# Mapping & Evolution of Android Permissions

Zach Lanier & Andrew Reiter

COUNTERMEASURE 2012

# Intro

Zach Lanier

- Security Researcher
- (Net | Web | Mobile | App) pen tester type

Andrew Reiter

- Security Researcher / Math Fan
- Former member of w00w00 & FreeBSD

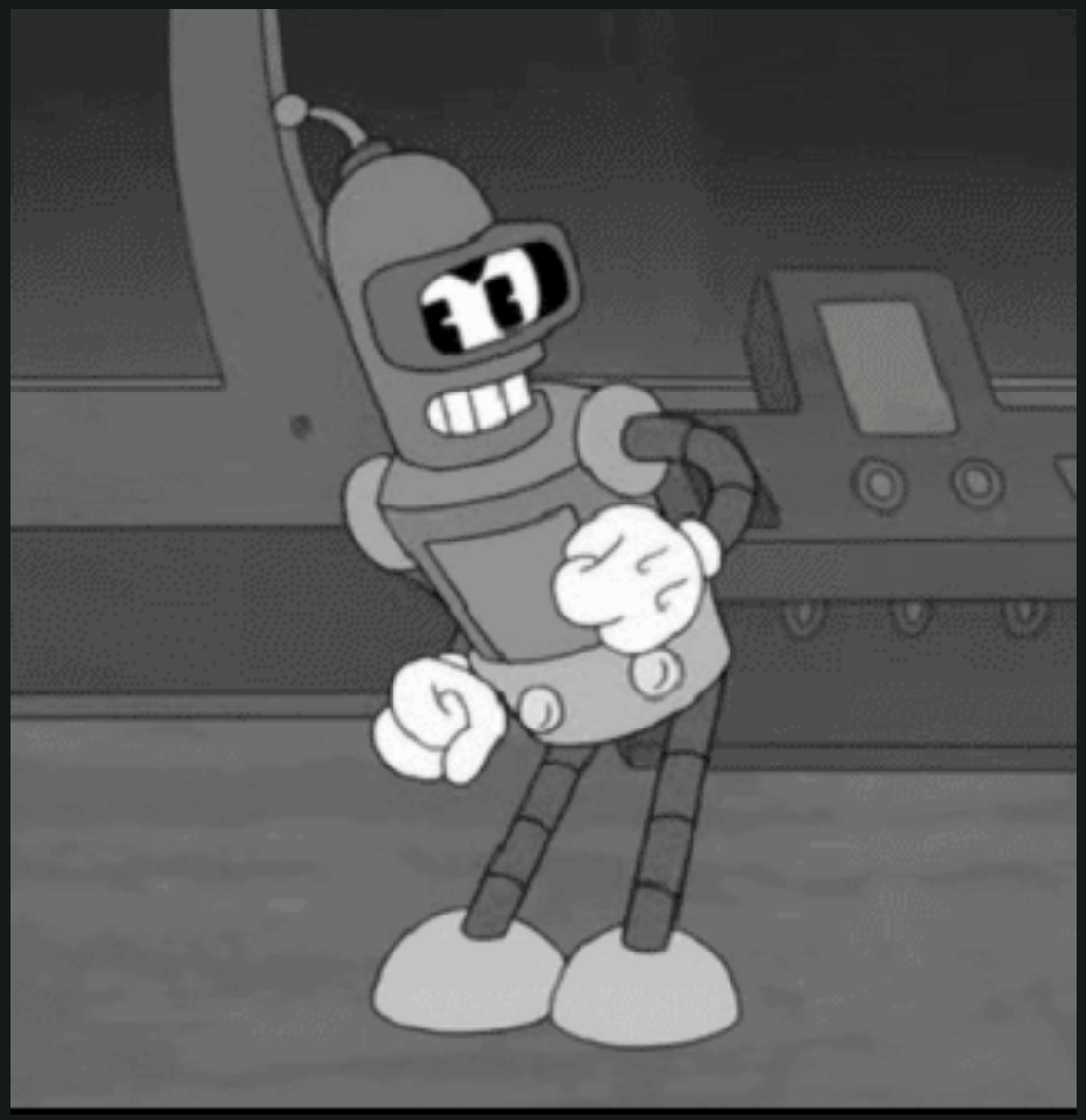
VERACODE

# Agenda

- AOSP
- Permissions, Problems, & Prior Research
- Our Approach
- Results
- Conclusion

VERACODE

# AOSP



# AOSP

- Android Open Source Project = core Android source and API under Apache 2.0 license
- We focused on API implemented in android.jar
  - Not including private/closed-source Google APIs

# Perms in Brief

- Permissions granted at app install time by Package Manager
  - INTERNET, SEND_SMS, READ_CONTACTS, VIBRATE, etc.
- Checked at time of a call:
  - Creating a Socket object, accessing Bluetooth functionality, starting an app's activity, operating on Content Provider, etc.
  - At a low-level, these checks are done in a variety of (sometimes inconsistent) ways :)

# Problems

- Reference docs include many (most!) permission requirements, but not 100% complete or 100% accurate
- No explicit map exists enabling developers to know exact permission(s)
- Implies developers must read source in order to *really* know permission requirements

# ...Their Consequences

- Numerous permissions => two main issues that arise:
  - Undergranting of privileges = reliability/usability/functionality issue (unhandled SecurityException => Force Close)
  - Overgranting of privileges = security issue (buggy, overprivileged app exploited by malicious app => privesc [permesc?])

# How does this happen?

- Most devs don't have the time, energy, patience, sanity to do their own permission map analysis
- They will do one of two things:
  - Guess and test, but this is time consuming, so...
  - Add many permissions and hope the errors go away -- what could possibly go wrong?

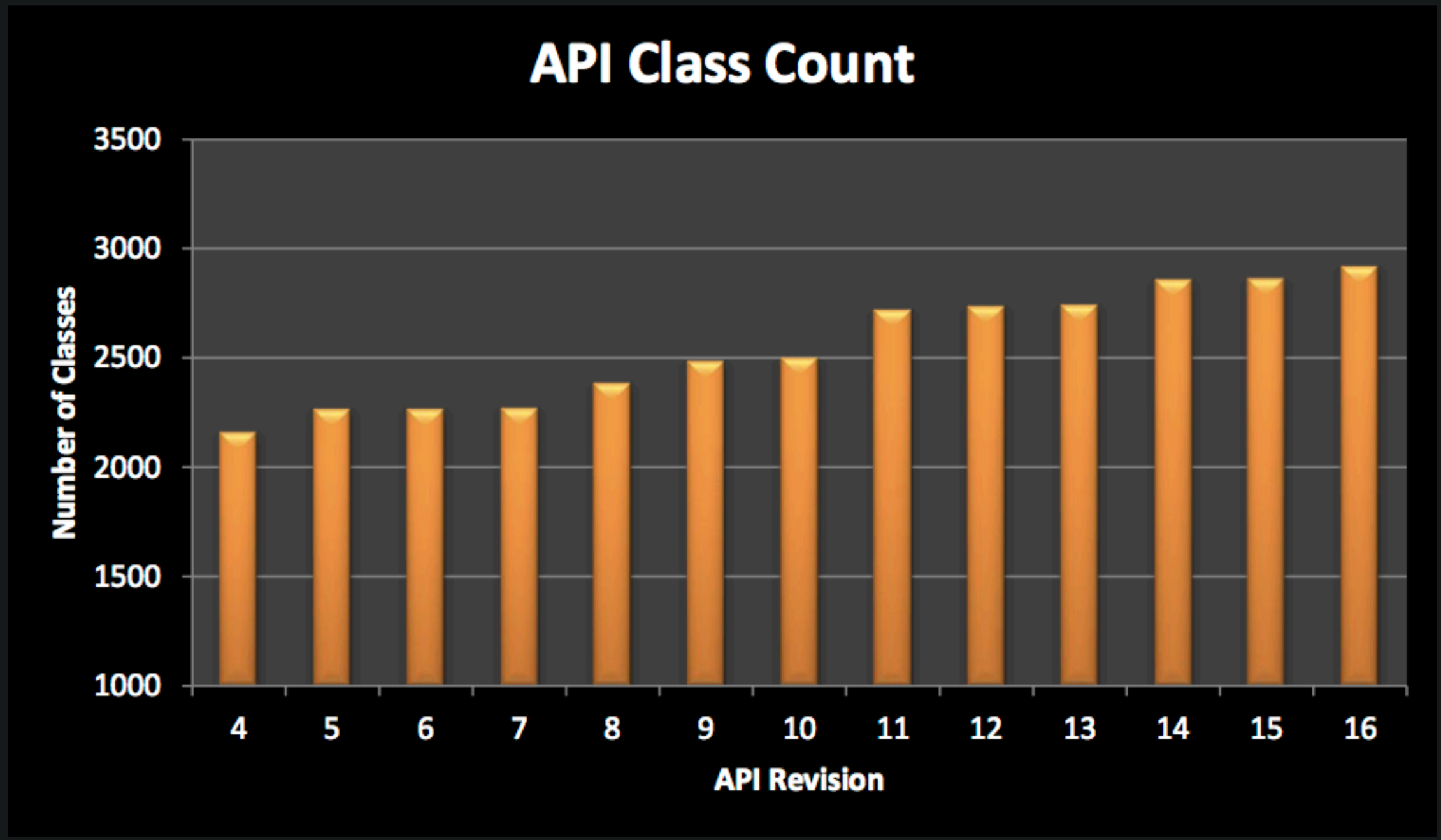
# Why no map?

- Nature of AOSP development - dynamic, evolving, no incentive
- It is not a Google priority
- Difficulty in generation of a map
- There is no "wow factor" to this map -- no 0-day ;PpPPpP

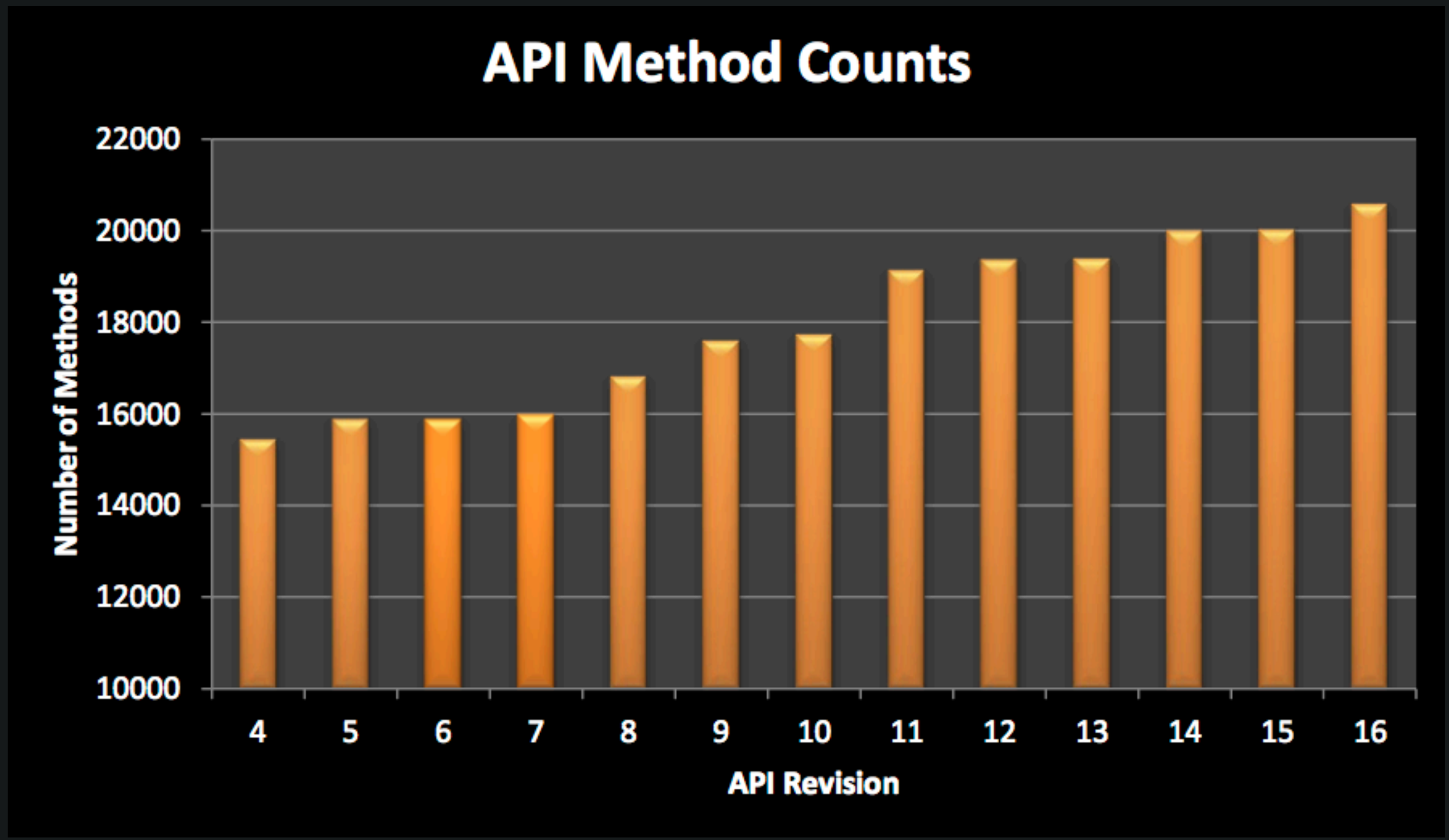
# Why So Difficult?

- API size
- Source code complexity
- Dynamic analysis difficulties

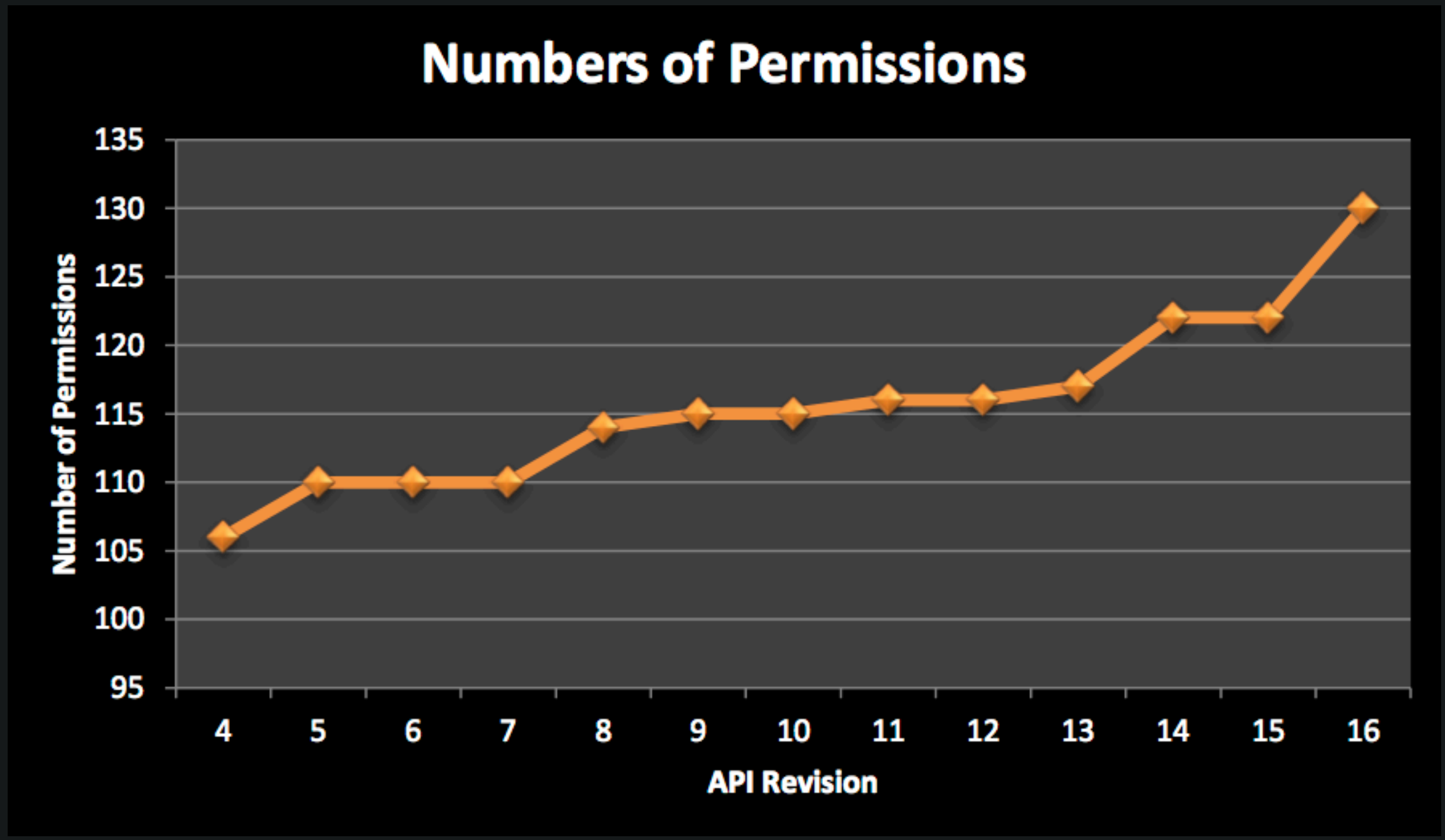
# AOSP Size


API Class Count
3500
3000
2500
2000
1500
1000
Number of Classes
4
5
6
7
8
9
10
11
12
13
14
15
16
API Revision

# AOSP Size


API Method Counts
Number of Methods
22000
20000
18000
16000
14000
12000
10000
4
5
6
7
8
9
10
11
12
13
14
15
16
API Revision

# AOSP Size


Numbers of Permissions
Number of Permissions
135
130
125
120
115
110
105
100
95
4
5
6
7
8
9
10
11
12
13
14
15
16
API Revision

# Source Complexity

- Numerous methods for permission checking, but often wrapped, so must look closer at call graph:

```
boolean checkAudioSettingsPermission(String method) {
    if (mContext.checkCallingOrSelfPermission("android.permission.MODIFY_AUDIO_SETTINGS")
            == PackageManager.PERMISSION_GRANTED) {
        return true;
    }
```

- Some checks not done using "standard" methods:

```
if (serviceInfo.applicationInfo.targetSdkVersion >= Build.VERSION_CODES.JELLY_BEAN
        && !android.Manifest.permission.BIND_ACCESSIBILITY_SERVICE.equals(
        serviceInfo.permission)) {
```

- Some routing through IBinder can be confusing and not clear as to which API method maps to which calls via IBinder

- For API level 16, roughly 9700+ files and over 2 million lines of code

# Dynamic Difficulties

- Large API to cover
  - Lots of code to generate
  - Lots of code to run to completion
- Accurate method arguments
- System state to trigger permission checks
- Emulator versus hardware
- Retrieving test results

# Prior Research: UCB

- D. Wagner, et al. ,of UC Berkeley: "permission requirement verification" (permission map) (2011)
- Dynamic approach:
  - Modified permission checks in Android to log additional/ verbose info (i.e. when exceptions were thrown)
  - Generate "zero-permission" test case apps (using Randoop [JUnit]) that call all API methods
- Poor coverage: constructors and methods requiring complex types

# Prior Research: UCB

- **Then**, decided to develop their own tool to generate test cases:
- Good:
  - Seems to be fairly accurate
  - Covers more than public methods in android.jar
- Not as Good:
  - Requires modifying Android source
  - Requires human intervention
  - Not public and only produced map for revision 8 of SDK

# Prior Research: Lelle and Luxembourg

- Bartel, Klein, et al., produced a paper describing COrrect Permission Set (COPES) tool they wrote (May 2012)
- Static analysis approach by using Soot
- Must specify entry points (i.e., root of tree)
- The other trick is: linking service calls to the API method making the request

# Prior Research: Lelle and Luxembourg

- Good:
  - No need to modify Android source
  - Code re-use via Soot is a good thing
  - Paper gives encouraging results comparison to UCB
- Not as Good:
  - Manual steps: entry point specification, service call linking
  - Inconsistent permission checks cause inaccuracies

# Our Approach

- Originally wanted to explore the feasibility of a purely static approach
- Later decided to enhance map results with hybrid (dynamic + static) approach
- Our toolset is developed in Python

# Static Analysis

- Decided to employ cscope (source code browsing/ indexing tool) to achieve this
- Modified cscope's lexer to better support Java-isms
  - e.g. “.” (period) in class name caused issues

# Static Analysis

- Generate a list of permission check methods: { checkPermission, checkCallingPermission, enforceCallingPermission, checkCallingOrSelfPermission, ... }
- Use cscope's "*Find functions calling this function*" mechanism to discover all methods calling entries in list above
- Search parent functions to discover API calls
- Label these API calls with the permission being checked
- Search for all comment-based permission requirement declarations
- Store into SQLite DB

# Dynamic Analysis

- Uses class and method information from static to generate test APKs
- Runs APKs against emulator and/or hardware
- Parses Logcat output for permissions information

# Code Generation

- Tests are generated in batches:
  - Service manager classes
  - Static method calls
  - Non-constructor-having classes
  - Generic, constructor-having classes
- Seems to stabilize testing

# Service Managers

- Classes like WifiManager, AudioManager, TelephonyManager
- Objects retrieved via our app's Context:

```
thisContext.getSystemService(android.content.Context.WIFI_SERVICE);
```

# Non-Constructor Case

- Some classes do not provide a constructor
- Must use other means of getting an object to such a class
  - Requires research / work, but usually quick
  - Example getting BluetoothDevice:

```
if (android.bluetooth.BluetoothAdapter.getDefaultAdapter() != null) {
    android.bluetooth.BluetoothAdapter tmp = android.bluetooth.BluetoothAdapter.getDefaultAdapter();
    BluetoothDevice_obj = tmp.getRemoteDevice(tmp.getAddress());
}
```

# Method Arguments

- For static calls, generic constructors, and methods, must pass arguments for successful calling
- Provide an argument pool and a method-argument builder
  - Primitive types: int, long, float, byte, boolean, etc.
  - Common classes: String, Socket, Throwable, List, URL, etc.
  - Android-specific classes: content.Context, os.Parcel, os.Bundle, os.IBinder, content.Intent, PackageManager, etc.

# Building the APKs

- Simple automation wrapper around typical APK build process:
  - ant (compile/build)
  - aapt (packaging resources/assets into APK)
  - jarsigner (digital cert / signature)
  - zipalign (optimization step)

# Running APKs

- Python code that handles:
  - emulator startup (unless a target device specified)
  - Logcat capture
  - Install, launch, uninstall of APK

# Parsing Logcat...

- *For the most part*, when a permission check is done and fails, a java.lang.SecurityException is thrown
- This exception can be seen in logcat
- This is often easily parsed for the desired permission:

```
W/System.err(  329): java.lang.SecurityException: Neither user 10036 nor current process has android.permission.DEVICE_POWER.
W/System.err(  329):    at android.os.Parcel.readException(Parcel.java:1247)
W/System.err(  329):    at android.os.Parcel.readException(Parcel.java:1235)
W/System.err(  329):    at android.os.IPowerManager$Stub$Proxy.goToSleep(IPowerManager.java:251)
W/System.err(  329):    at android.os.PowerManager.goToSleep(PowerManager.java:404)
```

# ...is not always easy

- Since the exception string is actually human coded, lack of consistency makes life hard:

W/AudioService( 64): Audio Settings Permission Denial: setSpeakerphoneOn() from pid=329, uid=10036

- Or just not using SecurityException:

W/System.err( 511): java.net.SocketException: Permission denied

# Our Results

- Good:
  - No source code modification required
  - Runtime is not long:
    - ~15 minutes for static, ~1 hour for dynamic
  - Not *much* human intervention required
  - Accuracy is good: compares well with UCB
  - Easily run against multiple SDK revisions

# Our Results

- Comparison to UCB:
  - We can only compare to their 2.2.x map (SDK rev 8)
    - but *we* can easily do different SDK revs
  - NOTE: in their map they included things that are not explicit public methods in android.jar
  - Looking just at methods we chose, we compare well

# Our Results

- Bad:
  - Does not cover all methods
    - Some JNI stubs into native code (if the perm check is done in native code) - though we're more likely to find these dynamically
    - Abstract classes
    - *Currently* limited *only* to AOSP (doesn't cover private Google or 3rd party frameworks)
  - Requires changes to cscope
  - AOSP source tree(s) required (not *hard* but...large...)

# Example: BluetoothHealth

- As a demonstration of "strong" call graph analysis (also a sanity check), found permission checks in BluetoothHealth class (note: perm reqs are in official docs)

public List<BluetoothDevice> **getDevicesMatchingConnectionStates** (int[] states) Since: API Level 14

Get a list of devices that match any of the given connection states.

If none of the devices match any of the given states, an empty list will be returned.

Requires `BLUETOOTH` permission. This is not specific to any application configuration but represents the connection state of the local Bluetooth adapter for this profile. This can be used by applications like status bar which would just like to know the state of the local adapter.

public boolean **unregisterAppConfiguration** (BluetoothHealthAppConfiguration config) Since: API Level 14

Unregister an application configuration that has been registered using `registerSinkAppConfiguration(String, int, BluetoothHealthCallback)`

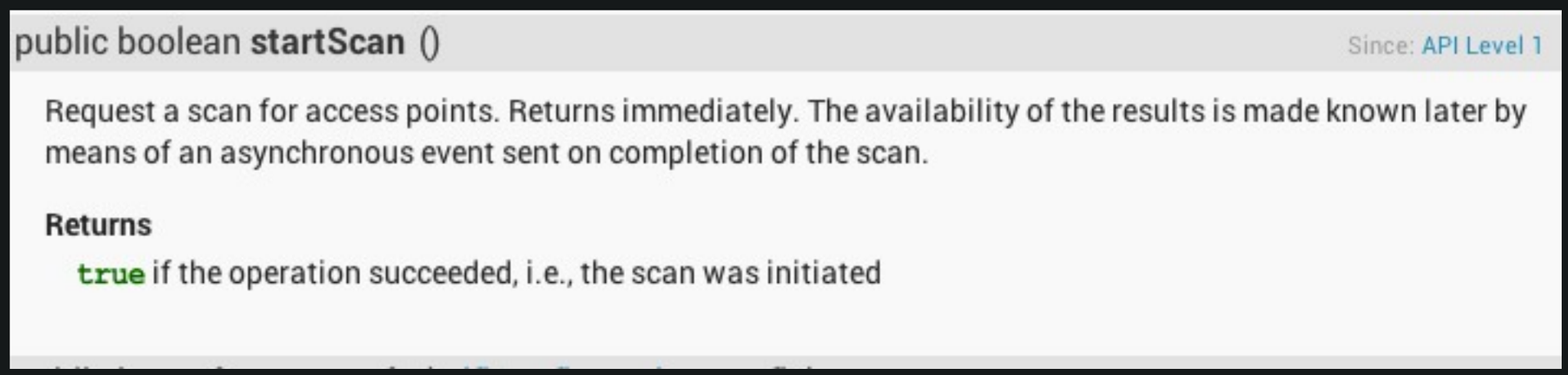
Requires `BLUETOOTH` permission.

# Example: BluetoothHealth

```
(15) android.bluetooth.BluetoothHealth.connectChannelToSource(android.bluetooth.BluetoothDevice, android.
    bluetooth.BluetoothHealthAppConfiguration) android.permission.BLUETOOTH <enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.disconnectChannel(android.bluetooth.BluetoothDevice, android.bluetooth.
    BluetoothHealthAppConfiguration, int) android.permission.BLUETOOTH <enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.getConnectedDevices() android.permission.BLUETOOTH <
    enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.getConnectionState(android.bluetooth.BluetoothDevice) android.permission.
    BLUETOOTH <enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.getDevicesMatchingConnectionStates(int[]) android.permission.BLUETOOTH <
    enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.getMainChannelFd(android.bluetooth.BluetoothDevice, android.bluetooth.
    BluetoothHealthAppConfiguration) android.permission.BLUETOOTH <enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.registerSinkAppConfiguration(java.lang.String, int, android.bluetooth.
    BluetoothHealthCallback) android.permission.BLUETOOTH <enforceCallingOrSelfPermission>
(15) android.bluetooth.BluetoothHealth.unregisterAppConfiguration(android.bluetooth.
    BluetoothHealthAppConfiguration) android.permission.BLUETOOTH <enforceCallingOrSelfPermission>
```

# Example: WiFiManager

- We see inconsistencies between documentation and implementation for some methods in WiFiManager
- For example, documentation does not mention perm requirements for startScan()



public boolean **startScan** ()

Since: API Level 1

Request a scan for access points. Returns immediately. The availability of the results is made known later by means of an asynchronous event sent on completion of the scan.

**Returns**

`true` if the operation succeeded, i.e., the scan was initiated

Source: developer.android.com

# Example: WiFiManager

```
public void startScan(boolean forceActive) {
    enforceChangePermission();

    int uid = Binder.getCallingUid();
    int count = 0;
    synchronized (mScanCount) {
        if (mScanCount.containsKey(uid)) {
            count = mScanCount.get(uid);
        }
        mScanCount.put(uid, ++count);
    }
    mWifiStateMachine.startScan(forceActive);
}

private void enforceAccessPermission() {
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.ACCESS_WIFI_STATE,
                                            "WifiService");
}

private void enforceChangePermission() {
    mContext.enforceCallingOrSelfPermission(android.Manifest.permission.CHANGE_WIFI_STATE,
                                            "WifiService");

}
```

```
(15) android.net.wifi.WifiManager.disableNetwork(int) android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.disconnect() android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.enableNetwork(int, boolean) android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.getConfiguredNetworks() android.permission.ACCESS_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.getConnectionInfo() android.permission.ACCESS_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.getDhcpInfo() android.permission.ACCESS_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.getScanResults() android.permission.ACCESS_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.pingSupplicant() android.permission.ACCESS_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.reassociate() android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.reconnect() android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>>
(15) android.net.wifi.WifiManager.removeNetwork(int) android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.saveConfiguration() android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.setWifiEnabled(boolean) android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
(15) android.net.wifi.WifiManager.startScan() android.permission.CHANGE_WIFI_STATE <enforceCallingOrSelfPermission>
```

# Example: TelephonyManager

- We also see similar inconsistencies for TelephonyManager in getNeighboringCellInfo()

public List<NeighboringCellInfo> **getNeighboringCellInfo** ()

Returns the neighboring cell information of the device.

**Returns**

List of NeighboringCellInfo or null if info unavailable.

Requires Permission: (@link android.Manifest.permission#ACCESS_COARSE_UPDATES)

Source: developer.android.com

- This is a known issue (#19192 on Google Code Android bug tracker)

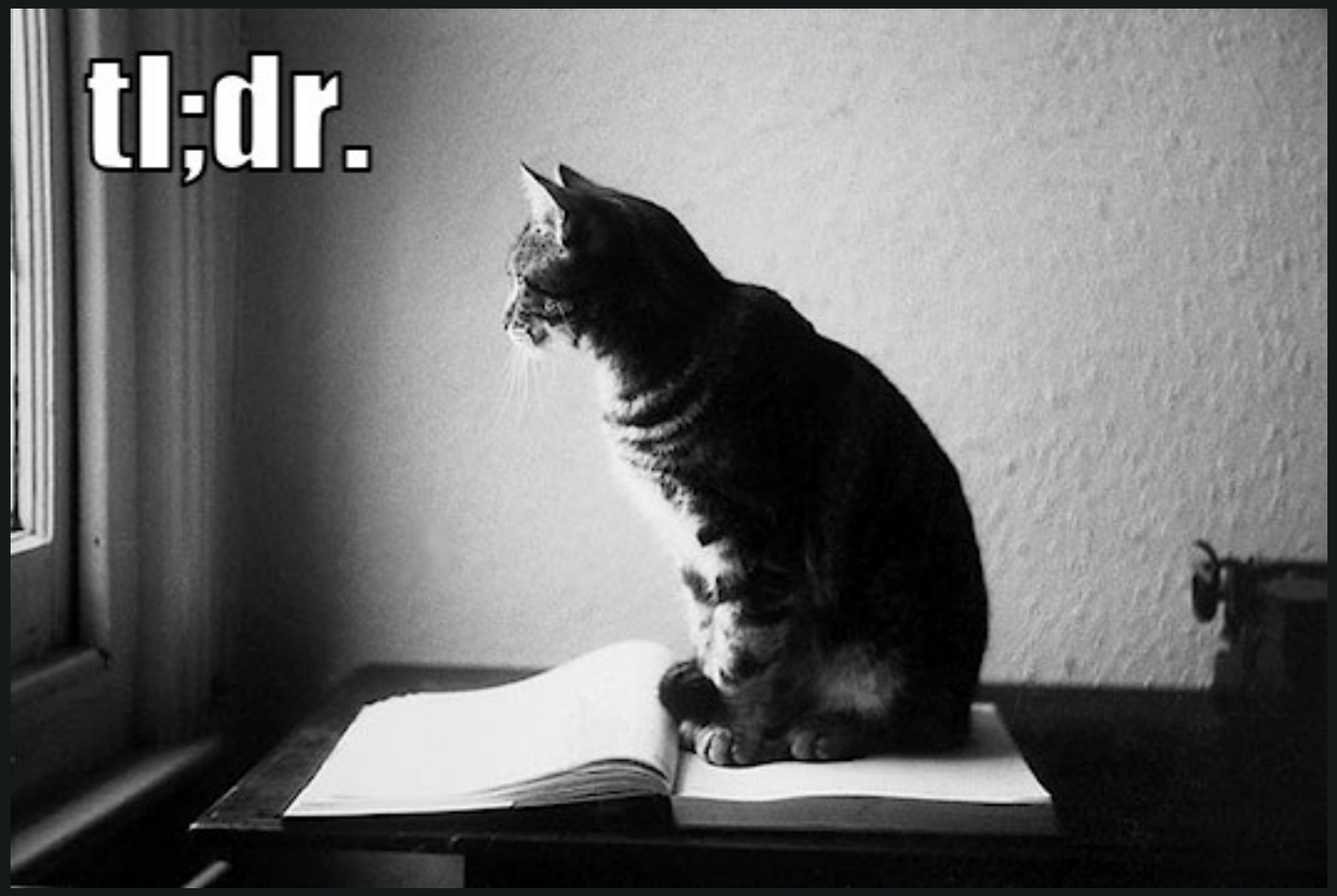
# Example: TelephonyManager

```
* <p>Requires Permission:
* {@link android.Manifest.permission#ACCESS_COARSE_UPDATES}
*/
```

```
public List<NeighboringCellInfo> getNeighboringCellInfo() {
    try {
        mApp.enforceCallingOrSelfPermission(
                android.Manifest.permission.ACCESS_FINE_LOCATION, null);
    } catch (SecurityException e) {
        // If we have ACCESS_FINE_LOCATION permission, skip the check
        // for ACCESS_COARSE_LOCATION
        // A failure should throw the SecurityException from
        // ACCESS_COARSE_LOCATION since this is the weaker precondition
        mApp.enforceCallingOrSelfPermission(
                android.Manifest.permission.ACCESS_COARSE_LOCATION, null);
    }
```

```
(15) android.telephony.TelephonyManager.getNeighboringCellInfo() android.
    permission.ACCESS_FINE_LOCATION OR android.permission.ACCESS_COARSE_LOCATION
     <enforceCallingOrSelfPermission>
```

# Conclusion



# Conclusion

- The existence of a table of API methods and the permissions required would be a boon to developers and AppSecFolk
- AppSecFolk and developers should care about these maps as there may be strange edge cases where perms are/aren't enforced consistently
- Having Google control this may be best as they have the resources to continue the effort

# Conclusion

- Good exercise in understanding what not to do
- What to do:
  - Consistent naming scheme
  - Using a central permission check clearing house API
  - Keep track of API calls that **do** require permissions

# Conclusion

- Future plans for our tool(s) include:
  - Refining static discovery
  - Expand dynamic coverage: sub-class abstract classes
    - Inverting the map (API portions *not* requiring permissions)
- See http://veracode.com/blog for upcoming post(s) with additional information (data, tools, etc.)

# Q&A / Contact

zach@n0where.org

https://twitter.com/quine

areiter@veracode.com

https://twitter.com/paucis__verbis

VERACODE

**Greetz:**

#busticati

bNull, jduck, HockeyInJune, drraid, jlamer, jono, threads

dataworm, bliss, aloria, hypatia, txs, sa7ori, vf, aempirei

WSD