EECS 448 Smartphone Security
Android Permissions

Xusheng Xiao

Electrical Engineering and Computer Science
Case Western Reserve University

Last Lecture: Why Unnecessary Permissions

- ``Related'' Permission names
  - MOUNT_UNMOUNT_FILESYSTEMS for android.intent.action.MEDIA_MOUNTED Intent
  - ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE permissions
    - ACCESS_NETWORK_STATE for Internet connection, e.g., celluar network or wifi
    - ACCESS_WIFI_STATE for info about WIFI
- Deputy
  - One application asks the Android Market to install another application. The sender asks for INSTALL_PACKAGES
  - Ask for CAMERA while sending intent to Camera app

Permissions: https://developer.android.com/reference/android/Manifest.permission
Intents: https://developer.android.com/reference/android/content/Intent

Last Lecture: Why Unnecessary Permissions

- Related methods
  - Setters in android.provider. Settings.Secure need WRITE_SETTING, but getters do not
- Copy and Paste
  - Registers to receive the android.net.wifi.STATE_ CHANGE Intent and requests the ACCESS_WIFI_STATE permission.
- Deprecated Permissions
  - ACCESS_GPS or ACCESS_LOCATION
- Testing Artifacts
  - ACCESS_MOCK_LOCATION
- Signature/System Permissions
  - Not working on standard handsets

Key Questions about Permissions

[Au et al. CCS'12]

- Are there any redundant permissions?
- Are undocumented APIs used?
  - Undocumented APIs are APIs that are not listed in the Android API reference
- How complex is the Android specification?
  - How are permission mappings interconnected?
- How has it evolved over time?

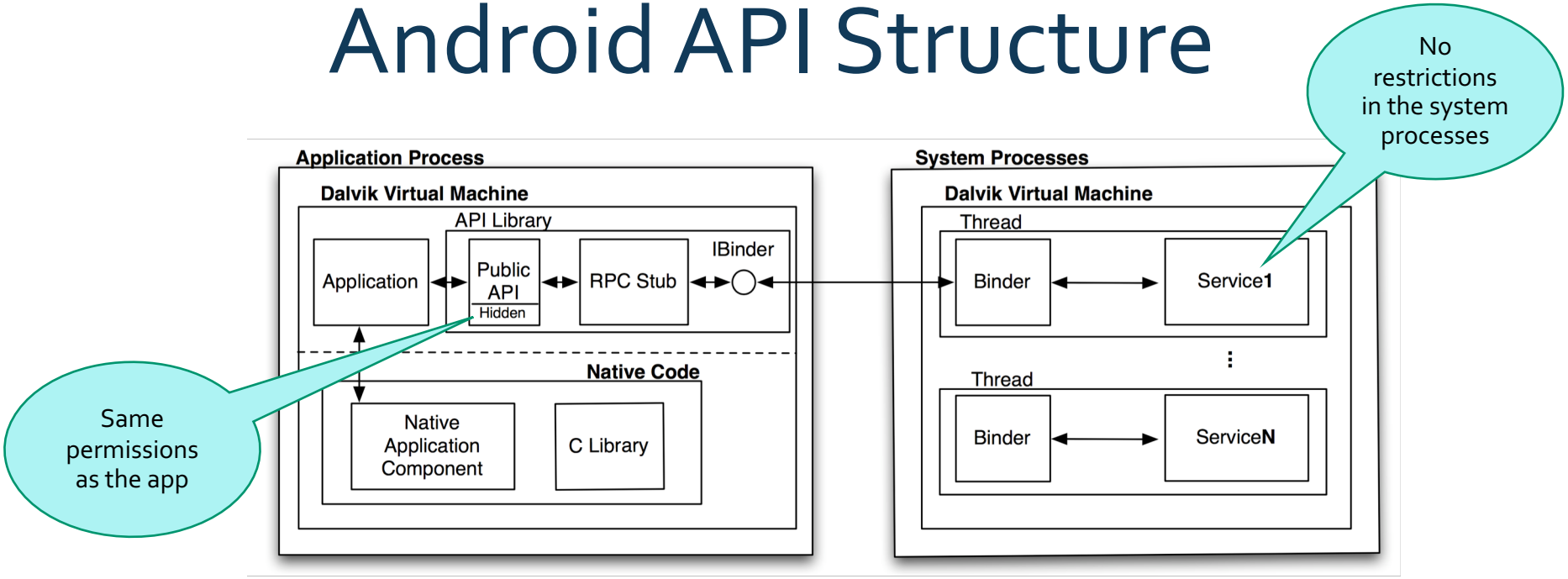
PScout: Analyzing the Android Permission Specification .
Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie.
In the Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012). October 2012.

Android Permission System

- API to permission mapping:
  - Android.net.wifi.WifiManager.reassociate();
    - CHANGE_WIFI_STATE
  - Android.telephony.TelephonyManager.getDeviceId();
    - READ_PHONE_STATE

- Complete mapping NOT available due to incomplete documentation

Android API Structure



- Public APIs → Private APIs → RPC Stub → System Processes
  - E.g., ClipboardManager.getText() → IClipboard$Stub$Proxy → ClipboardService
- Problem: Java reflection to access hidden and private classes, which may change between releases

Enforcement (Content Providers)

```
<uses-permission android:name="android.permission.READ_USER_DICTIONARY">
```

- They are protected by both static and dynamic permission checks
- They support fine granularity access controls
  - e.g., Requires default permission to visit A, public access to "content://A/b" and permission P_2 to visit part "content://A/c"

```
<provider …>
      <path-permission android:pathPrefix="/subpath1"
      android:readPermission="com.example.SUBPATH1_READ_PERMISSION"
      android:writePermission="com.example.SUBPATH1_WRITE_PERMISSION" />
      <path-permission android:pathPrefix="/subpath2"
      android:readPermission="com.example.SUBPATH2_READ_PERMISSION"
      android:writePermission="com.example.SUBPATH2_WRITE_PERMISSION" />
</provider>
```

Enforcement (Content Providers)

- Developers can do it programmatically (explicitly calls system validation mechanism for incoming queries)

```
if (ContextCompat.checkSelfPermission(thisActivity,
Manifest.permission.WRITE_CALENDAR)
        != PackageManager.PERMISSION_GRANTED) {
    // Permission is not granted
}
```

Enforcement (Intents)

Sender:

```
Intent broadcast = new Intent(this, MyBroadcastReceiver.class);
sendBroadcast(broadcast, "android.demo.mypermission");
```

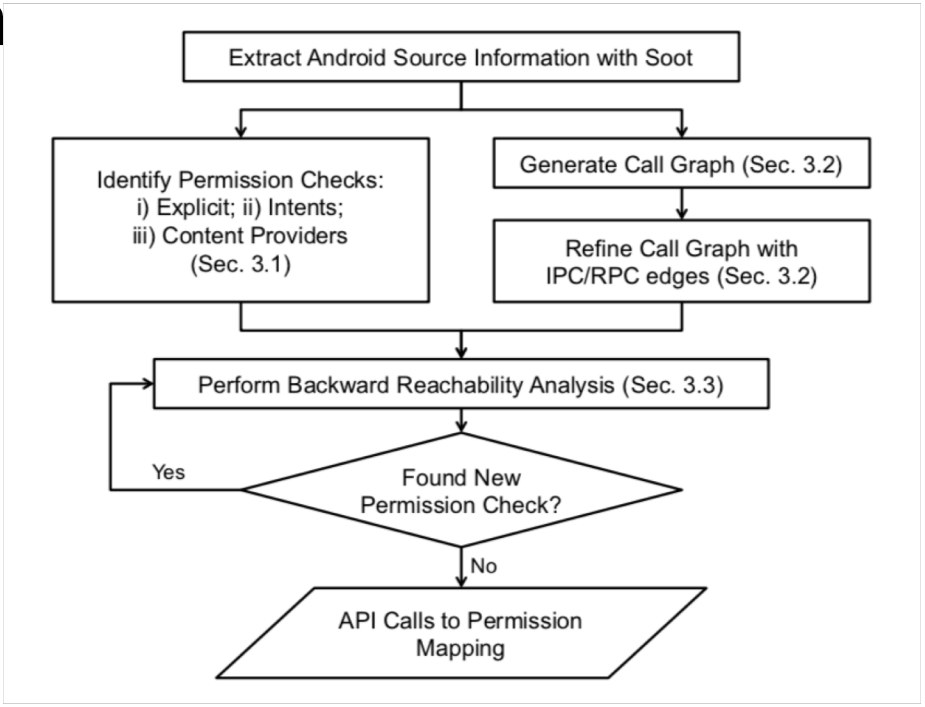
Receiver:

```
<uses-permission android:name="andro.demo.mypermission"/>
```

- Intent helps pass messages and interact between the code in different applications
- Risk: application A has been granted permission *p*, application B uses intents to access resources protected by *p* from A.
- ActivityManagerService: pass all the intents and enforce restriction policy.
  - Some intents can be sent only by apps with permissions
  - System intents can only be sent via system processes (check UID)

PScout Design and Implementation

- PScout produces a permission specification
  - API calls <-> permissions
- Three Phases
  - Permission Check Identification
  - Call graph Generation
  - Reachability Analysis



Permission Check Identification (1)

- Explicit Call
  - Permission strings and App's User ID -> checkPermission

`static int`	**`checkCallingOrSelfPermission(Context context, String permission)`** Checks whether the IPC you are handling or your app has a given permission and whether the app op that corresponds to this permission is allowed.
`static int`	**`checkCallingPermission(Context context, String permission, String packageName)`** Checks whether the IPC you are handling has a given permission and whether the app op that corresponds to this permission is allowed.
`static int`	**`checkPermission(Context context, String permission, int pid, int uid, String packageName)`** Checks whether a given package in a UID and PID has a given permission and whether the app op that corresponds to this permission is allowed.
`static int`	**`checkSelfPermission(Context context, String permission)`** Checks whether your app has a given permission and whether the app op that corresponds to this permission is allowed.

Permission Check Identification (1)

- Intents
  - Permission (send/receive) in AndroidManifest
  - Permission (send/receive) expressed programmatically
    - E.g. sendBroadcast
    - E.g. registerReceiver

Sender:

```
Intent broadcast = new Intent(this, MyBroadcastReceiver.class);
sendBroadcast(broadcast, "android.demo.mypermission");
```

Receiver:

```
<uses-permission android:name="andro.demo.mypermission"/>
```

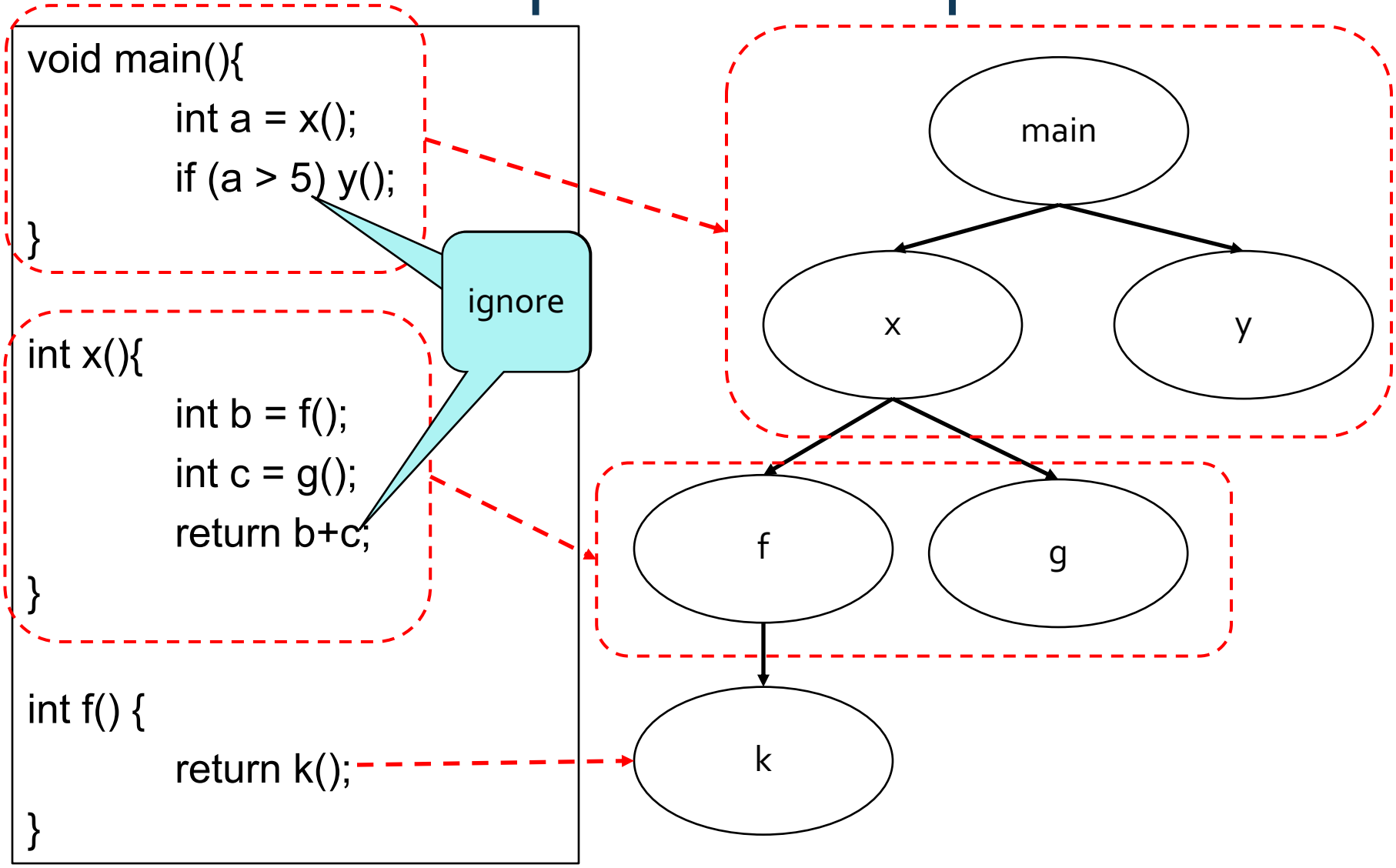
- Content Provider
  - Parse the manifest file

Call Graph

- A call graph (also known as a call multigraph) is a control flow graph, which represents calling relationships between subroutines (methods) in a computer program.
  - Each node represents a procedure and each edge <*f*, *g*> indicates that procedure *f* calls procedure *g*.
  - A cycle in the graph indicates recursive procedure calls.

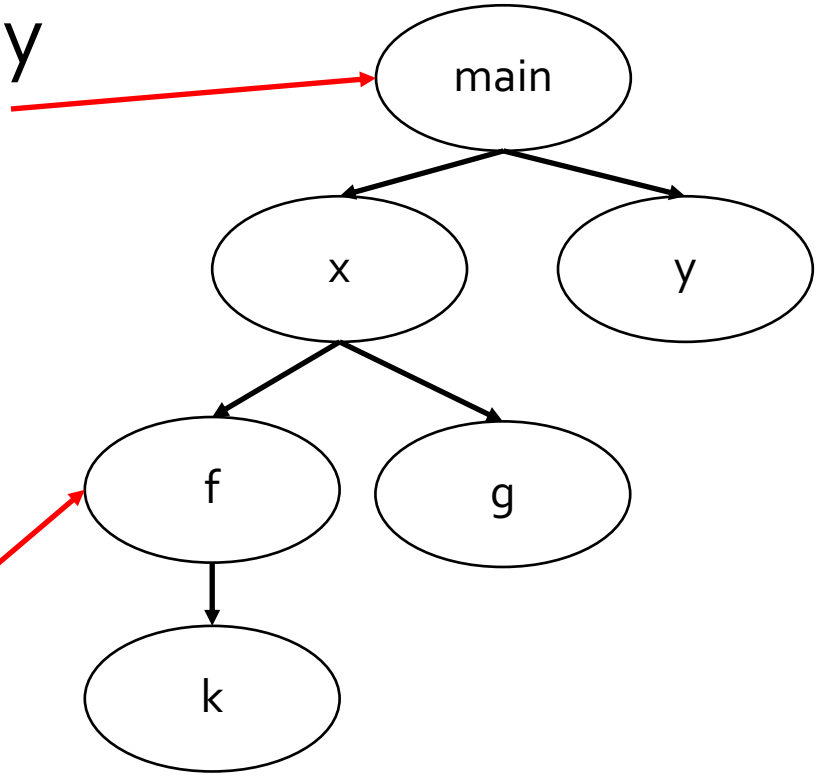
https://en.wikipedia.org/wiki/Call_graph

Example Call Graph



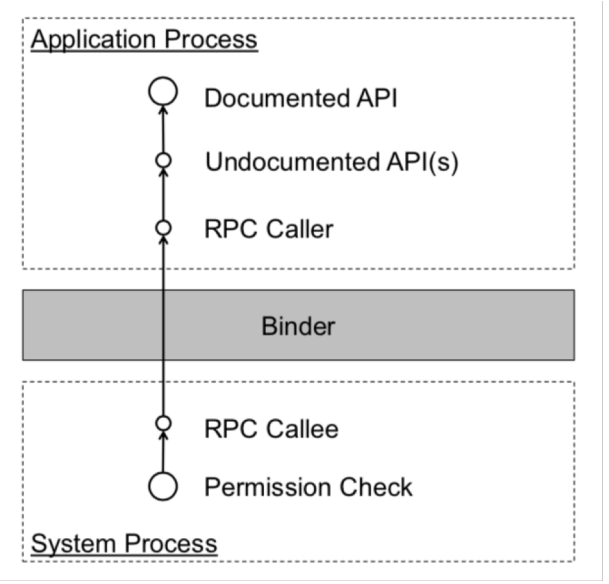
Why Do We Need Call Graph

- Top level method calls may not invoke permission check
- But several levels down, some child method calls may invoke permission check



Call Graph Generation (2)

- Call Graph Generation
  - Entire Android framework
  - Refined with RPC/IPC information
    - RPC (Remote procedure calls):
    - IPC (Inter-Process Communication): Intents, Service binders, Broadcast Receivers



Reachability: Starting Points

- *Permission check* definition:
  - An execution point in the OS after which the calling application must have the required permission
- Three Type
  - Explicit calls to *checkPermission* functions
  - Sending/receiving of specific intents
  - Accesses to specific content providers

Reachability: Stopping Conditions

- Method caller ID is temporary cleared
  - Permission enforcement always pass when caller ID is cleared in system processes

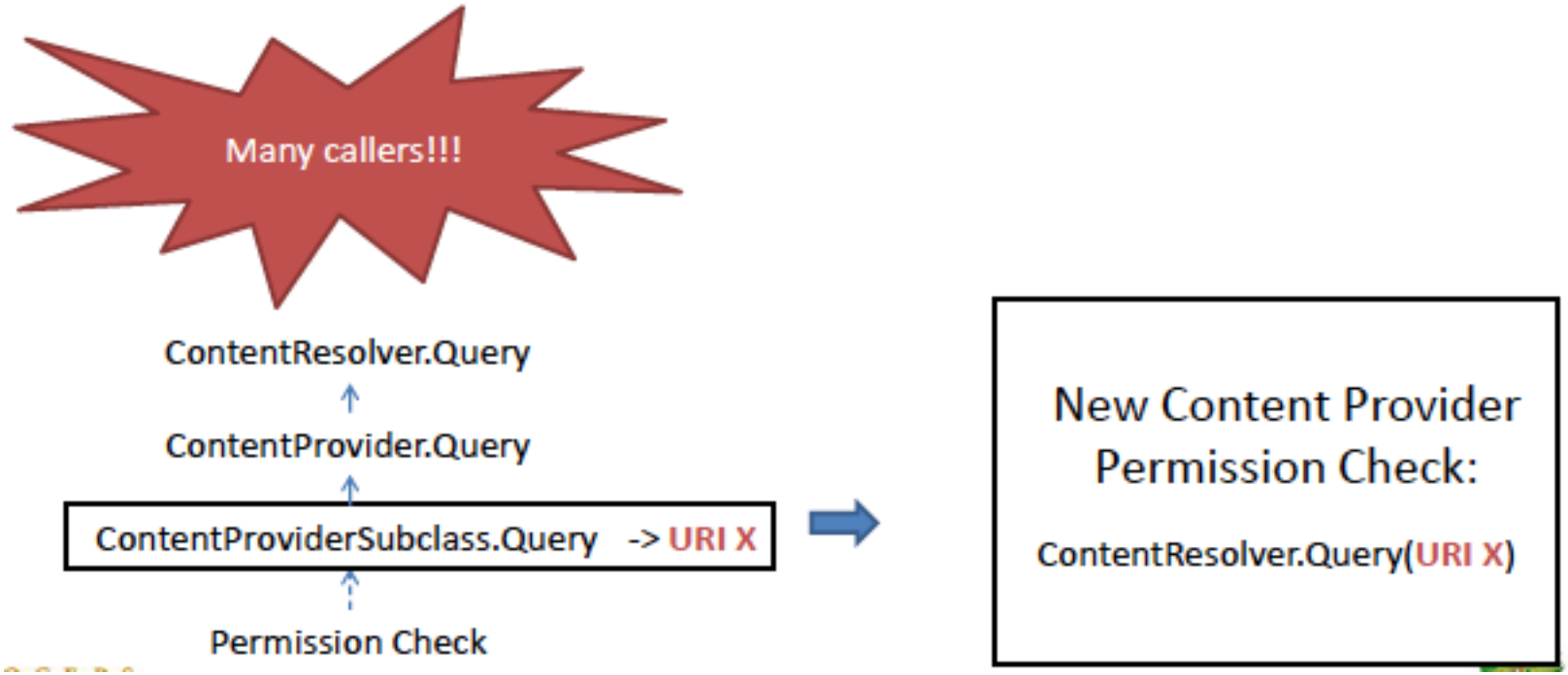
```
void Function() {
    clearCallingIdentity
    <enforce permission X>
    restoreCallingIdentity
}
```

Case 1:
Requires Permission X to proceed

Case 2:
Does not require permission to proceed

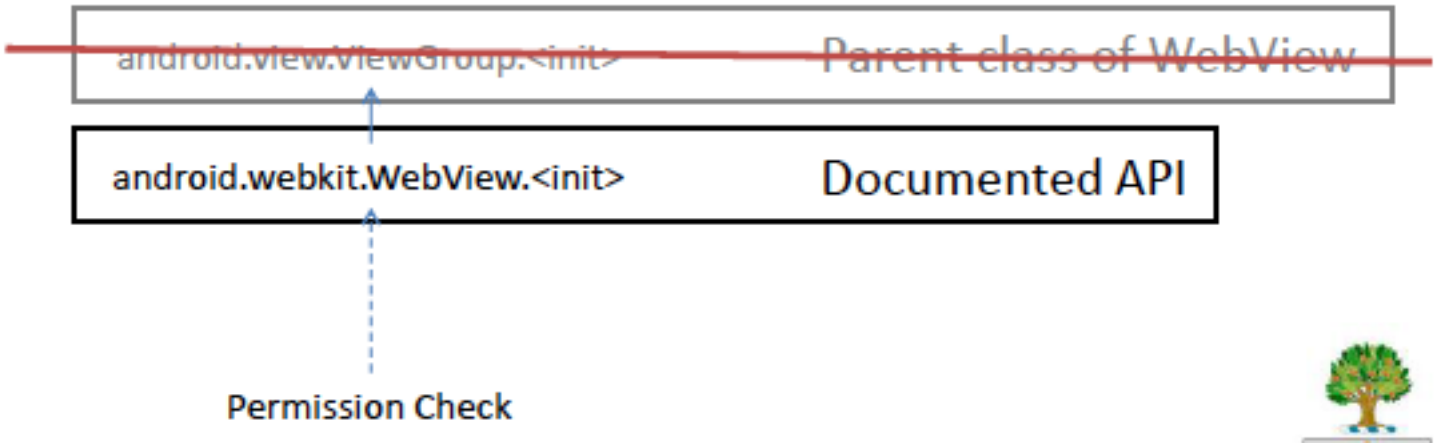
Reachability: Stopping Condition

- Reached content provider subclasses



Reachability: Stopping Conditions

- Reached generic parent classes of documented APIs



PScout Design and Implementation

- Time (33 hours)
  - Environment
    - Android 4.0 framework
    - Intel Core 2 Duo 2.53 GHz CPU
    - 4 GB memory

Key Questions

- Are there any redundant permissions?
- Are undocumented APIs used?
  - Undocumented APIs are APIs that are not listed in the Android API reference.
- How complex is the Android specification?
- How has it evolved over time?

Q1: Redundancy in Permissions?

- Conditional Probability
  - P(Y|X) = ?
  - Given an API that checks for permission X, what is the probability that the same API also check for Permission Y?
  - 79 permissions (Andorid 4.0) -> 6162 pairs of permissions

Highly Correlated Results

permission X	permission Y	P(Y|X)	P(X|Y)
KILL_BACKGROUND_PROCESS	RESTART_PACKAGES	1.00	1.00
WRITE_SOCIAL_STREAM	WRITE_CONTACTS	1.00	0.93
READ_SOCIAL_STREAM	READ_CONTACTS	1.00	0.92
USE_CREDENTIALS	MANAGE_ACCOUNTS	1.00	0.73
WRITE_SOCIAL_STREAM	READ_SYNC_SETTINGS	1.00	0.62
WRITE_SOCIAL_STREAM	READ_SOCIAL_STREAM	1.00	0.59
WRITE_CONTACTS	READ_CONTACTS	1.00	0.58
WRITE_SOCIAL_STREAM	READ_CONTACTS	1.00	0.54
WRITE_HISTORY_BOOKMARKS	READ_HISTORY_BOOKMARKS	1.00	0.39
WRITE_HISTORY_BOOKMARKS	GET_ACCOUNTS	1.00	0.30
WRITE_CALENDAR	READ_CALENDAR	1.00	0.17
ACCESS_LOCATION_EXTRA_COMMANDS	ACCESS_COARSE_LOCATION	1.00	0.05
ACCESS_LOCATION_EXTRA_COMMANDS	ACCESS_FINE_LOCATION	1.00	0.05
ADD_VOICEMAIL	READ_CONTACTS	1.00	0.04
ACCESS_COARSE_LOCATION	ACCESS_FINE_LOCATION	0.95	0.90

Table 2: Highly correlated permissions in Android. P(Y|X) denotes the conditional probability computed by taking the percentage of APIs that check for permission X that also check for permission Y.

Q1: Redundancy in Permissions?

- Redundant Relationship
  - Both permissions are always checked together
  - P(Y|X) = 100% and P(X|Y) = 100%
    - Only 1 pair found:
      - KILL_BACKGROUND_PROCESSES and RESTART_PACKAGES
        - » RESTART_PACKAGES is a deprecated permission

Q1: Redundancy in Permissions?

- Implicative Relationship
  - All APIs that check for permission X also checks for Permission Y
    - P(Y|X) = 100% and P(X|Y) = ?
      - Found 13 pairs
      - READ_ and WRITE_SOCIAL_STREAM imply READ_ and WRITE_CONTACTS
        - » Accessing contacts is requires for accessing social stream
      - Many write permissions imply read permissions for content providers
        - » E.g. WRITE_CONTACTS implies READ_CONTACTS

Q1: Redundancy in Permissions?

- Reciprocative Relationship
  - The checking of either permission by an API means the other permission is also likely checked
    - P(Y|X)>90% and P(X|Y) >90%
    - Found 1 pair:
      - ACCESS_COARSE_LOCATION vs. ACCESS_FINE_LOCATION
      - Consistent with the previous stuy

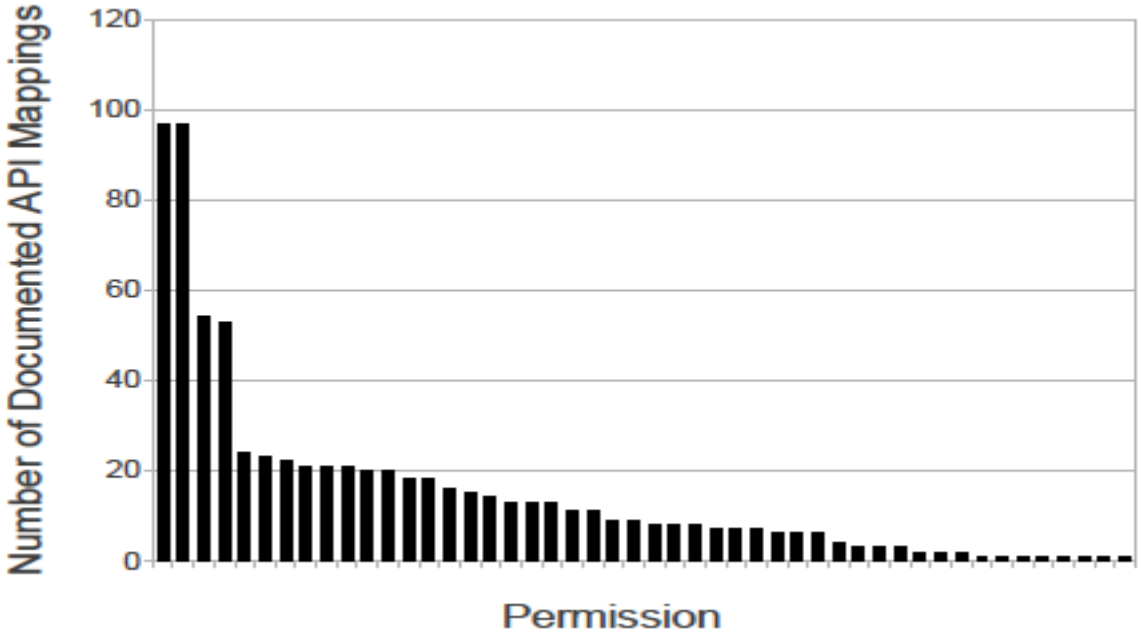
Q1: Redundancy in Permissions?

- 15/6162 all possible pairs of permission demonstrates to have close correlation.
- There is little redundancy in the Android permission system.

Q2: Undocumented API usage?

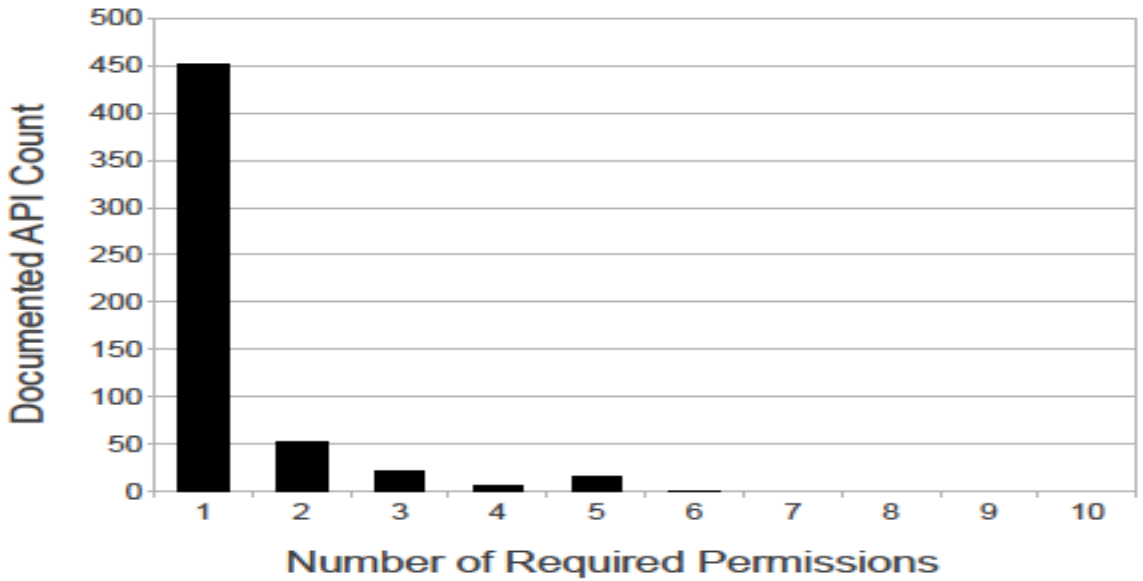
- 22-26% of the declared permissions are only checked through undocumented APIs
  - Can be hidden from most developers
  - E.g. SET_ALWAYS_FINISH, SET_DEBUG_APP are moved to system level permission in Android 4.1
- 3.7% (53/1,260 Android 2.2 apps) applications use undocumented APIs
- Undocumented APIs are rarely used in real applications, some permissions can be hidden.

Q3: Specification complexity



- 75% of permission map to <20 API calls
- Permissions guards specific functionalities

Q3: Specification complexity



- >80% APIs require only 1 permission, few need more than 3
- Sensitive APIs have relatively distinct functionality

Q3: Specification complexity

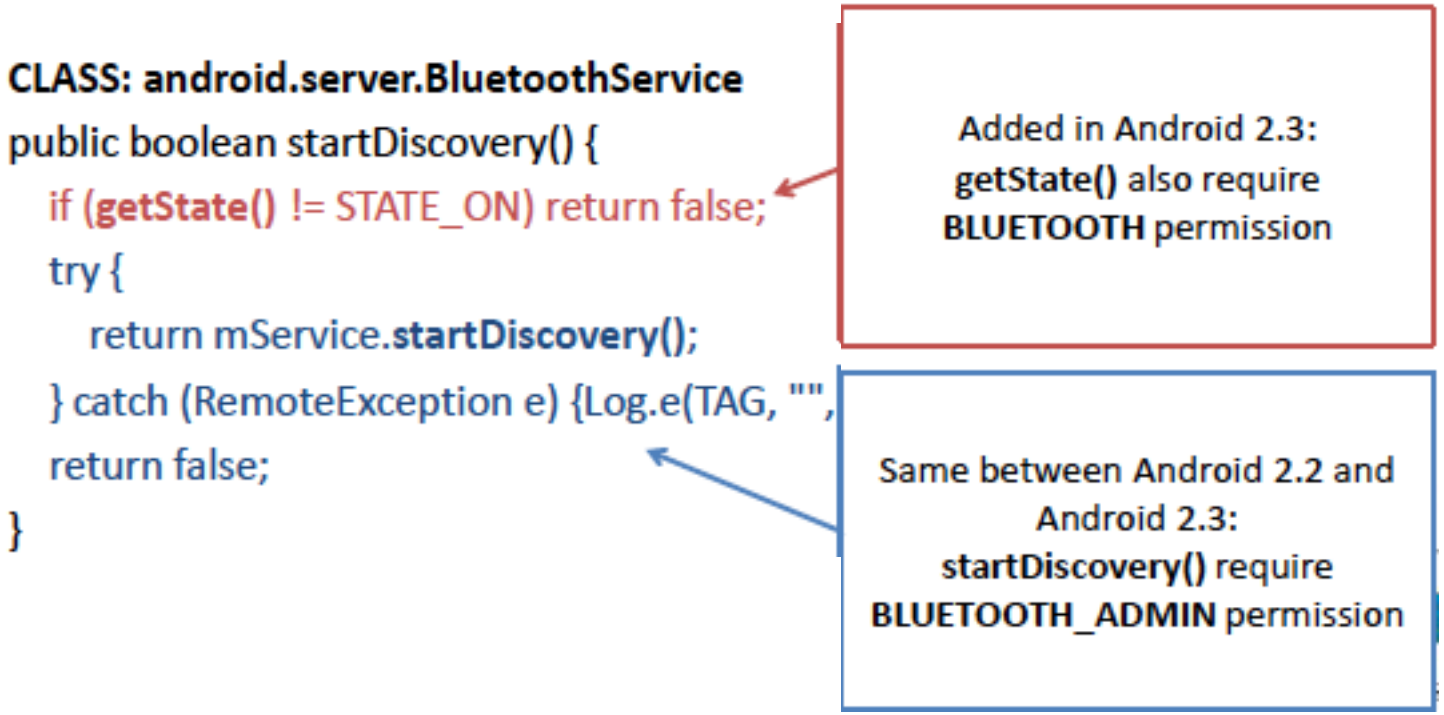
- Few overlaps in the permission mapping
- Permissions that have many API mappings tend to protect generic system resources rather than user data and have fewer permission checks.
  - SET_WALLPAPER and BROADCAST_STICKY are required for methods in the Context class which has 394 subclasses
- Android permission specification is simple.

Q4: Changes over time

- Permission checks grew proportionally with code size between 2.2 and 4.0
- More sensitive functionality are exposed through documented APIs over time
  - New APIs introduced with permissions
  - Undocumented -> documented API mapping
  - Existing APIs + new permission requirements

Q4: Changes over time

- Small changes can lead to permission changes
  - No fundamental changes in API functionality



Q4: Changes over time

- Tradeoff between fine-grain permission and permission specification stability
  - E.g. combining the BLUETOOTH and BLUETOOTH_ADMIN permission can prevent the permission change between 2.2 and 2.3 but reduces the least-privilege protection

Summary

- PScout extracts the Android permission specifications of multiple Android versions using static analysis.
  - Results show that the extracted specification is more complete than existing mappings
  - Error from static analysis imprecision is small
- There is little redundancy in the Android permission systems.
- Few application developer use undocumented APIs which some permissions are only required through undocumented APIs.
- There is a tradeoff between fine-grain permission and permission specification stability.

Thank You !

Questions ?