STATIC ANALYSIS WITH SOOT

An introduction into static analysis using Soot
AGENDA

- Motivation
- The Soot Compiler Framework
- Intra-Procedural Static Analysis with Soot
- Limitations of Static Program Analysis
- Inter-Procedural Static Analysis with Soot
- Static Data Flow Analysis with FlowDroid
- Recap
Who Am I?

- 4th year PhD Student at TU Darmstadt
- Researcher at Fraunhofer SIT

Research interests:
- Static analysis
- IT security

Community service
- Reviewer for conferences & journals
- Maintainer of Soot and FlowDroid
Hacking Group at Fraunhofer SIT

www.team-sik.org
Hacking Group at Fraunhofer SIT

- Team SIK is our hacking group at Fraunhofer SIT

- We analyze real-world software and systems
  - Backend-as-a-service libraries and how they are used inside apps
  - Antivirus and security apps for Android
  - SmartHome appliances
  - Password managers

- What is broken in the real-world?
- How can we improve the security of real systems?
Why Focus on Android Apps?
Why Focus on Android Apps?

- Image: Inauguration of the pope in 2005 vs. 2013

- Back then: Just people. Now: Everyone has a smartphone

- Android has >80% market share

- That’s why it’s important to look into Android app security
Software Has Defects
Software Has Defects

- Recap on the concrete projects of TeamSIK
- Pretty bad vulnerabilities in all analysis targets
- BaaS: Access to millions of sensitive records
  - Health data, car accidents, baby growth data, ...
- Antivirus apps: Remote code execution, disabling of scan engine, ...
- SmartHome, password managers: <not yet published>
Software Has Defects (2)

- Insecure Crypto
- Web-Based Vulnerabilities
- Poor Authentication
- Capability Leakage
- Old Android Versions
- Improper Error Handling
Software Has Defects (2)

- Web-based vulnerabilities
  - Calls from JavaScript into Java code
  - Pages can execute code inside the app

- Poor authentication
  - Unauthorized access to data
  - App authenticated instead of user

- Old Android versions
  - Old security vulnerabilities
  - Insecure defaults
Software Has Defects (2)

- Improper error handling
  - Ignored security warnings

- Capability leakage
  - Authentication objects passed to wrong receivers
  - Impersonation attacks possible

- Insecure Crypto
  - Home-brewed crypto algorithms
  - Own, insecure implementations of common algorithms
Software Has Defects (3)

- Web-Based Vulnerabilities
- Insecure Crypto
- Poor Authentication
- Capability Leakage
- Old Android Versions
- Improper Error Handling
Sometimes https url shows blank white screen in the android webview. It's because you have to trust the ssl certification or you need to override the ssl error in your webview client.

```java
private class MyWebViewClient extends WebViewClient {
    ...
    
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
        handler.proceed(); // Ignore SSL certificate errors
    }
}
```

class TrustEveryoneManager implements X509TrustManager {
    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType) throws CertificateException {
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException {
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
}

Source: http://stackoverflow.com/questions/25509296/trusting-all-certificates-with-okhttp
Software Can Be Malicious

Angry Birds and other Mobile Gaming apps leaking your private information to NSA
by Swati Khandelwal on Monday, January 27, 2014
Software Can Be Malicious (2)

- Sensitive Data Leakage
- Unauthorized Payments
- Remote Control
- Blackmailing
- Ransomware
- Eavesdropping
- Clickjacking
Software Can Be Malicious (2)

- Unauthorized payments
  - Send SMS messages to premium-rate telephone numbers
  - Dial premium-rate telephone numbers

- Remote Control
  - Command&control bots
  - Attacker sends commands via web service, SMS, IRC, etc.

- Ransomware
  - Lock phone or encrypt data
  - Request ransom for unlocking / decryption
Software Can Be Malicious (2)

- Clickjacking
  - Trick user into clicking on UI elements
  - UI elements cover sensitive controls such as accepting app permissions

- Eavesdropping
  - Record audio using microphone
  - Record video using camera

- Blackmailing
  - Extract photos / documents to blackmail users
  - Install fake data to blackmail people
Software Can Be Malicious (2)

- Sensitive Data Leakage
  - Send data from phone to attacker
  - Sensor data: GPS location (tracking)
  - Files (photos, documents)
  - Input control data (microphone + camera recordings)
Software Can Be Malicious (3)

- Unauthorized Payments
- Sensitive Data Leakage
- Blackmailing
- Remote Control
- Eavesdropping
- Ransomware
- Clickjacking
public void onCreate(Bundle b) {
    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage("0906 123456", null, mgr.getDeviceId(), null, null);
}

App sends message to costly telephone number without the user’s consent
Software Can Be Malicious (3)

- Sensitive Data Leakage
- Unauthorized Payments
- Remote Control
- Blackmailing
- Ransomware
- Eavesdropping
- Clickjacking
public void onCreate(Bundle b) {
    TelephonyManager mgr = (TelephonyManager)
            this.getSystemService(TELEPHONY_SERVICE);

    SmsManager sms = SmsManager.getDefault();
    sms.sendTextMessage("+49 1234", null,
                    mgr.getDeviceId(),
                    null, null);
}
Sensitive Data Leakage (2)
Sensitive Data Leakage (2)

- Many apps contain advertisement frameworks
  - Common revenue model
  - App is free, developer is payed through ad revenue

- Most add frameworks leak data
  - Unique device identification
  - User profiling for targeted advertisement

- Often more than one ad framework inside an app
  - More revenue for the developer
  - Multiple channels for obtaining the ads
Why Static Analysis?

- Find issues during development / before installation
- No code coverage problem
  - Can inspect internal software state
  - Inspect all cases, all paths, all states
- Simple to run in production
  - No execution environment needed
  - Can be fully automated
Common Static Analysis Tasks

- **Data flow analysis**
  - Is this app leaking sensitive information?
  - Is this server performing sensitive operations on untrusted data?

- **Callgraph analysis**
  - Is certain debug code still reachable in the final version?
  - Is there a control flow path in which no certificate check happens?

- **Type state analysis**
  - Can a banking client receive a loan without being checked?
Common Static Analysis Tasks (2)

- Value Analysis
  - To which phone numbers does the app send messages?
  - Does the app connect to known malicious URLs or IP addresses?
  - Are there hard-coded crypto keys or credentials in the app?
The Toolkit

General-Purpose Compiler and Analysis Framework

Efficient Data Flow Tracker Based on Soot
Analysis Targets

Java Source Code

Java Bytecode

Android Bytecode

Android Bytecode

Jimple IR

Java Bytecode

Android Bytecode
Analysis Targets

- Common IR: Jimple

- Front-ends convert input files to Jimple
  - Java Source Code
  - Java Bytecode
  - Android APK Files

- Back-ends convert Jimple to output files
  - Java Bytecode
  - Android APK Files
Hello World From Soot

```
C:\Users\varzt\Downloads>java -jar soot-trunk.jar -process-dir org.fdroid.k9_17046.apk -src-prec apk -output-format jimple -android-jars "\Program Files (x86)\Android\android-sdk\platforms\android-15\android.jar" as android.jar
Warning: java.lang.invoke.LambdaMetafactory is a phantom class!
Warning: java.lang.ref.Finalizer is a phantom class!
Warning: android.app.ActivityOptions is a phantom class!
Warning: android.app.Notification$BigPictureStyle is a phantom class!
Warning: android.app.Notification$BigTextStyle is a phantom class!
Warning: android.app.Notification$InboxStyle is a phantom class!
Warning: android.view.accessibility.AccessibilityNodeProvider is a phantom class!
Warning: org.apache.tools.ant.BuildException is a phantom class!
Warning: org.apache.tools.ant.Task is a phantom class!
Warning: org.jdom.DefaultDOMFactory is a phantom class!
Warning: org.jdom.Element is a phantom class!
Warning: org.jdom.Document is a phantom class!
Warning: org.jdom.CDATA is a phantom class!
Warning: org.jdom.Comment is a phantom class!
Warning: org.jdom.Content is a phantom class!
Warning: org.jdom.Namespace is a phantom class!
Warning: org.jdom.Text is a phantom class!
```
Hello World From Soot

- Convert APK file to Jimple files
- Basic usage of Soot on the command line
- We’ll look at the individual command-line options later on
- Result: One Jimple file per class
  - Flat structure, all files in one single directory
Hello World From Soot (2)
A Deeper Look At The Options

- **-process-dir**
  - Analysis target, the app or program
  - Is always fully loaded into memory and converted to Jimple

- **-soot-classpath**
  - Additional dependencies referenced by the app or program
  - Only loaded on demand as much as necessary

- **-allow-phantom.refs**
  - If a referenced class or method doesn’t exist, create a phantom
  - Not sound or complete, but highly practical
A Deeper Look At The Options (2)

- **-android-jars**
  - Base directory of Android platform JARs
  - Soot picks one according to the app’ manifest
  - Becomes part of the default Soot classpath
  - Attention: When overwriting the classpath, you must specify the Android JAR on your own!

- **-force-android-jar**
  - Pick one specific platform JAR, don’t let Soot choose
A Deeper Look At The Options (3)

- **-src-prec**
  - Format of the source files
  - APK not on the default list, must be set explicitly

- **-output-format**
  - Output file format
  - Use “none” if you’re just doing static analysis

Now, what do we get from all this?
The Jimple Intermediate Representation

```java
public void <init>() {
    soot.jimple.infoflow.Infoflow $this;
    java.util.HashSet r4;

    $this := @this: soot.jimple.infoflow.Infoflow;
    specialinvoke $this.<soot.jimple.infoflow.AbstractInfoflow: void <init>>();

    r4 = new java.util.HashSet;
    specialinvoke r4.<java.util.HashSet: void <init>>();
    $this.<soot.jimple.infoflow.Infoflow: java.util.Set onResultsAvailable> = r4;

    return;
}
```
The Jimple Intermediate Representation

- Each Jimple body consists of three parts
  - Local declarations (= local variables)
  - Units (= statements)
  - Traps (= exception handler definitions)

- All constructs explicit
  - Explicit constructor calls
  - Explicit returns from void methods
Using Soot for Static Analysis

Input Files → Soot → Custom Static Analysis Code

Do not parse Jimple files on your own!
Soot Workflow
Soot Workflow

- Soot was designed as a compiler framework
  - Linear processing queue
  - Packs contain transformers that change or analyze code
  - Pre-defined packs for standard tasks and optimizations

- Two classes of packs
  - Normal packs operate at one body at a time
  - Whole-program packs (start with w): Operate on the whole scene at once
  - Whole-program packs are only executed in whole-program mode!
Soot Workflow

- Jb: Jimple Body Pack
  - Basic transformations directly after loading
  - Examples: Type assignment, trap tightening

- Cg
  - Despite the name, a whole-program pack!
  - Constructs the callgraph

- Wtjp: Whole Jimple Transformation Pack
  - Your own whole-program transformers go here
Soot Workflow

- Wjop: Whole Jimple Optimization Pack
- Wjap: Whole Jimple Annotation Pack

- Jtp: Jimple Transformation Pack
  - Your normal, i.e., non-whole-program, transformers go here

- Jop: Jimple Optimization Pack
- Jap: Jimple Annotation Pack
Soot Workflow (2)
Let’s do it, then

```java
public static void main(String[] args) {
    PackManager.v().getPack("jtp").add(new Transform("jtp.MyTransformer", new BodyTransformer() {
        @Override
        protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
            System.out.println(b.getMethod().getSignature());
        }
    }));
}
```

```java
soot.Main.main(new String[] { "-process-dir", "D:/org.fdroid.k9_17046.apk", 
        "-src-prec", "apk", 
        "-output-format", "none", 
        "-allow-phantom-refs", 
        "-android-jars", "C:\\Program Files (x86)\\Android\\android-sdk\\platforms" });
```
Let’s do it, then (2)

<android.support.v4.app.FragmentActivity: void onStop()>
Soot Method Signatures

- Uniquely identifies a method

- Structure: `<class_name: return_type method_name(arg1, ...., argn)>`

- Class names are always fully-qualified
  - "java.lang.String" instead of just "String"
  - Arguments are fully-qualified type names

- Subsignature: `return_type method_name(arg1, ...., argn)`
  - Identifies a method inside a class
Find broken SSL implementations

Recall what the issue looked like:

```java
private class MyWebViewClient extends WebViewClient {
  ...

  @Override
  public void onReceivedSslError(WebView view,
      SslErrorHandler handler, SslError error) {
    handler.proceed(); // Ignore SSL certificate errors
  }
}
```
Find broken SSL implementations (2)

private class MyWebViewClient extends WebViewClient {
    @Override
    public void onReceivedSslError(WebView view,
        SslErrorHandler handler, SslError error) {
        handler.proceed(); // Ignore SSL certificate errors
    }
}

- Agenda for checking implementations:
  1. For each method, check whether it’s called “onReceivedSslError”
  2. Check whether the declaring class extends “WebViewClient”
  3. Check whether there is at least one path that doesn’t call “proceed()”
@Override
protected void internalTransform(Body b, String phaseName, Map<String, String> options) {
    // Check method signature
    if (b.getMethod().getSubSignature().equals("onReceivedSslError(android.webkit.WebView," + "android.webkit.SslErrorHandler,android.webkit.SslError)")) {
        // Check that we're inside a web view
        Type classType = b.getMethod().getDeclaringClass().getType();
        Type webViewType = RefType.v("android.webkit.WebView");
        if (Scene.v().getOrMakeFastHierarchy().canStoreType(classType, webViewType)) {
            // Look for calls to proceed()
            // TODO
        }
    }
}
private class MyWebViewClient extends WebViewClient {
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
        handler.proceed(); // Ignore SSL certificate errors
    }
}

- Agenda for checking implementations:
  1. For each method, check whether it’s called “onReceivedSslError”
  2. Check whether the declaring class extends “WebViewClient”
  3. Check whether there is at least one path that doesn’t call “proceed()”
Find broken SSL implementations (5)

// Look for calls to proceed()

for (Unit u : b.getUnits()) {
    Stmt stmt = (Stmt) u;
    if (stmt.containsInvokeExpr()) {
        SootMethod target = stmt.getInvokeExpr().getMethod();
        if (target.getSignature().equals("<android.webkit.SslErrorHandler: void proceed()>")) {
            System.out.println("Found a call to proceed()");
        }
    }
}
Instances of this class are created by the WebView and passed to onReceivedSslError(WebView, SslErrorHandler, SslError). The host application must call either proceed() or cancel() to set the WebView's response to the request.

- Updated agenda for checking implementations:
  1. For each method, check whether it’s called “onReceivedSslError”
  2. Check whether the declaring class extends “WebViewClient”
  3. Check whether there is at least one path that calls “abort()”

// Look for calls to proceed()
for (Unit u : b.getUnits()) {
  Stmt stmt = (Stmt) u;
  if (stmt.containsInvokeExpr()) {
    SootMethod target = stmt.getInvokeExpr().getMethod();
    if (target.getSignature().equals("<android.webkit.SslErrorHandler: void cancel()>")) {
      System.out.println("Found a call to cancel, we’re safe()");
    }
  }
}
Find broken SSL implementations - Recap

- Concept of a body transformer
- Access to method and declaring class
- Access to units inside the method body
- Cast-compatibility for checking inheritance
- Reduced the path-sensitive checking problem to a linear one

Task for self-study: Implement checker for other SSL vulnerabilities
- TrustManager.checkServerTrusted
- TrustManager.getAcceptedIssuers
Is that all?

- Some broken trust managers are debug-only
  - Never instantiated in production code
  - Reachability analysis required

- Android is more than Java classes
  - Manifest file
  - Layout XML files
  - Android framework callbacks
How good can we be?

- Can we build a perfect static analyzer?
  - Never report anything that isn’t true
  - Never miss anything that is true

- Answer: No

Image source: http://www.freestockphotos.biz/stockphoto/10661
Static Program Analysis

- Static analysis tackles an undecidable problem
  - Does property A hold in some execution of program P?
Static Program Analysis

Soundness

Precision

"P always holds"

Recall = \frac{\text{Number of found results}}{\text{Number of existing results}}

Precision = \frac{\text{Number of true results}}{\text{Number of reported results}}

"P never holds"
Static Program Analysis

Soundness

Analyzer

Precision

Performance
Static Program Analysis

- Three goals: Precision, Soundness, Performance

- Can’t have all of them
  - Doing better one of them makes it harder to get the other two
  - Domain-specific tradeoffs required
Trade-Offs in Static Analysis

- Do not load dependencies (libraries, etc.)
  - Improves performance, reduces memory consumption
  - Reduces precision, need to make assumptions

- Always over-approximate when unsure
  - Can even be totally sound
  - Precision degrades
Trade-Offs in Static Analysis (2)

- Always under-approximate when unsure
  - Leads to false-negatives
  - False-positives reduced

- Apply maximum-precision reasoning
  - Reduces performance
  - Reduces false-positives

Soundness
Precision
Performance
Speaking of Extremes

- What about a sound static analysis?
  - Can we always compromise on precision?
  - Are there any other implicit assumptions?

- Assumptions one can think of
  - Code is available
  - Code can be parsed
  - Code always behaves the same
  - ...
public static void gdadbjrj(String paramString1, String paramString2) {
    // Get class instance
    Class clz = Class.forName(gdadbjrj.gdadbjrj("VRIf3+In9a.a TA3RYnD1BcVR"));
    Object localObject = clz.getMethod(gdadbjrj.gdadbjrj("\]a9\")).invoke(null); 
    // Get method name
    String s = gdadbjrj.gdadbjrj("BaRIta*9caBBV]a");
    // Build parameter list
    Class[] arr = new Class[] { nglpsq.cbhgc, nglpsq.cbhgc, nglpsq.cbhgc }; 
    // Get method and invoke it
    clz.getMethod(s, arr).invoke((Object)localObject, paramString1, null, paramString2, null, null);
}
Welcome To Real-World Malware (2)

- Evades static analysis
  - Reflection
  - Native code
  - Dynamic code loading

- Evades dynamic analysis
  - Emulator detection
  - Environment dependencies
  - Timing bombs
  - Command & control architecture
Taking a Broader View

- So far: Focus on individual methods
  - Analyze one method body at a time

- Now: Whole-program analysis
  - Analyze properties of the program as a whole

- Example: Callgraph analysis
  - Which statement calls which method
  - Inverse: Which method is called where
  - Implicitly answers reachability
public static void main(String[] args) {
    X a = new X();
    X b = new Y();

    a.bar();
    b.bar();

    foo(a);
    foo(b);
}

void foo(X x) {
    x.bar();
}
Constructing a Callgraph

```java
public static void main(String[] args) {
    ...

    soot.Main.main(new String[] {
        "-process-dir", "D:/org.fdroid.k9_17046.apk",
        "-src-prec", "apk",
        "-output-format", "none",
        "-allow-phantom.refs",
        "-android-jars", "C:\Program Files (x86)\Android\android-sdk\platforms",
        "-whole-program",
        "-phase-option", "cg.spark", "on"
    });
}
```
Constructing a Callgraph (2)

PackManager.v().getPack("wjtp").add(new Transform("wjtp.MyTransformer",
   new SceneTransformer()) {

   @Override
   protected void internalTransform(String phaseName, Map<String, String> options) {
      // Iterate over the callgraph
      for (Iterator<Edge> edgeIt = Scene.v().getCallGraph().iterator(); edgeIt.hasNext(); ) {
         Edge edge = edgeIt.next();
         SootMethod smSrc = edge.src();
         Unit uSrc = edge.srcStmt();
         SootMethod smDest = edge.tgt();
         System.out.println("Edge from " + uSrc + " in " + smSrc + " to " + smDest);
      }
   }

}});
PackManager.v().getPack("wjtp").add(new Transform("wjtp.MyTransformer", new BodyTransformer()) {

@Override
protected void internalTransform(Body b, String phaseName, Map<String, String> options) {

// Print out the reachable methods
QueueReader<MethodOrMethodContext> rdr = Scene.v().getReachableMethods().listener();

while (rdr.hasNext()) {
    MethodOrMethodContext mmoc = rdr.next();
    System.out.println(mmoc.method());
}

...}
});
Callgraphs for Android Apps

- Traditional: Start at main() method
  - Iteratively process calls
  - Iteratively extend callgraph

- Android: No main() method available
  - Either over-approximate
  - Or create dummy main method
  - Use lifecycle model from FlowDroid
public static void main(String[] args) throws IOException, XmlPullParserException {
    // Initialize Soot
    SetupApplication analyzer = new SetupApplication("C:\Program Files (x86)\Android\android-sdk\platforms", "D:/org.fdroid.k9_17046.apk");
    analyzer.getConfig().setTaintAnalysisEnabled(false);
    analyzer.calculateSourcesSinksEntryPoints(Collections.emptySet(), Collections.emptySet());
    analyzer.runInfoflow();

    // Iterate over the callgraph
    for (Iterator<Edge> edgeIt = Scene.v().getCallGraph().iterator(); edgeIt.hasNext(); ) {
        Edge edge = edgeIt.next(); SootMethod smSrc = edge.src(); Unit uSrc = edge.srcStmt();
        SootMethod smDest = edge.tgt();
        System.out.println("Edge from " + uSrc + " in " + smSrc + " to " + smDest);
    }
}
Whole Program Analysis - Recap

- Enable whole program mode
- Enable callgraph construction
- Use a whole-program pack for your code

- Callgraph: Which statement calls which method?
- Reachable methods

- PointsTo information
- Inter-procedural alias analysis
Static Data Flow Analysis

- Given a set of sources of sensitive information
- Given a set of untrusted sinks
- Check whether information derived from a source arrives at a sink

Examples:
- Device ID is sent via SMS
- Address book is uploaded to the Internet
- Location data is shared with advertisement provider
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    int x = 4 + Math.random(10);
    if (x < 10) {
        TelephonyManager tm = (TelephonyManager) getSystemService(TELEPHONY_SERVICE);
        String id = tm.getDeviceId();
        Log.d("DroidBench", id);
    }
}
The FlowDroid Data Flow Tracker

- **Inputs**
  - Sources
  - Sinks
  - Program to analyze

- **Outputs**
  - Source-to-sink mappings in the program

- **Targets**
  - Android apps
  - Java programs
The FlowDroid Data Flow Tracker (2)

```java
public static void main(String[] args) throws IOException, XmlPullParserException {
    // Configure sources and sinks
    Set<AndroidMethod> sources = Collections.singleton(new AndroidMethod("getDeviceId", "java.lang.String", "android.telephony.TelephonyManager"));
    Set<AndroidMethod> sinks = Collections.singleton(new AndroidMethod("d",
        Collections.singletonList("java.lang.String"), "int", "android.util.Log"));

    // Start the data flow analysis
    SetupApplication analyzer = new SetupApplication("C:\Program Files (x86)\Android\android-sdk\platforms", "app.apk");
    analyzer.calculateSourcesSinksEntrypoints(sources, sinks);
    InfoflowResults results = analyzer.runInfoflow();

    // Print the results
    for (ResultSinkInfo sink : results.getResults().keySet())
        for (ResultSourceInfo source : results.getResults().get(sink))
            System.out.println("Found a connection between " + source + " and " + sink);
}
```
FlowDroid Behind The Scenes

1. Analyze App Configuration
2. Create Dummy Main Method
3. Taint Tracking
4. Build src-sink connections
Analyzing App Configuration

• Manifest File Declares Components
  • Activities
  • Services
  • Content Providers
  • Broadcast Receivers

• Every Component Type Has Its Own Lifecycle

• Create a First Version of the Dummy Main Method
i = 0;
l1: if (i == 0) goto l9;  // Skip the activity

Activity1 act1 = new com.ext.Activity1();
act1.onCreate(...); act1.onStart;
l2: act.onResume();
...
act1.onPause(...);
l1: if (i == 1) goto l2;
act1.onStop();
act1.onDestroy();
if (i == 2) goto l1;  // Run activity again
Modeling The Android Lifecycle

- No main method in Android apps
  - App overwrites framework methods called by the OS
  - Plug-in model rather than standalone program

- For analysis: Emulator main() method
  - Use opaque predicates to simulate all possible control flow paths

- Opaque predicate: Non-evaluated condition
  - Analyzer assumes both outcomes are possible
  - Use gotos to simulate loops of callbacks
FlowDroid Analysis Process

- Analyze App Configuration
- Create Dummy Main Method
- Taint Tracking
- Build src-sink connections
Modeling Callbacks

- Various Techniques For Defining Callbacks
  - In Code
  - In Layout XML Files

- Not a Linear Process
  - Callbacks can define new callbacks
  - Can register and deregister callbacks at runtime

- Simplification: Callbacks never die
Modeling Callbacks

Create Dummy Method → Get Reachable Methods → Look for New Callback

New Callbacks Found → Add XML Callbacks

Done

Else
FlowDroid Analysis Process

1. Analyze App Configuration
2. Create Dummy Main Method
3. Taint Tracking
4. Build src-sink connections
void main() {

w = source();

a.b = w + "s";

foo(a);
}

void foo(z) {

k = z.b;

leak(k);

return;
}
Highly Precise Taint Tracking

- Access paths are taint abstractions
  - Models a sequence of field dereferences
  - Create access path at source
  - Track access path over control flow graph
  - Check when tainted access path arrives at sink

- Inter-procedural tracking required
  - Map base objects into callee at call site
  - Map base objects back into caller at exit site
Highly Precise Taint Tracking

- Field-Sensitive

```java
void main() {
    A a = new A();
    a.f = source();
    a.g = "public";
    leak(a.f);
}
```
Highly Precise Taint Tracking

- Field-Sensitive
- Object-Sensitive

```java
void main() {
    A a1 = new A();
    A a2 = new A();
    a1.f = source();
    a1.doLeak();
}
```
Highly Precise Taint Tracking

- Field-Sensitive
- Object-Sensitive
- Flow-Sensitive

```c
void main() {
    s = "Hello World";
    leak(s);
    s = source();
}
```
Highly Precise Taint Tracking

- Field-Sensitive
- Object-Sensitive
- Flow-Sensitive
- Context-Sensitive
  - Unlimited Depth!
  - Fix-Point iteration until no new callee-side contexts

```c
void main() {
    a = "Hello World";
    b = source();
    leak(a);
    leak(b);
}
```
Highly Precise Taint Tracking

• Need an Alias Analysis With Same Precision

• Upfront Analysis Does Not Scale

• Solution: On-Demand Alias Analysis
  • Idea: Re-use same IFDS-based analysis
  • Two interleaved solvers
  • Technique adapted from Andromeda by Tripp et. al.
Highly Precise Taint Tracking

```c
void main() {
    a = new A();
    b = a.g;
    foo(a);
    sink(b.f);
}
void foo(z) {
    w = source();
    x.f = w;
    return;
}
```

Flow Sensitivity?
Highly Precise Taint Tracking

- Forward taint tracking as usual

- Start backwards alias analysis whenever a heap object is tainted
  - All assignments to array elements and fields

- Alias analysis is an IFDS problem as well
  - Runs on inverted control flow graph
  - Scans for aliases

- Re-injects discovered aliases
  - Adds artificial edges into forward taint solver
void main() {
    w = source();
    z = x;
    leak(z.f);
    x.f = w;
    return;
}
Highly Precise Taint Tracking

- **Problem:** Re-injection of found taints can lead to false positives
  - Aliases are taints of their own right
  - Aliases are injected at positions before the point where the alias analysis was originally started
  - If passed to a sink before the starting point, a false leak occurs

- **Solution:** Use activation statements
  - Keep track of where the alias analysis was started
  - Only after this point, alias taints can cause leaks
FlowDroid Analysis Process

- Analyze App Configuration
- Create Dummy Main Method
- Taint Tracking
- Build src-sink connections
void main() {
    w = source();
    a.b = w + "s";
    foo(a);
}

void foo(z) {
    k = z.b;
    leak(k);
    return;
}

Found a connection from source() to leak()
Building Source-to-Sink Connections

- Trivial approach: Track source as part of the access path

- Problem: Different propagations through same code
  - Just for different sources -> inefficient

- Solution: Each access path only knows its predecessor

- After analysis is complete: Track predecessor chain backwards
  - Rediscover source as “start” of predecessor graph
FlowDroid: Transparent and Scalable

• Highly Configurable Framework

• Many Extension Points
  • Replacement of default algorithms possible
  • Sometimes multiple implementations already exist

• Tailor FlowDroid to Specific Analysis Problem
  • E.g. Trade Precision for performance
SOOT IN REAL-WORLD PRODUCTS

Professional Android analysis based on Soot
Soot in CodeInspect

- Decompile Android apps
  - Jimple and Java

- Debug Android apps
  - Directly on Jimple
  - Inject code at runtime
  - Inspect memory state

- Permission use analysis
  - Which permission is used where?
Soot in CodeInspect (2)

- Constant string analysis
  - Overview of what the app is about

- Sensitive method analysis
  - What is potentially critical?

- App manipulation
  - Inject monitoring code
  - Remove anti-analysis techniques
Soot in CodeInspect (3)
Recap

- Static analysis with Soot
  - Intra-procedural
  - Inter-procedural
THANK YOU FOR YOUR ATTENTION