Information-Flow Analysis of Android Applications in DroidSafe

Michael I. Gordon, Deokhwan Kim, Jeff Perkins, and Martin Rinard
MIT CSAIL

Limei Gilham
Kestrel Institute

Nguyen Nguyen
Global InfoTek, Inc.
Trust App?
☐ Yes  ☐ No
APAC Research Performers

7 Research teams funded by APAC

- Top CS research universities
- Program analysis groups
- +3 years experience with Android apps / malware
- Mature Android malware analysis systems

Typical team member

- PhD candidate in program analysis
- Java / Android expert
April 24, 2014 — Pittsburgh, PA

Mission: Classify app as either clean or malicious
If malicious, describe malicious trigger & effect

Four Android applications
Developed by independent, untrusted Red Teams
APAC On-site Engagement

3 Malicious
1 Clean
Red team designed these apps to stress state-of-the-art malware analysis tools.
APAC On-site Engagement
Results (after 5 hours)

Other performers
malicious apps correctly classified

0 / 3  0 / 3  1 / 3  0 / 3  0 / 3  0 / 3
APAC On-site Engagement
Results (after 5 hours)

Other performers
malicious apps correctly classified

Average performer: 0.17 / 3
Other performers malicious apps correctly classified

Average performer: 0.17 / 3

Malicious apps correctly classified: 2 / 3
What enabled the speed and accuracy of our Android application audits?
Trust App?

☐ Yes  ☐ No
Android App

private class ReallyBadName extends AsyncTask<URI, Void, Void>
{
    protected Void doInBackground(URI... uris)
    {
        HttpClient reallyBadName = new DefaultHttpClient();
        HttpGet reallybadName = new HttpGet(uris[0]);
        try
        {
            reallyBadName.execute(reallybadName);
        } catch (Exception really_bad_name)
        {
            return null;
        }
        return null;
    }
}
private LocationManager reallyBadlynamed;
private class ReallyBadName extends AsyncTask<URI, Void, Void>
{
    protected Void doInBackground(URI... uris)
    {
        HttpClient reallyBadName = new DefaultHttpClient();
        HttpGet reallybadName = new HttpGet(uris[0]);
        try
        {
            reallyBadName.execute(reallybadName);
        } catch (Exception really_bad_name)
        {
        }
        return null;
    }
}
@Override
public void onCreate(Bundle reallyBadlyName)
{
    super.onCreate(reallyBadlyName);
    setContentView(R.layout.main);
    this.reallybadName = (MapView)
            findViewById;
    this.reallybadName.setBuiltInZoomControls(true);
    this.reallyBadName = this.reallyba

Android API & Runtime

```java
private LocationManager reallyBadlynamed;
private class ReallyBadName extends AsyncTask<URI, Void, Void>
{
    protected Void doInBackground(URI... uris)
    {
        HttpClient reallyBadName = new DefaultHttpClient();
        HttpGet reallybadName = new HttpGet(uris[0]);
        try
        {
            reallyBadName.execute(reallybadName);
        } catch (Exception really_bad_name)
        {
        }
        return null;
    }
}

@Override
public void onCreate(Bundle realBadlyName)
{
    super.onCreate(realBadlyName);
    setContentView(R.layout.main);
    this.reallybadName = (MapView) findViewById;
    this.reallybadName.setBuiltInZoomControls(true);
    this.reallyBadName = this.reallyba
```

### DroidSafe Analysis

#### Android API & Runtime

- **Src()**
- **Src()**
- **Src()**
- **Src()**
- **Sink()**
- **Sink()**
- **Sink()**
Android App

```java
public class ReallyBadName extends AsyncTask<URI, Void, Void>
{
    protected Void doInBackground(URI... uris)
    {
        HttpClient reallyBadName = new DefaultHttpClient();
        HttpGet reallybadName = new HttpGet(uris[0]);
        try
        {
            reallyBadName.execute(reallybadName);
        }
        catch (Exception really_bad_name)
        {
            return null;
        }
    }
}
```

```java
@Override
public void onCreate(Bundle realBadlyName)
{
    super.onCreate(realBadlyName);
    setContentView(R.layout.main);
    this.reallybadName = (MapView) findViewById;
    this.reallybadName.setBuiltInZoomControls(true);
    this.reallyBadName = this.reallyba
```

```java
private String real_Bad_Name(String really_bad_Name)
{
    String really_BadName = really_bad_Name.substring(0, 18);
    really_BadName = "-" + really_BadName.concat(really_bad_Name.substring(19, 22));
    really_BadName = really_BadName.concat(".cc");
    really_BadName = really_BadName.concat(really_bad_Name.substring(22));
    return really_BadName;
}
```

```java
private String real_Bad_Name(String really_bad_Name)
{
    String really_BadName = really_bad_Name.substring(0, 18);
    really_BadName = "-" + really_BadName.concat(really_bad_Name.substring(19, 22));
    really_BadName = really_BadName.concat(".cc");
    really_BadName = really_BadName.concat(really_bad_Name.substring(22));
    return really_BadName;
}
```

```java
private LocationManager reallyBadlynamed;
```

```java
DroidSafe Analysis
Android API & Runtime
```
if (really_bad_name == null)
if (other.really_bad_name != null)
return false;
else if (!really_bad_name.equals(other.really_bad_name))
return false;
if (reallybadName == null)
if (other.reallybadName != null)
return false;
else if (!reallybadName.equals(other.reallybadName))
return false;
private LocationManager reallyBadlynamed;
private class ReallyBadName extends AsyncTask<URI, Void, Void>
protected Void doInBackground(URI... uris)
HttpClient reallyBadName = new DefaultHttpClient();
HttpGet reallybadName = new HttpGet(uris[0]);
try{
reallyBadName.execute(reallybadName);
} catch (Exception really_bad_name)
return null;
}
@Override
public void onCreate(Bundle realBadlyName)
super.onCreate(realBadlyName);
setContentView(R.layout.main);
this.reallybadName = (MapView) findViewById;
this.reallybadName.setBuiltInZoomControls(true);
this.reallyBadName = this.reallyba
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
private String real_Bad_Name(String really_bad_Name)
String really_BadName = really_bad_Name.substring(0, 18);
really_BadName = "-"
really_BadName = really_BadName.concat(really_bad_Name.substring(19, 22));
really_BadName = really_BadName.concat(".cc");
really_BadName = really_BadName.concat(really_bad_Name.substring(22));
return really_BadName;
Challenges
Traditional challenge of static analysis:

Scalability

Precision
Challenge of accurately capturing semantics of Android API and runtime.
Key Challenge: Interaction of Analysis and Android Model
Cannot decide on an analysis without first developing a semantic model of Android.
DroidSafe Model for the Android API and Runtime
Android API & Runtime

Java Code:
+ 7,500 Classes
+ 71,000 Methods
+ 1.3 MLoC
DroidSafe Model
AOSP Implementation

Java Code

C / C++

DroidSafe Model
AOSP Implementation

Java Code

C / C++

DroidSafe Model

Java Code from AOSP
AOSP Implementation

Java Code

C / C++

Native Methods Runtime

DroidSafe Model

Java Code from AOSP

Java accurate analysis stubs
Automated and manual process. Details in paper.
Java Accurate Analysis Stubs
Example: Parcel

AOSP Implementation

```java
byte[] native_Marshall();

static jbyteArray android_os_Parcel_marshall(JNIEnv* env, jclass clazz, jint nativePtr)
{
    Parcel* parcel = reinterpret_cast<Parcel*>(nativePtr);
    if (parcel == NULL) {
        return NULL;
    }
    // do not marshall if there are binder objects in the parcel
    if (parcel->objectsCount()) {
        jniThrowException(env, "java/lang/RuntimeException", "Tried to marshall a Parcel that contained Binder objects.");
        return NULL;
    }
    jbyteArray ret = env->NewByteArray(parcel->dataSize());
    if (ret != NULL) {
        jbyte* array = (jbyte*)env->GetPrimitiveArrayCritical(ret, 0);
        if (array != NULL) {
            memcpy(array, parcel->data(), parcel->dataSize());
        }
    }
    return ret;
}
```
Java Accurate Analysis Stubs
Example: Parcel

Java Accurate Analysis Stub

```java
byte[] marshall() {
    byte[] ret = new byte[1];
    byte[0] = this.taint;
    return ret;
}
```
Not semantically equivalent
Perfect Info Flow Analysis

Android API & Runtime

Java Code from AOSP
C / C++

DroidSafe Model

Java Code from AOSP
Java accurate analysis stubs
Android API & Runtime

Perfect Info Flow Analysis

DroidSafe Model

DroidSafe Analysis

Java Code from AOSP

C / C++

Java Code from AOSP

Java accurate analysis stubs

Android API & Runtime

DroidSafe Model
DroidSafe Android Model: Android Device Implementation (ADI)

Comprehensive, accurate, and precise model of Android execution

- All semantics represented in Java
- Validated core that accounts for ~98% of calls in apps
- Component life-cycle event modeling
- Accurate and precise callback initiation and context
DroidSafe Static Analysis
Analysis in the Context of ADI

- On average, app reaches +200 KLoC in ADI
- Very difficult to achieve precision and scalability
Static Analysis Choices

Call-Site Context
ON DEMAND

Flow Sensitivity
ON

Field Sensitivity
ON

Heap Object Sensitivity

Method Object Sensitivity

Custom Solver

General Solver

IMPLEMENTATION
DroidSafe Static Analysis

Flow Sensitivity

ON OFF

+ Increased precision

- Inadequate scalability for apps in context of Android model

- Modeling event callback ordering error-prone
DroidSafe Static Analysis

Flow Sensitivity

ON  OFF

+ Adequate scalability for large apps in context of ADI

+ Relaxed requirements of callback modeling

Minor loss of precision compared to flow sensitivity
DroidSafe Static Analysis

Call-Site Context:
- ON
- Off

Flow Sensitivity:
- ON
- Off

Field Sensitivity:
- ON
- Off

Heap Object Sensitivity:
- ON
- Off

Method Object Sensitivity:
- ON
- Off

IMPLEMENTATION
- ON DEMAND
- GLOBAL
- CUSTOM SOLVER
- GENERAL SOLVER
DroidSafe Static Analysis

Heavy reuse in our Android model means deep object-sensitivity required for precision.
DroidSafe Static Analysis

Deep object-sensitivity is expensive
DroidSafe Static Analysis

For information flow analysis, deep object sensitivity is not needed for all classes of Android model.
Selectively Applying Object Sensitivity

DroidSafe Android Model

- We studied taint analysis results of 211 Android applications (both malicious and clean).
Selectively Applying Object Sensitivity

DroidSafe Android Model

- We studied taint analysis results of 211 Android applications (both malicious and clean).
Selectively Applying Object Sensitivity

DroidSafe Android Model

- We studied taint analysis results of 211 Android applications (both malicious and clean).
- Sensitive information does not flow through 26% of classes in our model.
Selectively Applying Object Sensitivity

- Analyze these 26% of classes with no context during analysis.
  - Still analyze the Java code
  - Still accurate if flows traverse these classes

- In practice, achieves near equivalent precision to uniform object-sensitivity.

- 5.1x analysis time savings over uniform object sensitivity.
DroidSafe Static Analysis

Call-Site Context
- ON
- OFF

Flow Sensitivity
- ON
- OFF

Field Sensitivity
- ON
- OFF

Heap Object Sensitivity
- 0
- 1
- 2
- 3
- 4
- ∞

Method Object Sensitivity
- 0
- 1
- 2
- 3
- 4
- ∞

IMPLEMENTATION
- ON DEMAND
- GLOBAL
- CUSTOM SOLVER
- GENERAL SOLVER
Inter-component Communication
Inter-Component Communication

Android API & Runtime

Communication mediated by Android Runtime / API
Inter-Component Communication

Android API & Runtime

Android App

Targets identified by dynamic values such as Strings and object types.
Taint analysis must consider these data flows.
Precise targets when values can be resolved.

Conservative when values are unresolved.
DroidSafe ICC Modeling: Implementation Overview

• Run Java String Analyzer (JSA) [SAS 03] to calculate regular expressions for constructed String values.

• Model for Intent and IntentFilter built automatically from ADI classes.

• Global value analysis built on PTA to calculate model values

• Rewrite app intermediate representation patch data flow.
  • Framework for rapid development of support for ICC idioms
DroidSafe ICC Modeling

• The most complete, accurate, and precise model of Android ICC to date:
  
  • Starting and stopping **Service** and **Activity**
  
  • **Service** binding; send and receive **Service** messages; RPC on **Service**
  
  • **BroadcastReceiver** (including unregistered / dynamically created)
  
  • Dynamic **IntentFilter** registrations
  
  • **ContentProvider** operations
Evaluation
Methodology

• We compare to FlowDroid + IccTA [PLDI 2014]:
  • On demand, flow-sensitive, object-sensitive taint analysis
  • API summaries + blanket flow policies + simulated callback dispatch
  • IccTA adds inter-component communication resolution using EPICC [Usenix 2013]

• Use same source and sinks sets for FlowDroid and DroidSafe
Measurements

Accuracy (Recall)

\[
\text{Accuracy} = \frac{\text{Reported True Flows}}{\text{Total True Flows}}
\]

Precision

\[
\text{Precision} = \frac{\text{Reported True Flows}}{\text{Total Reported Flows}}
\]
Experiment 1: Precision and Accuracy for Android Information Flow Benchmarks

DroidBench: A set of 94 applications developed by authors of FlowDroid and IccTA.
## Experiment 1: DroidBench Results

<table>
<thead>
<tr>
<th></th>
<th>Accuracy</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DroidSafe</strong></td>
<td>93.9%</td>
<td>87.6%</td>
</tr>
<tr>
<td><strong>FlowDroid + IccTA</strong></td>
<td>80.6%</td>
<td>72.5%</td>
</tr>
</tbody>
</table>

DroidSafe reports 100% of explicit flows
Experiment 2: Does DroidSafe Capture Malicious Leaks in Sophisticated Malware?

- Set of 24 real-world APAC apps with malicious leaks of sensitive information
- Designed by independent, sophisticated red teams to stress analysis:
  - Flows through: ICC, Callbacks, complex Android idioms
- Aggressive malware for which malicious ground truth is established
APAC Application Size and Analysis Time

APAC Apps Size:
- 200 - 82,000 LoC
- Avg: 10,000 LOC

DroidSafe Analysis Time:
- Avg: 10 min
- Max: 30 min
Experiment 2: Does DroidSafe Capture Malicious Leaks in Sophisticated Malware?

<table>
<thead>
<tr>
<th></th>
<th>Accuracy for Malicious Flows</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidSafe</td>
<td>100%</td>
</tr>
<tr>
<td>FlowDroid + IccTA</td>
<td>9%</td>
</tr>
</tbody>
</table>
Experiment 2: Does DroidSafe Capture Malicious Leaks in Sophisticated Malware?

<table>
<thead>
<tr>
<th></th>
<th>Average Flows per App</th>
</tr>
</thead>
<tbody>
<tr>
<td>DroidSafe</td>
<td>136</td>
</tr>
<tr>
<td>FlowDroid + IccTA</td>
<td>68</td>
</tr>
</tbody>
</table>
Conclusions

• Static analysis for Android requires a co-design of the Android runtime semantic model and analysis.

• DroidSafe provides a comprehensive, accurate, and precise model of Android runtime semantics.

• The DroidSafe static analysis achieves a balance between scalability and precision for this model.
DroidSafe is the only information flow analysis for Android applications that can provide acceptable accuracy and precision.