apps
- Reverse-engineering tools: 53% of the Trojans
- Information flow types : 82% of the Trojans
  - 96% of apps with malicious information flow
- Outperformed a control team using static and dynamic analysis

Types of Trojan behavior

18: Android permission not in the app's description

- SMS Backup app uses READ_BROWSER_HISTORY
- Analysis requires only the Android manifest file

18: information flow not in the app's description

- 2D Game uses READ_EXTERNAL_STORAGE -> INTERNET
- RSS Reader uses RANDOM -> VIBRATE
- Would not be caught by current analyses

11: maliciously exploit an information flow that is in the app's description

- Calculator used USER_INPUT -> FILESYSTEM
- Mapping used LOCATION -> INTERNET (wrong destination)
- The best way to defeat our system

11: malware is not related to information flow

- Battery drain, performance, encryption
- Use a complementary analysis

Challenge Apps
Exploited information flow

The app had a valid reason to use the information flow, but also used it maliciously

App	Description of Malware	Exploited Flow
UltraCoolMap	Location data is sent to maps.google-com.cc rather than maps.google.com	LOCATION->NETWORK
SMS Reminder	When a text of 000000000 is sent, all messages are deleted	SMS->CONDITIONAL
Text Secure	Sends all SMSes to attacker's phone	USER_INPUT->SMS
InstantMessage	Drops all incoming SMSes	SMS->DISPLAY
AndroidIRC	Sends chats to user "0XFFF"	USER_INPUT->INTERNET
Snapshot Share	Sends "About screen" to Dropbox account	READ_EXTERNAL_FILESYSTEM ->INTERNET

Misuses of valid information flows

- SMS Reminder
  - Sends SMS Reminders
  - Deletes all SMS
  - Satisfies flow: SMS $\longrightarrow$ CONDITIONAL
- Text Secure
  - SMS encryption app
  - Forwards all SMS to attacker
  - Satisfies flows: USER_INPUT $\longrightarrow$ SMS and SMS $\longrightarrow$ NETWORK

Bugs and bugdoors

Missing functionality:

- Picture Sharing app claims to send pictures to the user's contacts
  - Does not have the READ_CONTACT permission

Bugdoors:

- GPS app passes device ID as a waypoint to the remote server
  - Server can correlate location to device
  - Server can correlate location to other info collected using the device ID
- Password Saver saves unencrypted passwords in shared preferences
  - Accessible to other applications on the device

The Red Teams were unaware of these bugdoors

Lessons learned

Sneaky patterns:

- 17 apps used reflection to access an API
- 3 apps exploited the ACTION_VIEW intent to access a URL without the INTERNET permission
- 6 apps exfiltrated sensitive data to the log
  - Does not require an Android permission
  - Does require a permission in our finer-grained permission system

Benefits of information flow types:

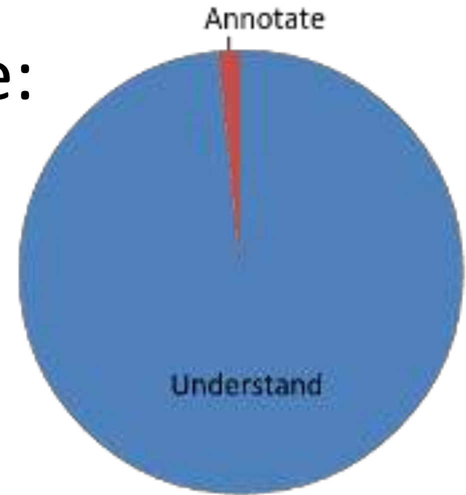
- Program annotations foil some of the ways to hide malware:
- Hamper data exfiltration
  - Hiding data-flow based malware in an annotated application is difficult
- General
  - Revealed malicious data flow in the payload as well as the injected triggers
  - Easy to extend as we discovered new properties
- Extensible to integrity as well as privacy

Annotation burden

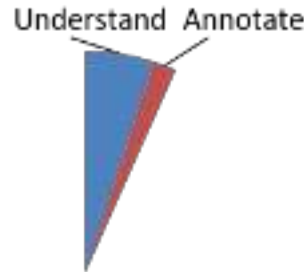
- 6 apps (3924 LOC, 2 Trojans) fully annotated by UW team
- Annotated 8 LOC per minute
  - Most time was reverse-engineering
  - Much lower if familiar with app
  - Much lower to audit an already-annotated app
  - Industry averages 20 LOC per day
- 1 annotation per 19 LOC
  - 4% of annotatable locations
  - Jif [Myers et al.]: 1 annotation per 4 LOC
- 3 false alarms per 1000 lines of code

Vendor effort:
How hard is it to annotate code?

- If you are not the author of the code:



- For the author:



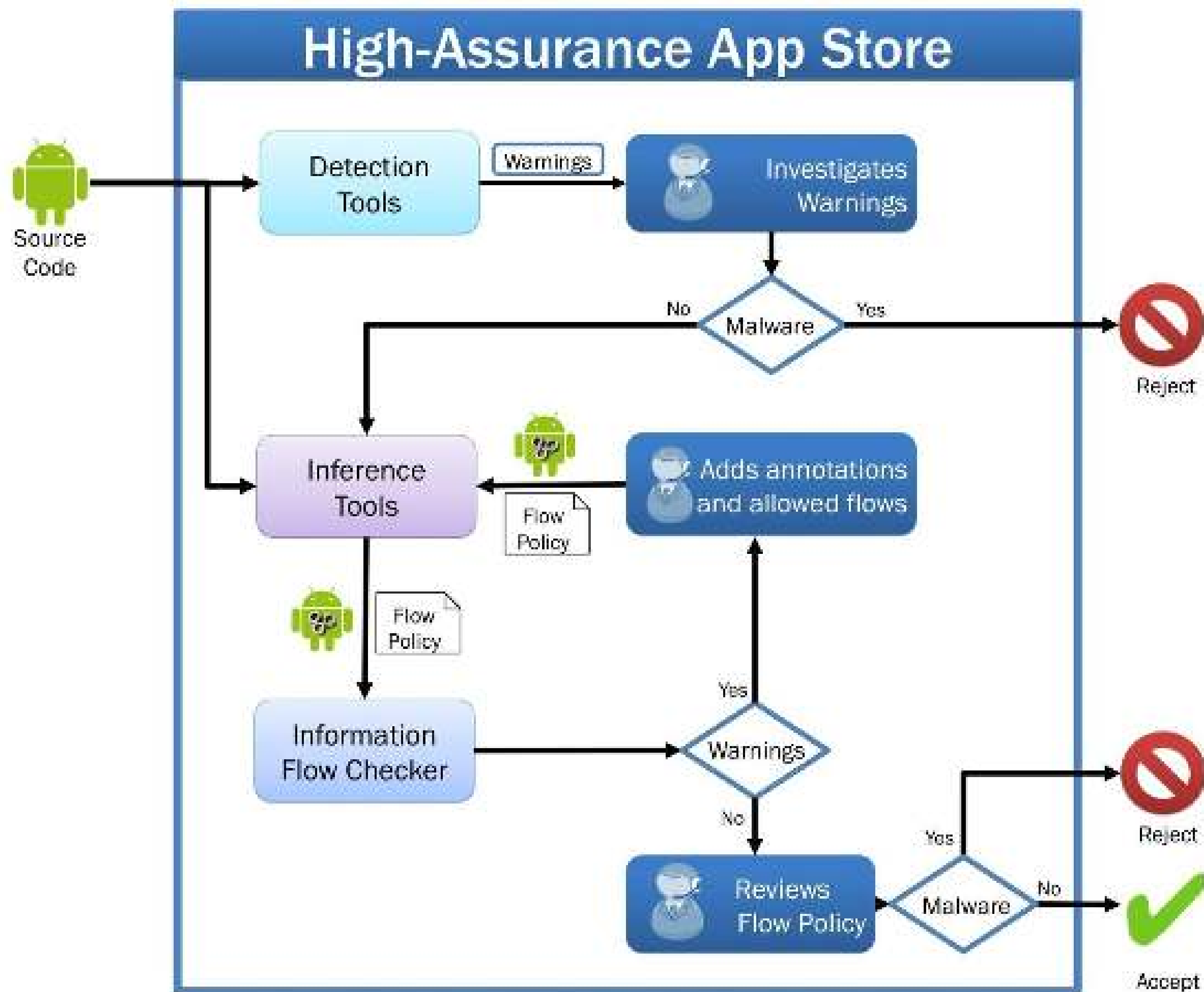
- If the author writes the annotations along with the code:
  - It might even save time

Future work: enrich flow policies

- Information flow path rather than just endpoints
  - Personal information can go to the Internet only after passing through an encryption module
- Conditional information flows
  - MICROPHONE -> INTERNET only while the user presses the "transmit" button

Related work

- Information flow (a very old idea):
  - Jif [Zdancewic 2001]: heavier-weight, complex
  - WebSSARI [Huang 2004]: PHP; inserts runtime checks
- Android
  - Many apps are overprivileged [Felt 2011]
  - Static analysis: Woodpecker for capabilities [Grace 2012], ComDroid for intents [Chin 2011]
  - Dynamic analysis: TaintDroid [Enck 2010], DroidScope [Yan 2012], AppFence [Hornyack 2011], Aurasium [Xu 2012]
  - Many others


High-Assurance App Store
Source Code
Detection Tools
Warnings
Investigates Warnings
No
Malware
Yes
Reject
Inference Tools
Adds annotations and allowed flows
Flow Policy
Flow Policy
Information Flow Checker
Yes
Warnings
No
Reviews Flow Policy
Yes
Malware
No
Reject
Accept

Contributions

- Collaborative verification model
  - Vendor and app store do work that is easy for them
- Information flow type system
  - Flow-sensitive, context-sensitive, indirect flow, reflection, intents
- Implementation for Java Android apps
- Red Team evaluation
  - Effective, easy to use, few false positives

Disclaimer

This material is based on research sponsored by Defense Advanced Research Project Agency (DARPA) under agreement number FA8750-12-C-0107. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Project Agency (DARPA) or the U.S. Government.

Credits

Paulo Barros
Ravi Bhoraskar
Jonathan Burke
Sunjay Cauligi
Alexei Czeskis
Tammy Denning
Werner Dietl
Michael Ernst
Seungyeop Han
Carl Hartung
René Just
Tadayoshi Kohno
Karl Koscher
Philip Lai
David McArthur
Suzanne Millstein
Stuart Pernsteiner
Mark Roberts
Franzi Roesner
Rafael Vertido
Paul Vines
David Wetherall
Edward Wu
Shawn Zhang

Further details

- "Collaborative verification of information flow for a high-assurance app store"
  by M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu;
  in Computer and Communications Security (CCS), 2014.
- "Static analysis of implicit control flow: Resolving Java reflection and Android intents"
  by P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst;
  in Automated Software Engineering (ASE), 2015.