Understanding and Detecting Runtime Permission Issues in Android Applications

Yepang Liu

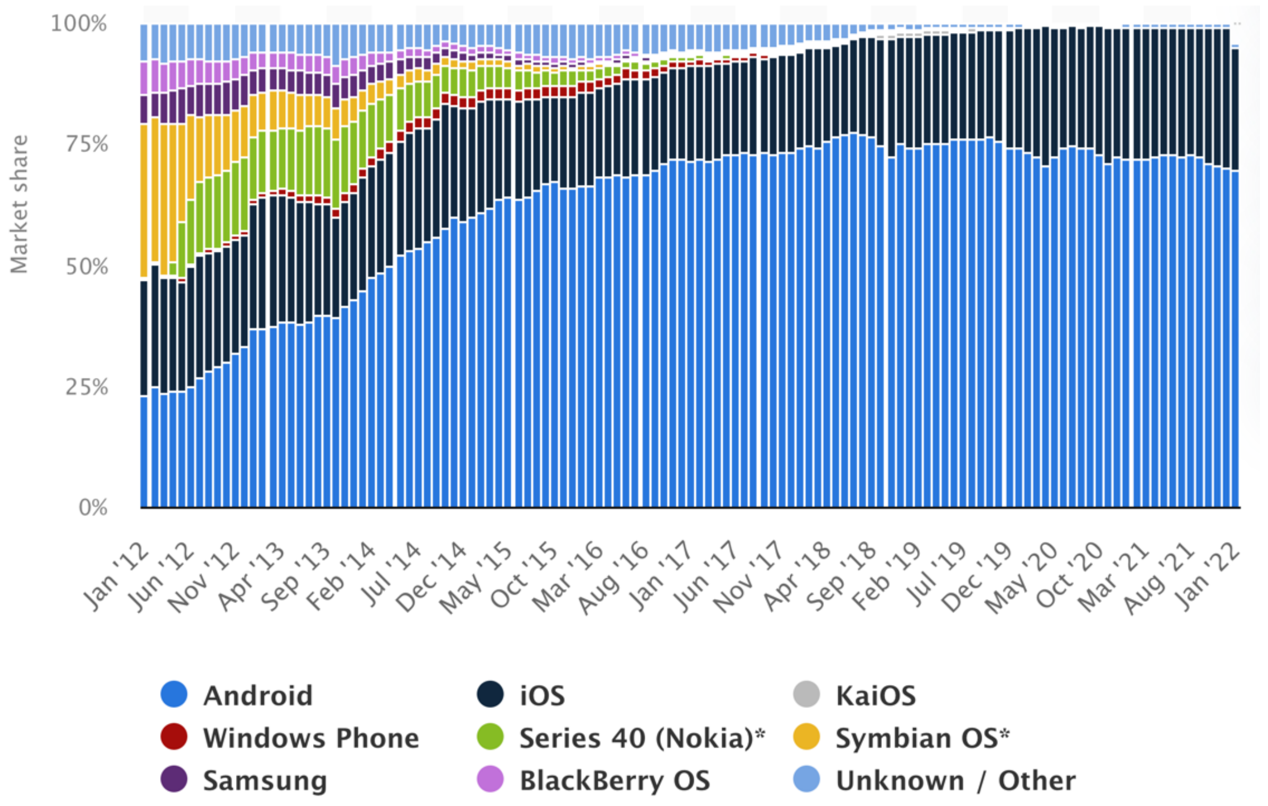
Department of Computer Science and Engineering

Southern University of Science and Technology

▶ Background

Permissions on Android

- Android is the most popular smartphone OS
  - 2+ billion users
  - 69.7% market share
- To protect user privacy, Android OS uses permissions to restrict apps to access:
  - User data (e.g., contacts, photos)
  - Sensitive functionalities (e.g., device pairing, audio recording, location sensing)

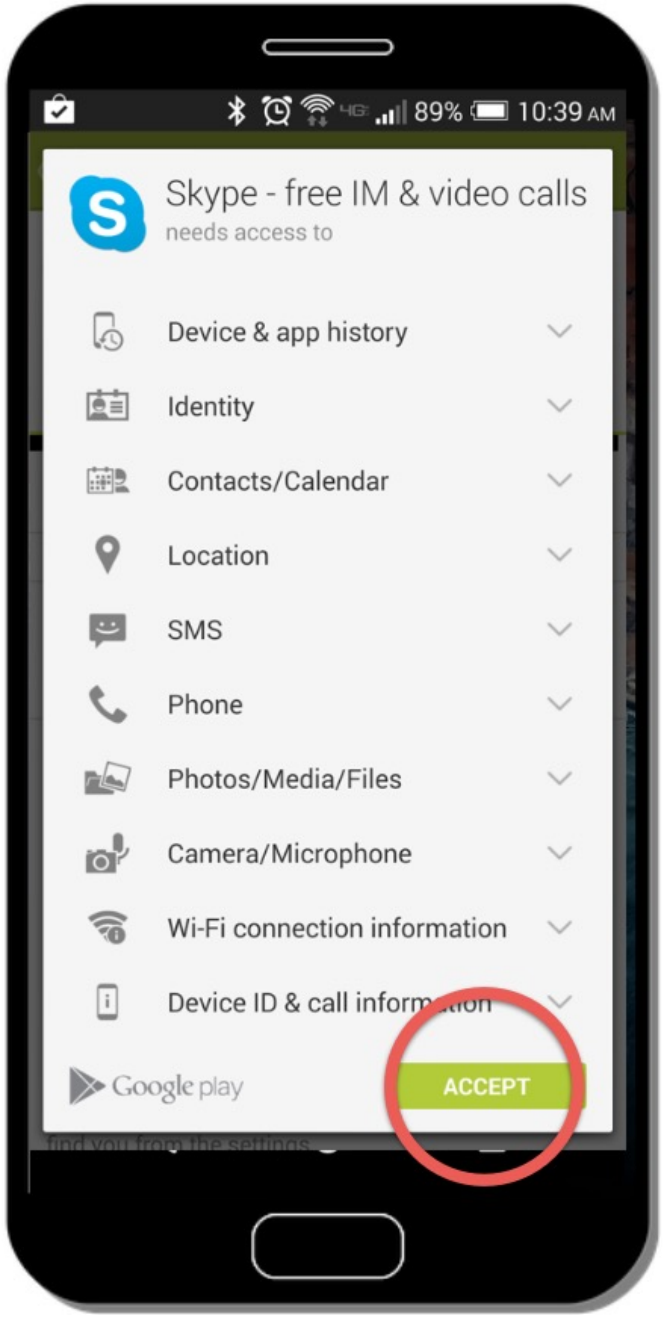


Source: https://www.statista.com/topics/876/android/#dossierKeyfigures

Static Permission Model

- Before Android 6.0

- Ask users for permissions at install time
- Cannot revoke permissions

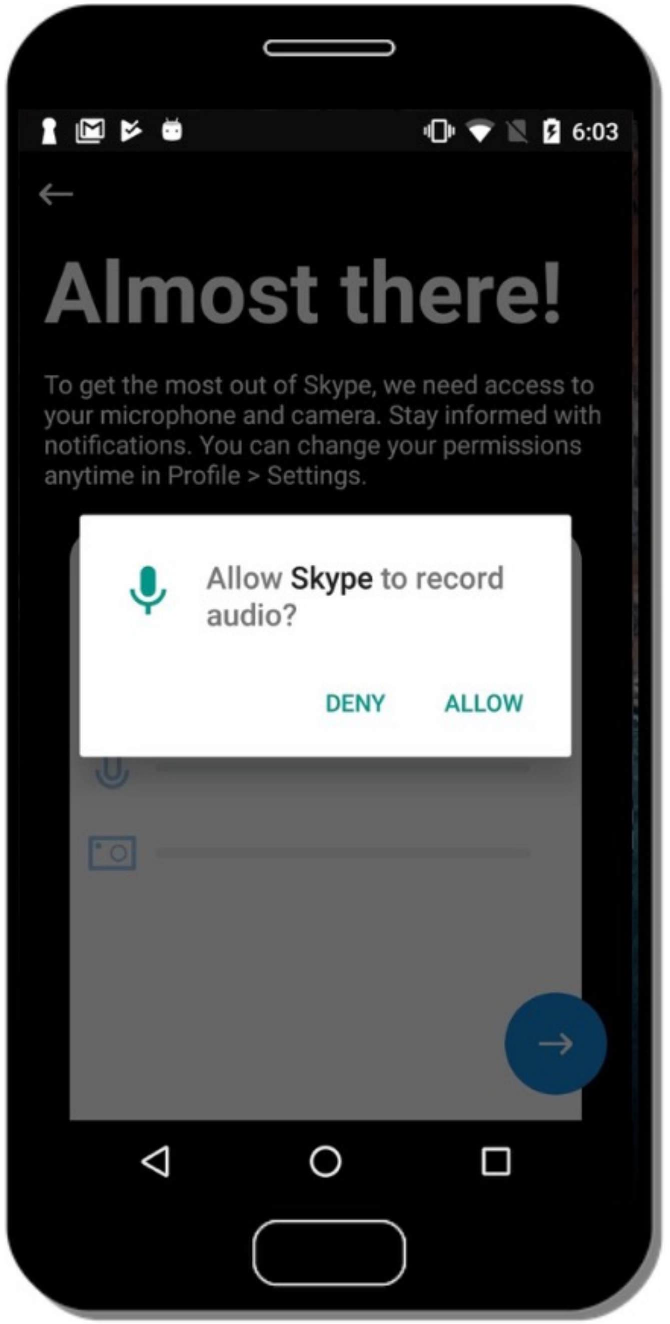

Skype - free IM & video calls
needs access to

Device & app history
Identity
Contacts/Calendar
Location
SMS
Phone
Photos/Media/Files
Camera/Microphone
Wi-Fi connection information
Device ID & call information

Runtime Permission Model

- Introduced in Android 6.0 (in 2015)

- Request users for permissions at runtime
- Users can revoke granted permissions



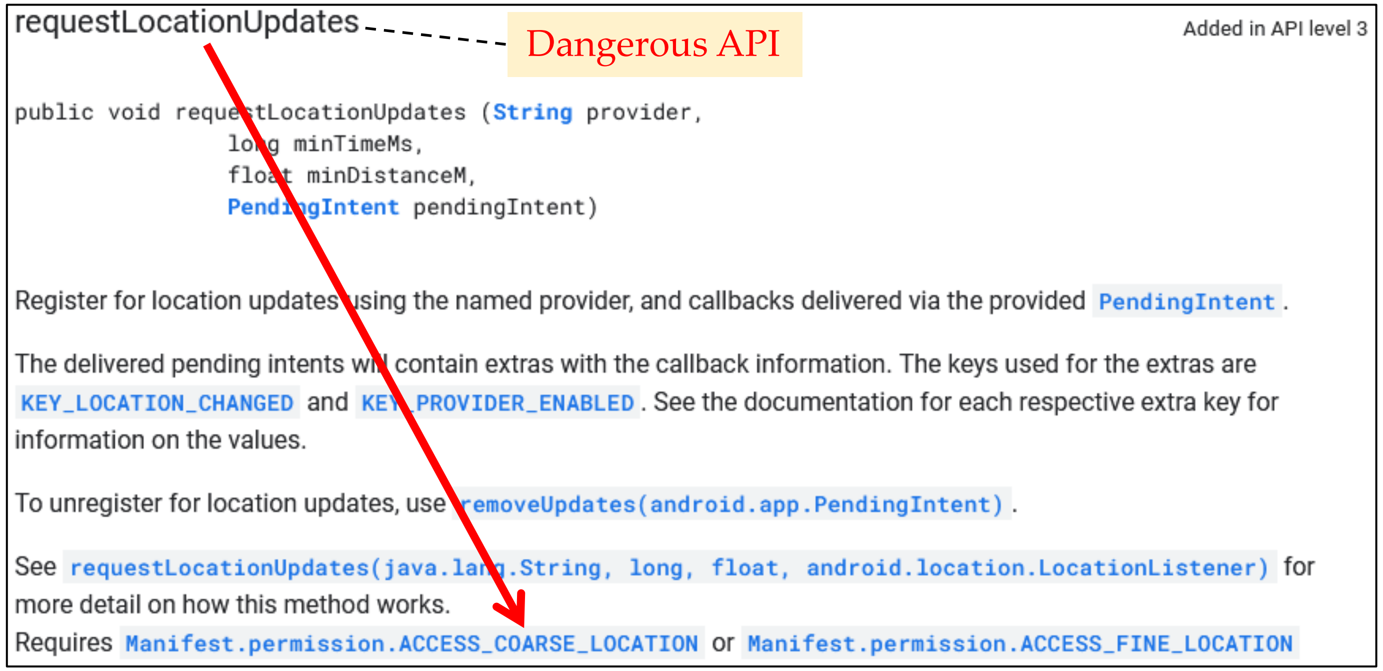
Almost there!

To get the most out of Skype, we need access to your microphone and camera. Stay informed with notifications. You can change your permissions anytime in Profile > Settings.

Allow **Skype** to record audio?

Permission Management

- **Step 1:** Understand the required dangerous permissions



requestLocationUpdates

```
public void requestLocationUpdates (String provider,
                long minTimeMs,
                float minDistanceM,
                PendingIntent pendingIntent)
```

Register for location updates using the named provider, and callbacks delivered via the provided `PendingIntent`.

The delivered pending intents will contain extras with the callback information. The keys used for the extras are `KEY_LOCATION_CHANGED` and `KEY_PROVIDER_ENABLED`. See the documentation for each respective extra key for information on the values.

To unregister for location updates, use `removeUpdates(android.app.PendingIntent)`.

See `requestLocationUpdates(java.lang.String, long, float, android.location.LocationListener)` for more detail on how this method works.

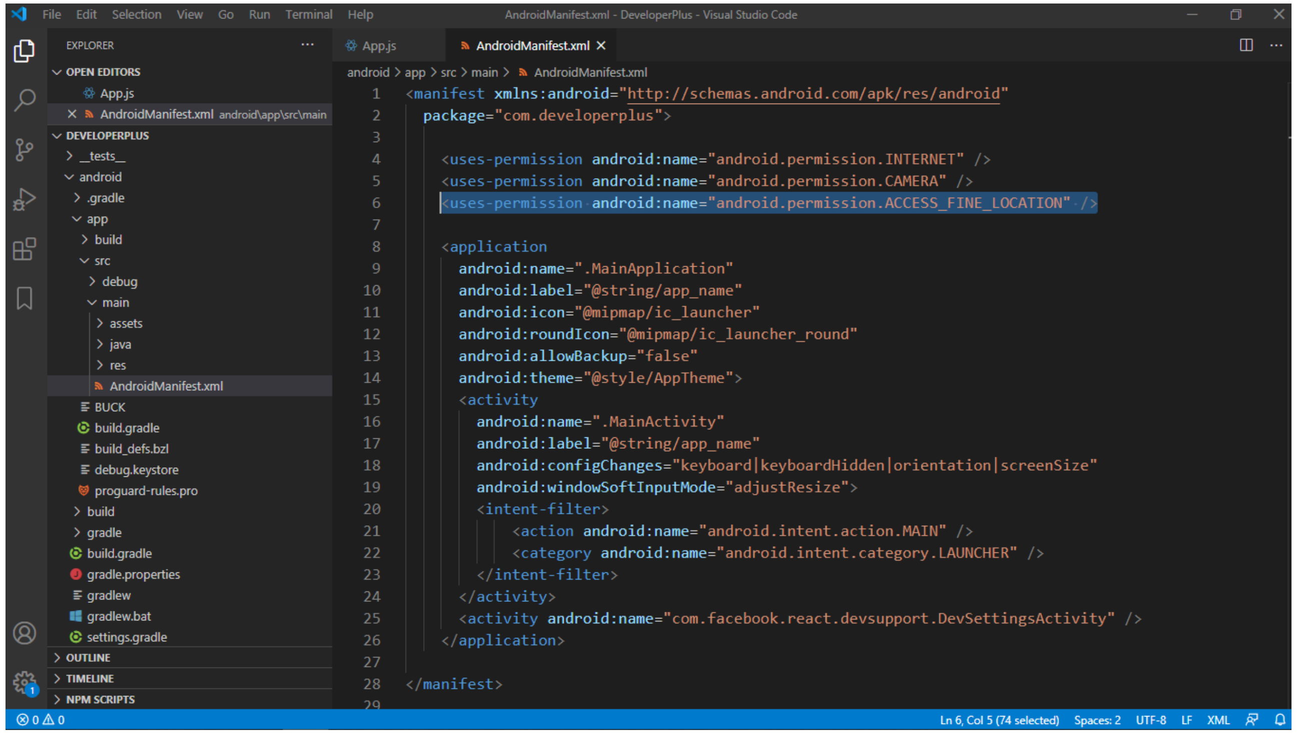
Requires `Manifest.permission.ACCESS_COARSE_LOCATION` or `Manifest.permission.ACCESS_FINE_LOCATION`

To invoke the location sensing API, the app needs to obtain either of the permissions:

- `ACCESS_COARSE_LOCATION`
- `ACCESS_FINE_LOCATION`

Permission Management

- **Step 2:** Declare the required permisions in `AndroidManifest.xml`



```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.developerplus">

    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

    <application
      android:name=".MainApplication"
      android:label="@string/app_name"
      android:icon="@mipmap/ic_launcher"
      android:roundIcon="@mipmap/ic_launcher_round"
      android:allowBackup="false"
      android:theme="@style/AppTheme">
      <activity
        android:name=".MainActivity"
        android:label="@string/app_name"
        android:configChanges="keyboard|keyboardHidden|orientation|screenSize"
        android:windowSoftInputMode="adjustResize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
      </activity>
      <activity android:name="com.facebook.react.devsupport.DevSettingsActivity" />
    </application>

</manifest>
```

To tell the Android OS that the app might need the declared permissions at runtime

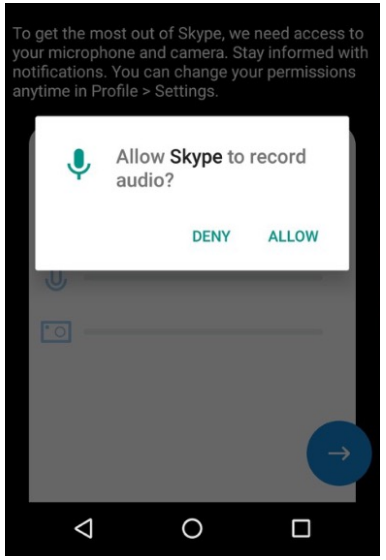
Permission Management

- **Step 3:** Check permission status before calling the dangourous API

```
if(checkSelfPermission(Manifest.permission.RECORD_AUDIO) == PackageManager.PERMISSION_GRANTED) {
```

- **Step 4:** Request the permission from users if not granted

```
String[] permissions = new String[]{Manifest.permission.RECORD_AUDIO};
requestPermissions(permissions, 1001);
```



To get the most out of Skype, we need access to your microphone and camera. Stay informed with notifications. You can change your permissions anytime in Profile > Settings.

Allow **Skype** to record audio?

Permission Management

- **Step 5:** Handle users' choices
- **Step 6:** Explain to users why the permissions are necessary…

- The permission management process is quite complicated under the new model.
- Developers (especailly novices) may often make mistakes, thus inducing various runtime permission issues.

- **Understanding Runtime Permission Issues**

Research Questions

- **RQ1:** Are runtime permission issues common in real-world Android projects?
- **RQ2:** What are the common types of runtime permission issues?

For more RQs and findings, please refer to our TSE 2022 paper:

Ying Wang, Yibo Wang, Sinan Wang, Yepang Liu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. "Runtime Permission Issues in Android Apps: Taxonomy, Practices, and Ways Forward." *IEEE Transactions on Software Engineering*, 2022.

Empirical Findings

- We studied 415 popular open-source Android projects (hosted on GitHub), which declare at least one dangerous permission

FINDINGS

1. Runtime permission issues are common. We found 199 real issues in the 415 projects.
2. The 199 real issues can be categorized into 11 types according to their root cause.

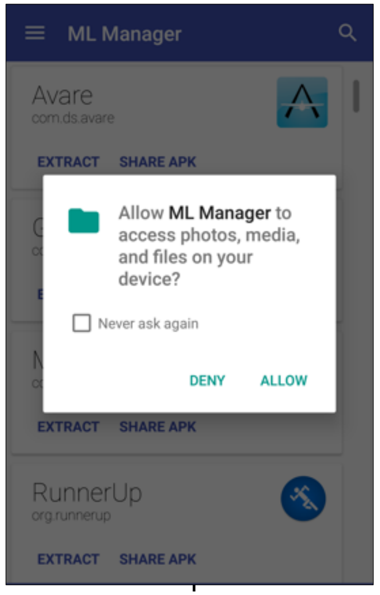
Two Common Issue Types

- **Type-1 (Missing Permission Check):**
  - A dangerous API is called without a permission check on the target Android version
- **Type-2 (Incompatible Permission Usage):**
  - A dangerous API can be called on incompatible platforms, or the evolution of permission specification is not fully handled

Type-1 Issue Example

ML Manager:

- An APK manager
- 100,000+ downloads on Google Play


ML Manager

Avare
com.ds.avare
EXTRACT SHARE APK

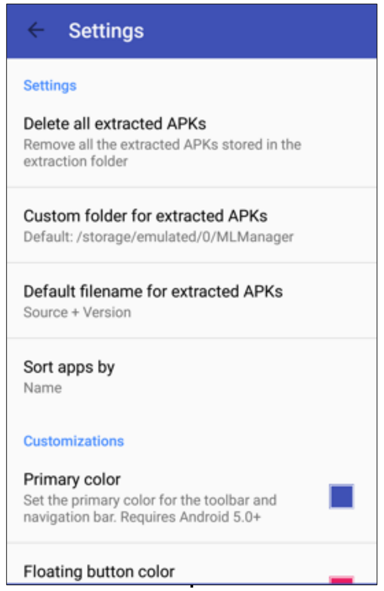
Allow **ML Manager** to access photos, media, and files on your device?

☐ Never ask again

DENY ALLOW

EXTRACT SHARE APK

RunnerUp
org.runnerup
EXTRACT SHARE APK


Settings

Settings

Delete all extracted APKs
Remove all the extracted APKs stored in the extraction folder

Custom folder for extracted APKs
Default: /storage/emulated/0/MLManager

Default filename for extracted APKs
Source + Version

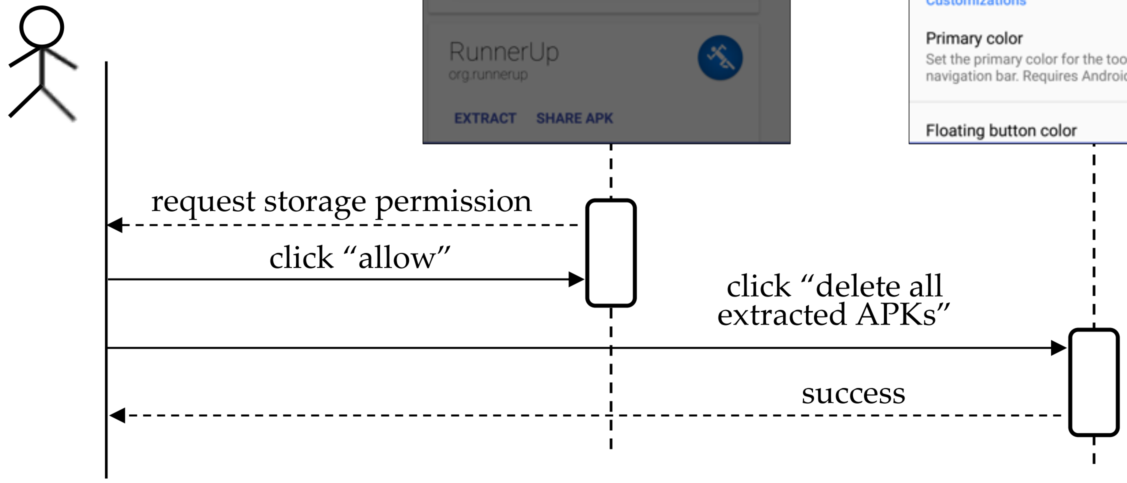
Sort apps by
Name

Customizations

Primary color
Set the primary color for the toolbar and navigation bar. Requires Android 5.0+

Floating button color

A safe scenario:

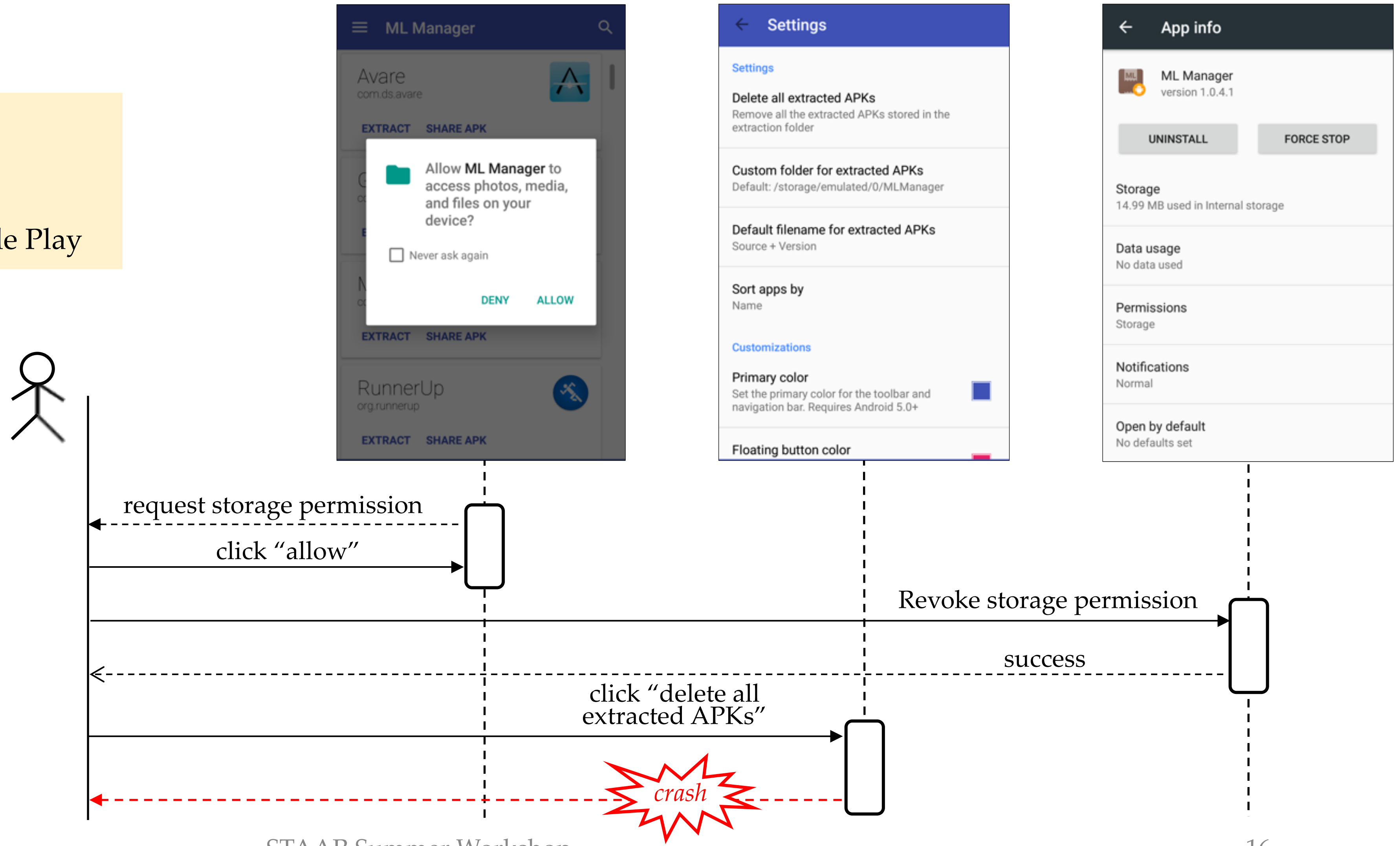


Type-1 Issue Example

ML Manager:

- An APK manager
- 100,000+ downloads on Google Play

A buggy **scenario:**



request storage permission
click “allow”
Revoke storage permission
success
click “delete all extracted APKs”
crash

ML Manager

Avare
com.ds.avare
EXTRACT SHARE APK

Allow **ML Manager** to access photos, media, and files on your device?

Never ask again

DENY ALLOW

EXTRACT SHARE APK

RunnerUp
org.runnerup
EXTRACT SHARE APK

Settings

Settings

Delete all extracted APKs
Remove all the extracted APKs stored in the extraction folder

Custom folder for extracted APKs
Default: /storage/emulated/0/MLManager

Default filename for extracted APKs
Source + Version

Sort apps by
Name

Customizations

Primary color
Set the primary color for the toolbar and navigation bar. Requires Android 5.0+

Floating button color

App info

ML Manager
version 1.0.4.1

UNINSTALL FORCE STOP

Storage
14.99 MB used in Internal storage

Data usage
No data used

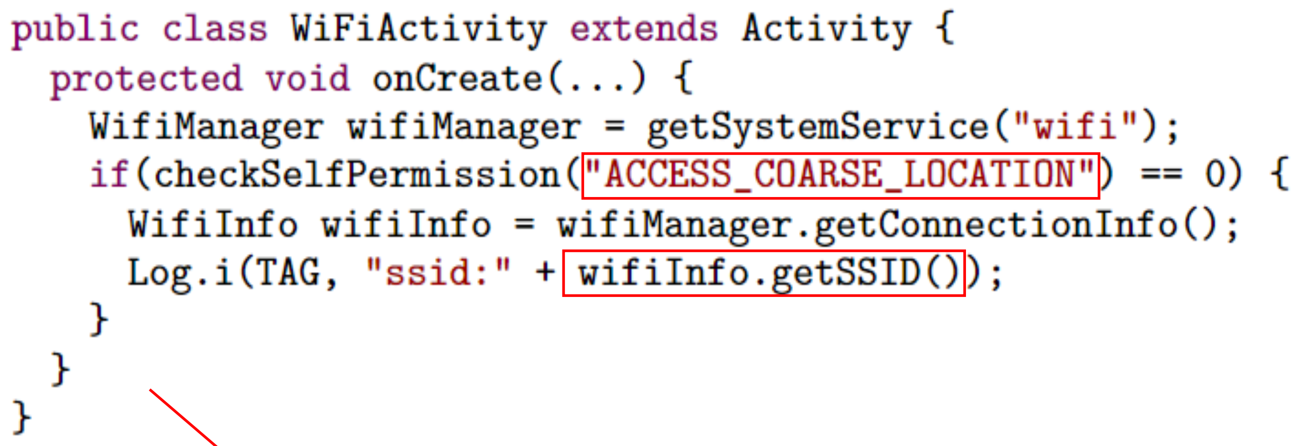
Permissions
Storage

Notifications
Normal

Open by default
No defaults set

Type-2 Issue Example

- Type-2 issues are compatibility issues caused by the evolution of the Android framework



```
public class WiFiActivity extends Activity {
  protected void onCreate(...) {
    WifiManager wifiManager = getSystemService("wifi");
    if(checkSelfPermission("ACCESS_COARSE_LOCATION") == 0) {
      WifiInfo wifiInfo = wifiManager.getConnectionInfo();
      Log.i(TAG, "ssid:" + wifiInfo.getSSID());
    }
  }
}
```

- ➢ Before API level 29, `getSSID()` requires either `COARSE` or `FINE` location permission
- ➢ Since API level 29, `getSSID()` requires `FINE` location permission

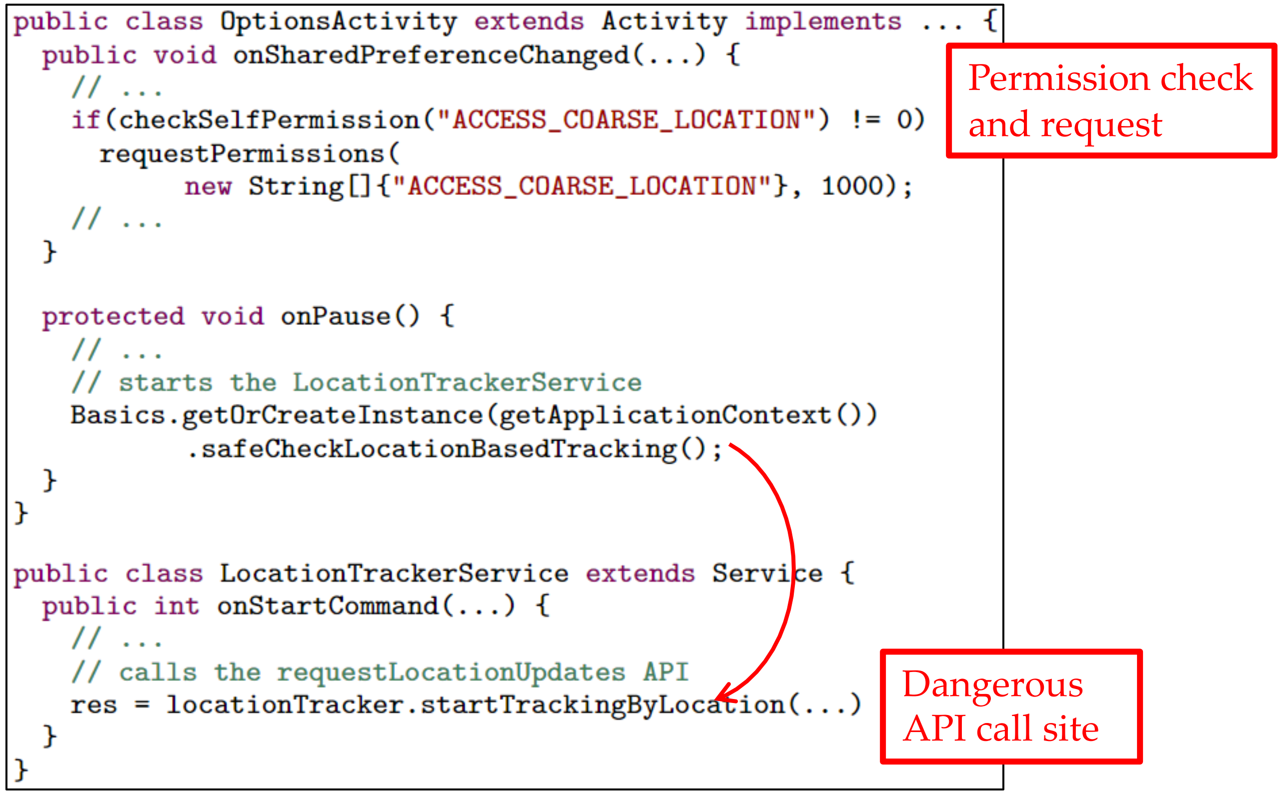
The app does not work on Android 10 (API Level 29) or newer versions but works well on older versions (prior to Android 10).

▶ Detecting Runtime Permission Issues

Technical Challenges

- **Challenge 1:** Asynchronous permission managements and usages

  - Android programs have multiple possible entry points (due to their event-driven nature)
  - Permission check/request operations and permission use (at the call site of dangerous API) can be in different callbacks, or even across different app components

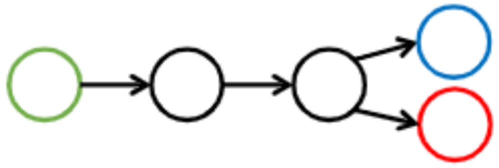


```
public class OptionsActivity extends Activity implements ... {
  public void onSharedPreferenceChanged(...) {
    // ...
    if(checkSelfPermission("ACCESS_COARSE_LOCATION") != 0)
      requestPermissions(
            new String[]{"ACCESS_COARSE_LOCATION"}, 1000);
    // ...
  }

  protected void onPause() {
    // ...
    // starts the LocationTrackerService
    Basics.getOrCreateInstance(getApplicationContext())
            .safeCheckLocationBasedTracking();
  }
}

public class LocationTrackerService extends Service {
  public int onStartCommand(...) {
    // ...
    // calls the requestLocationUpdates API
    res = locationTracker.startTrackingByLocation(...)
  }
}
```

Possible Practices: Synchronous Management


(a) Intra-procedure

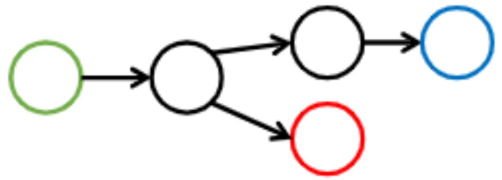
The intra-procedural case:

The permission check/request methods and the dangerous APIs are invoked in the same method

○ Normal Method ○○ Component Lifecycle Callback ○ Permission Check or Request

○ Dangerous API → Method Call Relation

Possible Practices: Synchronous Management



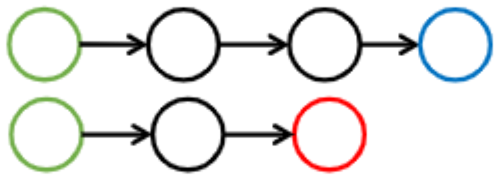
(b) Inter-procedure

The inter-procedural case:

The permission check/request method calls are put in another wrapper method

○ Normal Method
○○ Component Lifecycle Callback
○ Permission Check or Request
○ Dangerous API
⟶ Method Call Relation

Possible Practices: Asynchronous Management



(c) Inter-callback

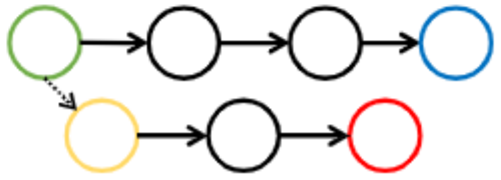
The inter-callback case:

The permission check/request methods and the dangerous APIs are invoked in different callbacks of the same app component

○ Normal Method ○○ Component Lifecycle Callback ○ Permission Check or Request

○ Dangerous API → Method Call Relation

Possible Practices: Asynchronous Management


(d) Inter-component

The inter-component case:

The permission check/request methods and the dangerous APIs are invoked in different callbacks of different app components

○ Normal Method   ○○ Component Lifecycle Callback   ○ Permission Check or Request

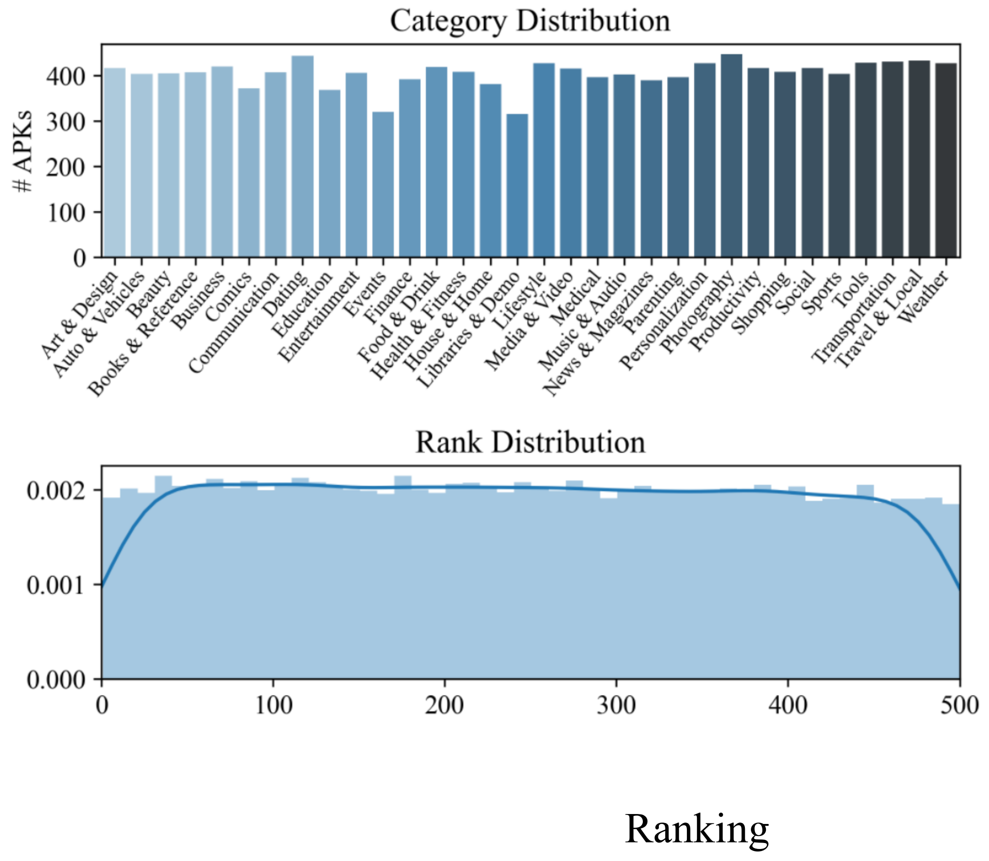
○ Dangerous API   ⟶ Method Call Relation   ⇢ Inter-component Communication

How Common Are The Practices?

- Dataset: Google Play

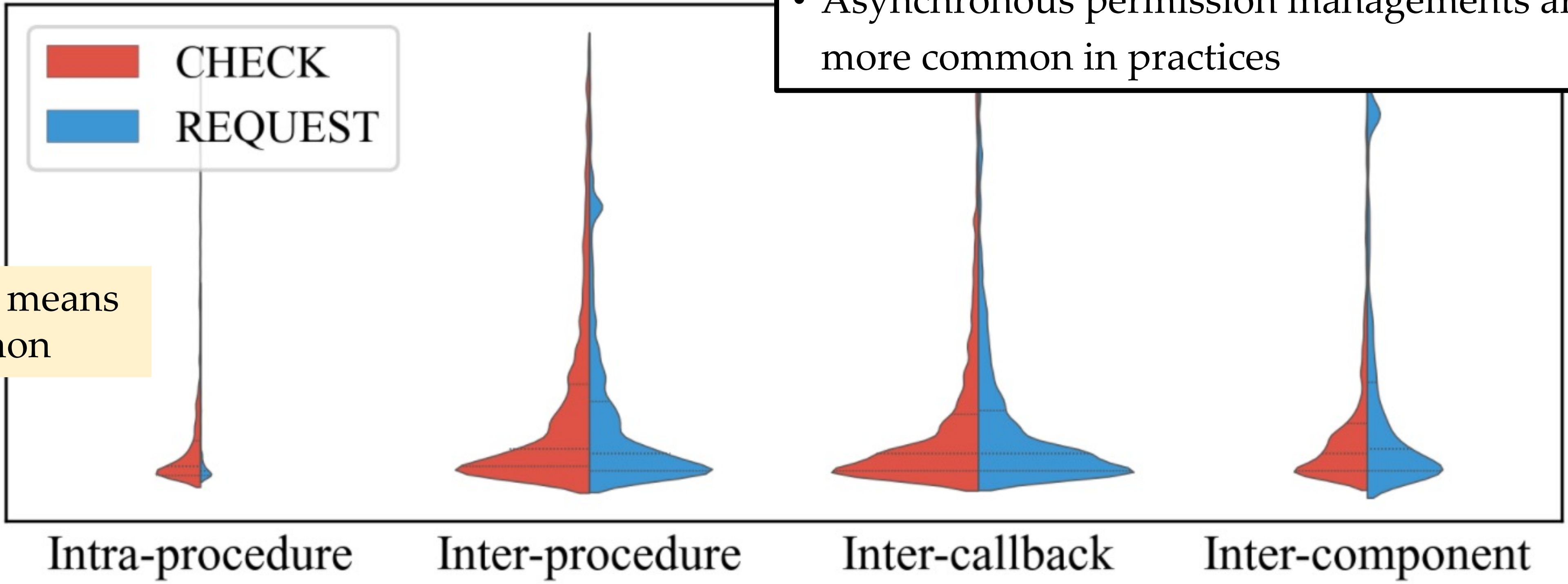
- Top-500 apps in 32 categories
- Using at least one dangerous API and permission check/request API

# apps: 13,352



How Common Are The Practices?

- Intra-procedural management is not common
- Asynchronous permission managements are more common in practices



Larger area means more common

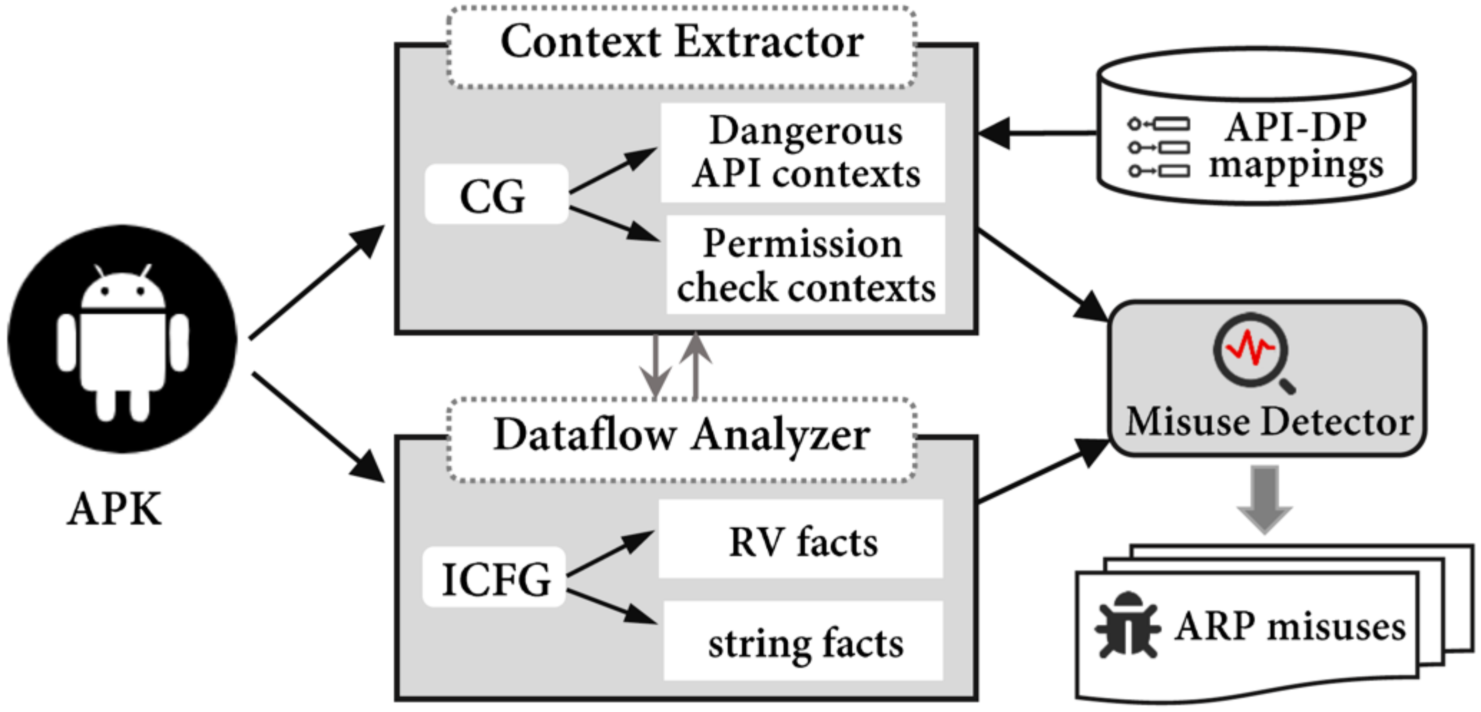
Technical Challenges

- **Challenge 2:** Evolving API-permission mappings and mechanism
  - API-permission mappings have been constantly changing over time
    1. Dangerous API additions and removals
    2. Same API corresponds to different permissions
  - Changes of permission mechanisms:
    1. Permission groups often change
    2. The introduction of one-time permission (Android 11)

Need to analyze app behaviors on different Android versions!

Calciati, P., Kuznetsov, K., Gorla, A., & Zeller, A. (2020, June). Automatically granted permissions in Android apps: An empirical study on their prevalence and on the potential threats for privacy. In Proceedings of the 17th International Conference on Mining Software Repositories (pp. 114-124).

The Aper Tool



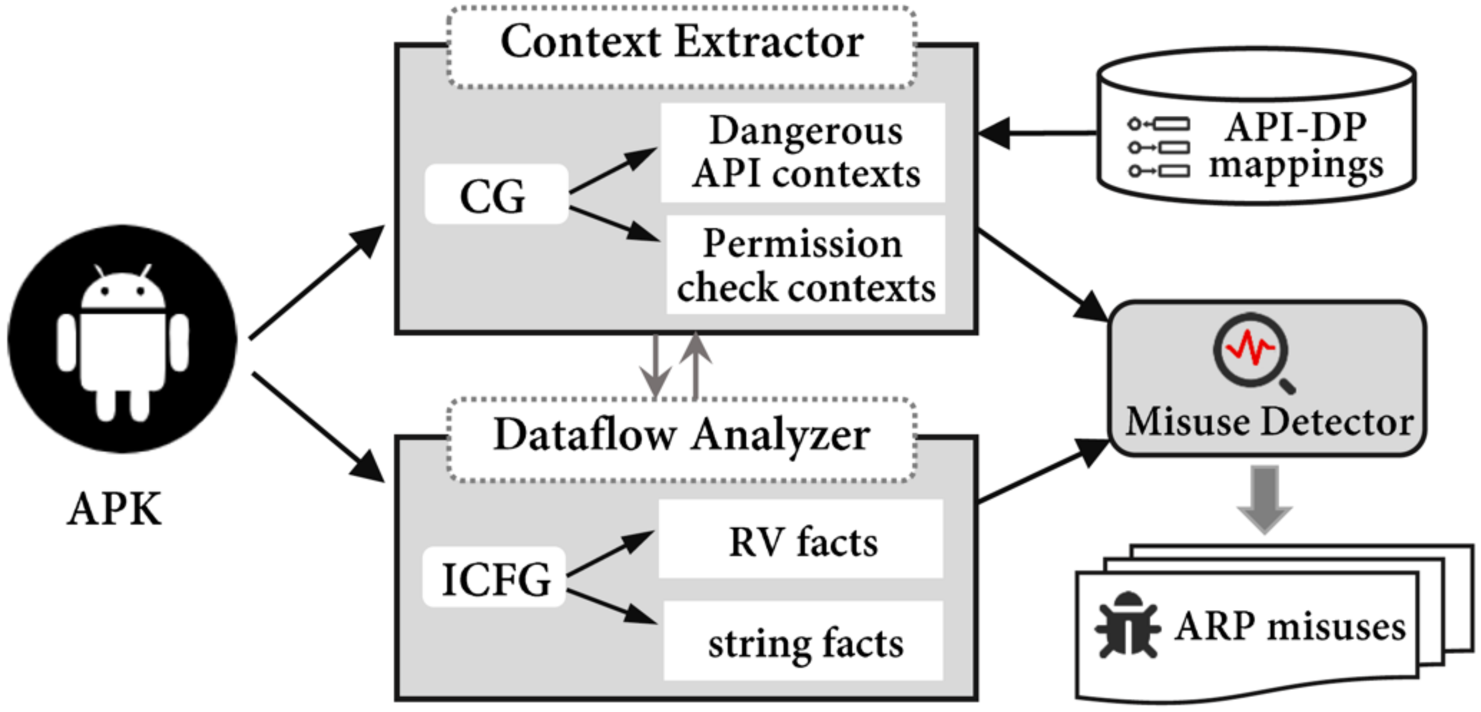
Input:

- An app's APK file
- The mappings between APIs and permissions (extracted from framework code)

Output:

- Type-1 & Type-2 bugs
- Calling contexts of dangerous APIs (for debugging)

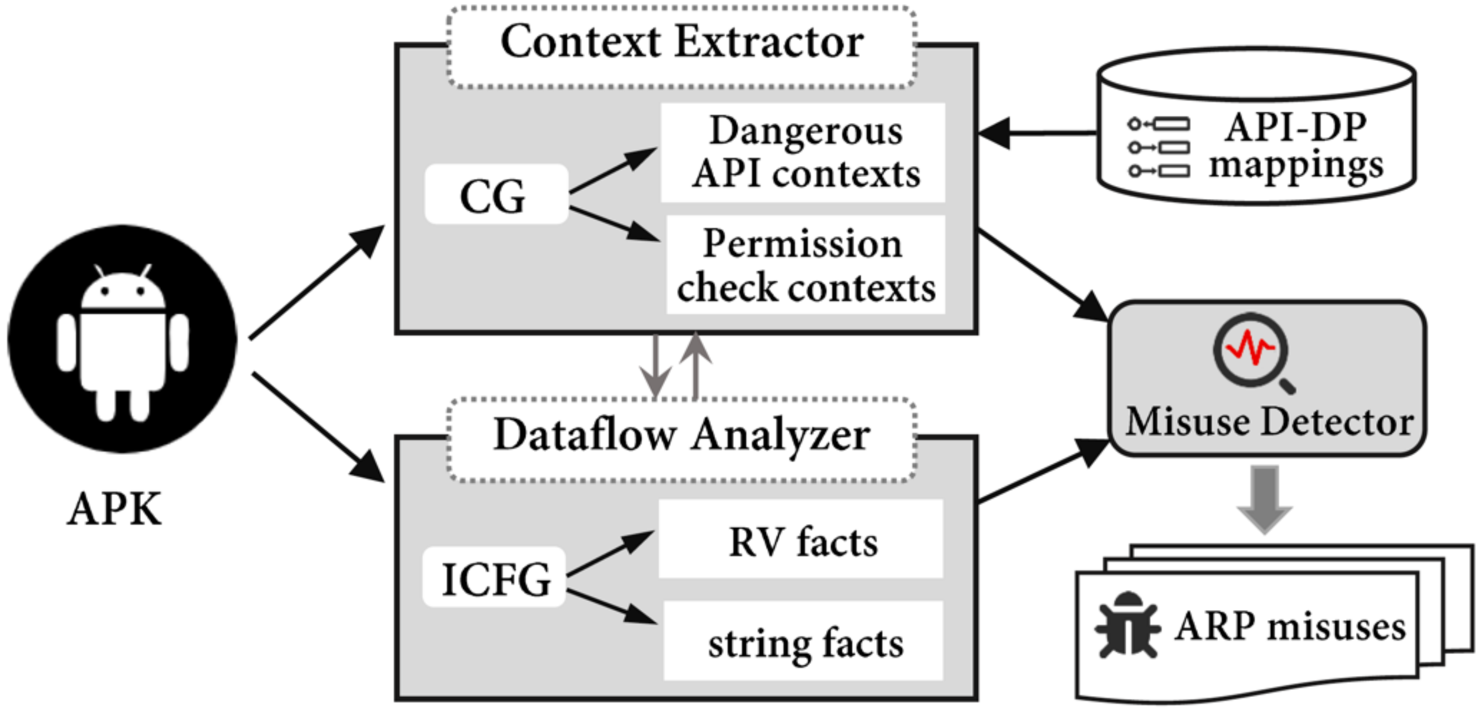
The Aper Tool



Context Extractor:

- ➢ Traverse the call graph to find the call sites of (also extract calling contexts):
  - ➢ Dangerous APIs
  - ➢ Permission check APIs

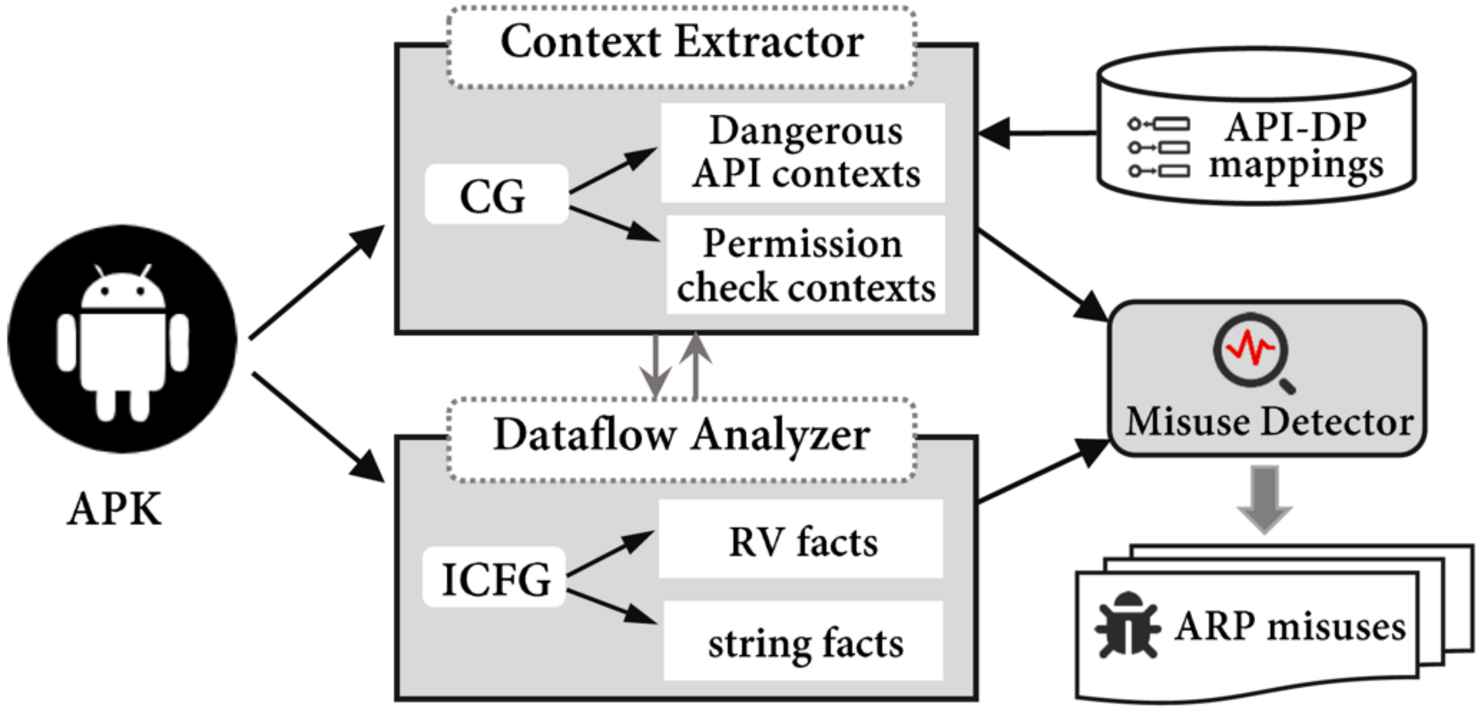
The Aper Tool



Dataflow Analyzer:

- Perform reaching-definition analysis on inter-procedural control-flow graph to infer two sets of facts:
  - Possible string values passed to the check APIs
  - Reachable runtime versions on which dangerous APIs can be called

The Aper Tool

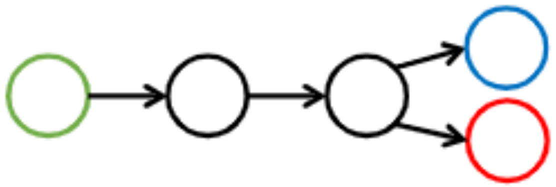


Misuse detector:

- ➢ Leverage the API calling contexts and dataflow facts to detect Type-1 & Type-2 bugs

The Basic Idea

- Permission management should happen before dangerous API calls (domination requirement)

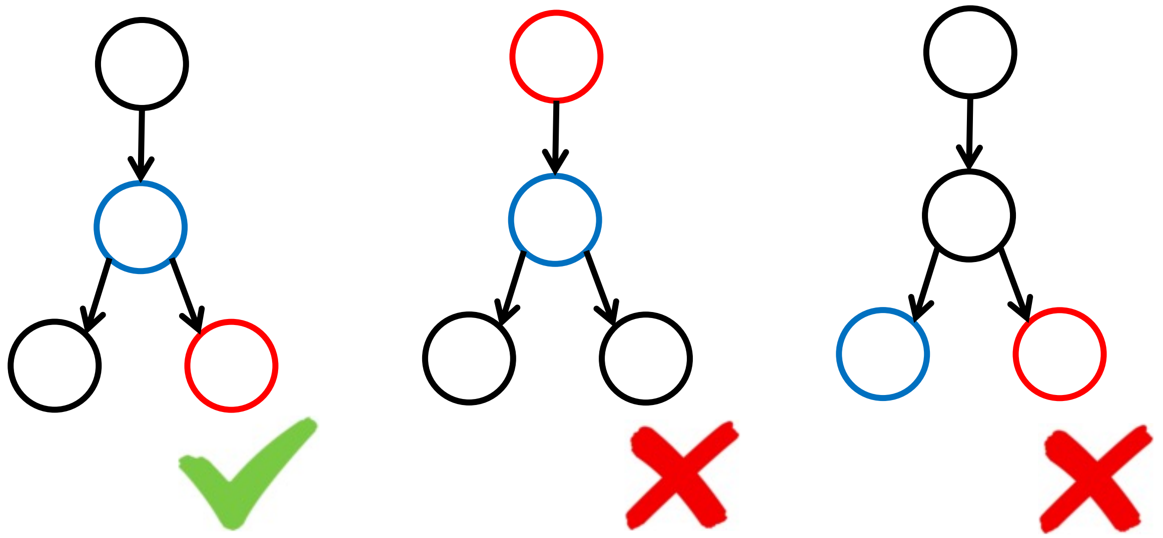

(a) Intra-procedure

○ SHOULD happen-before ○

Domination relation between API call sites on control-flow graph.

○ Permission management (check & request) ○ Dangerous API

The Basic Idea

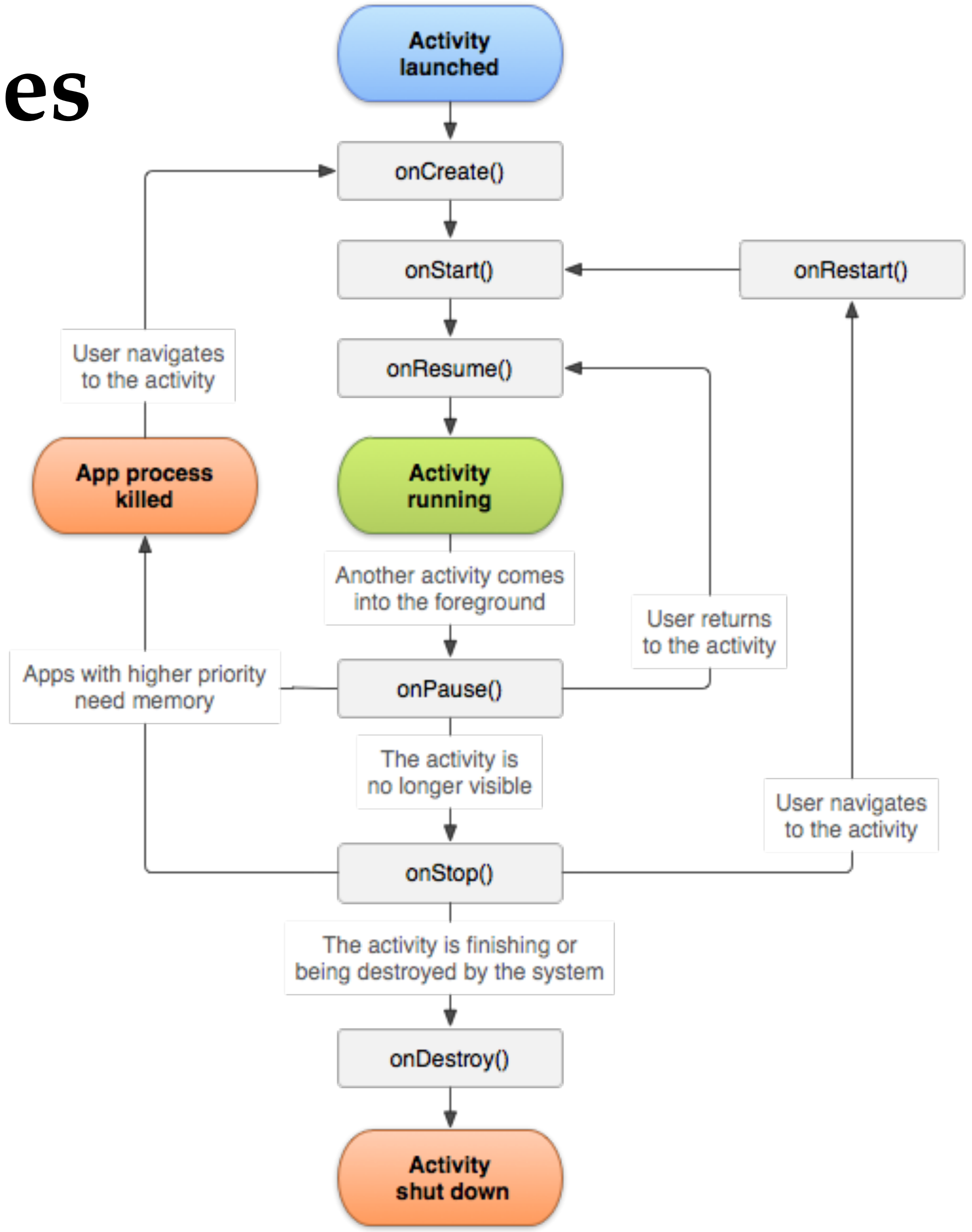


Bugs may occur if we find violations of the domination requirement on the control-flow graph

○ Permission check site ○ Dangerous API call site → Control flow

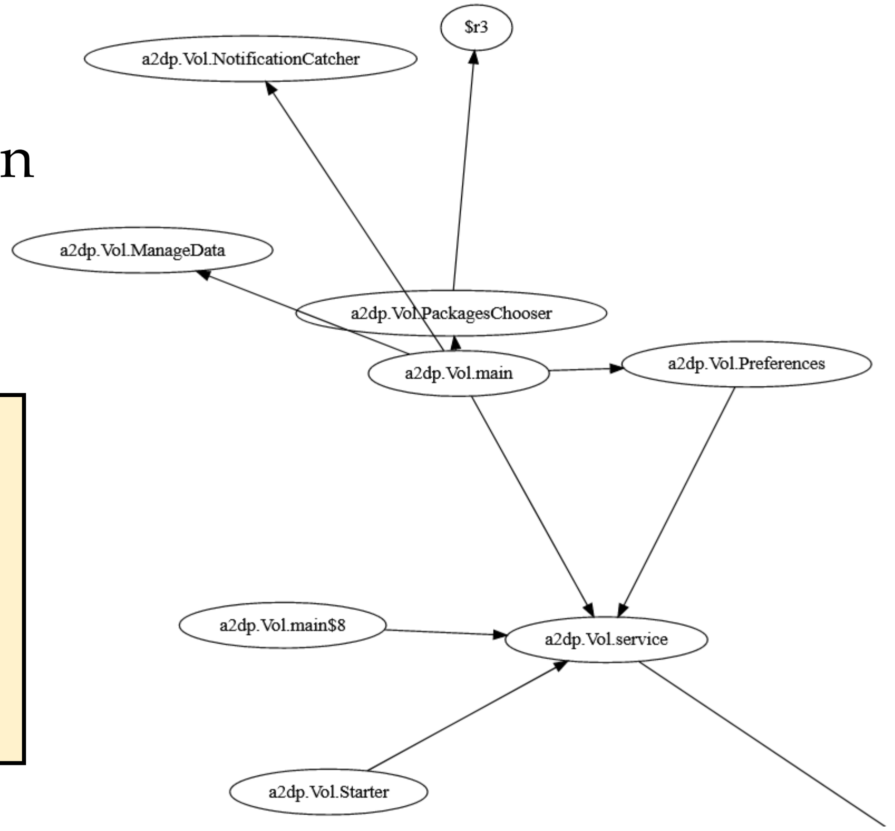
Handling Asynchronous Cases

1. We consider the temporal order of callbacks by referring to the lifecycle model of app components


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Handling Asynchronous Cases

2. We also built the inter-component communication graph following Li et al.'s work

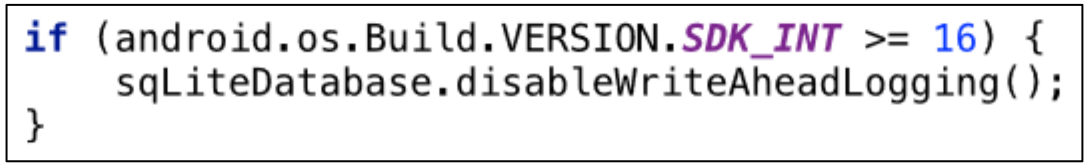


The two steps allow us to know the execution order among callback methods, such that we can find the violations of the domination requirement even if the developers manage permissions asynchronously.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., ... & McDaniel, P. (2015, May). Iccta: Detecting inter-component privacy leaks in android apps. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 280-291). IEEE.

Detecting Compatibility Issues

- We extend the previous domination requirement by further requiring that the runtime version checks **should dominate** the call to the evolving dangerous APIs



```
if (android.os.Build.VERSION.SDK_INT >= 16) {
    sqLiteDatabase.disableWriteAheadLogging();
}
```

Wei, L., Liu, Y., Cheung, S. C., Huang, H., Lu, X., & Liu, X. (2018). Understanding and detecting fragmentation-induced compatibility issues for android apps. IEEE Transactions on Software Engineering, 46(11), 1176-1199.

Evaluation

- **RQ3:** How effective is Aper in detecting runtime permission issues, compared with the existing tools?

- **RQ4:** Can Aper detect unknown issues in real-world apps?

The Baseline Tools

1. **ARPDroid**[1]: For migrating pre-6.0 apps to new platforms

2. **RevDroid**[2]: Find issues related to permission revocation

3. **Lint**[3]: A rule-based checker in Android Studio

[1] Dilhara Malinda, Haipeng Cai, and John Jenkins. "Automated detection and repair of incompatible uses of runtime permissions in android apps." *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. 2018.

[2] Fang, Zheran, et al. "revDroid: Code analysis of the side effects after dynamic permission revocation of android apps." *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 2016.

[3] Lint. https://developer.android.com/studio/write/lint

The ARPfix Benchmark

- We built the benchmark from open-source Android projects:
  - ARP-related commits/PRs
  - Manually remove irrelevant code
  - Two versions for each issue: buggy/patched
  - Size:
    - Type-1: 35 × 2 APKs
    - Type-2: 25 × 2 APKs

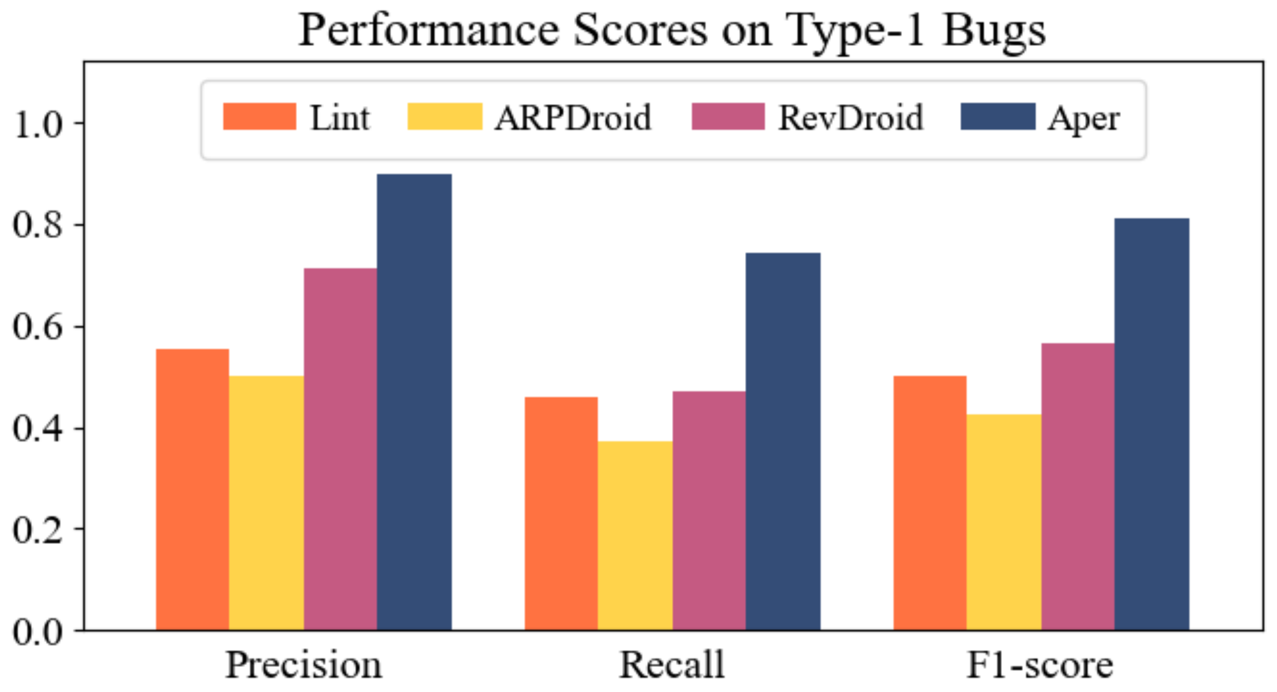
- Available at: https://github.com/sqlab-sustech/APER-ARPfix-benchmark

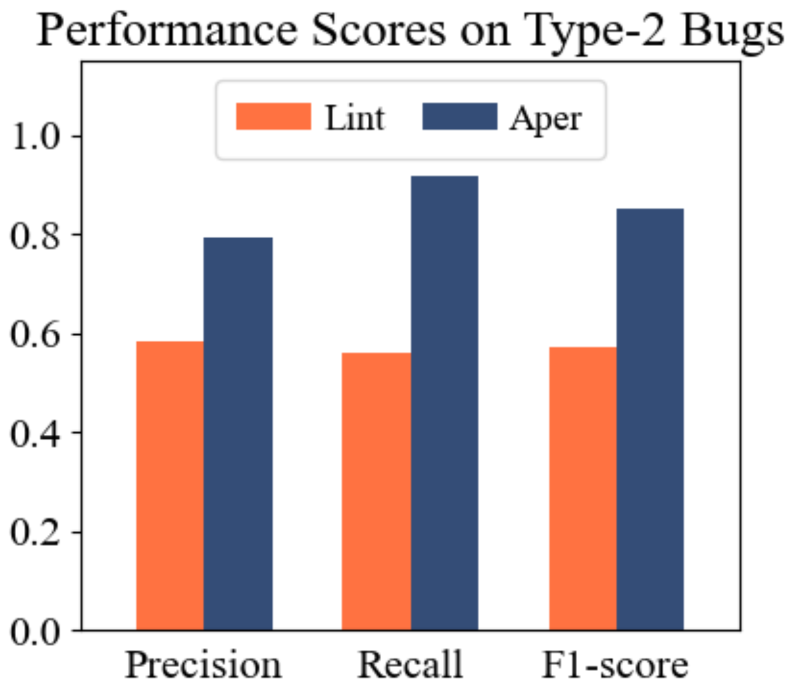
The Metrics

- **True positives (TP):** # buggy versions on which a tool reports issues
- **True negatives (TN):** # patched versions on which a tool reports no issues
- **False positives (FP):** # patched versions on which a tool reports issues
- **False negatives (FN):** # buggy versions on which a tool reports no issues

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \qquad F_1 - score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

Tool Performance





Common Limitations of Existing Tools

- No explicit modeling of asynchronous permission management
  - This causes many false positives
- Does not take the API and permission evolution into consideration
  - Cannot effectively find Type-2 issues

Detecting Unknown Issues

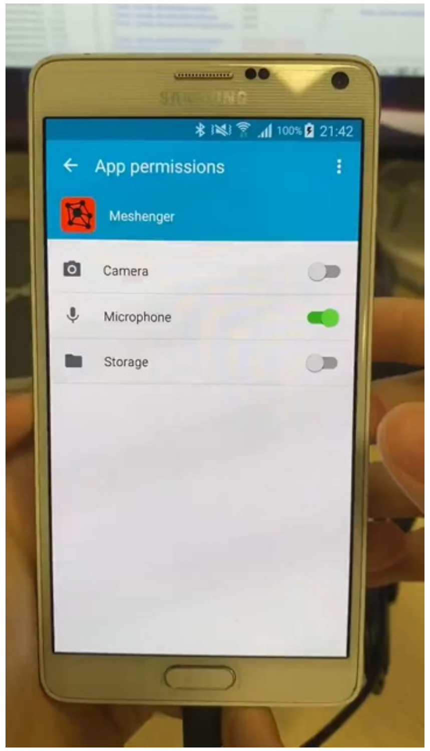
F-Droid

- We used 214 open-source apps indexed by F-Droid for the experiment
  - Should use dangerous permissions
  - Should be actively maintained

	Detected	Verified	Recorded Video
Type-1	66	23	21
Type-2	18	11	3
Total	84	34	24

(unverified ≠ false alarm)

All videos available at: https://aper-project.github.io/



Conclusion

- Runtime permission issues are common in Android apps
- We found 11 types of runtime permission issues from open-source projects
- We built a static analyzer, Aper, to detect two prominent types of issues
- Future work:
  - Study runtime permission issues caused by the interference of third-party libraries
  - Improve the Aper tool (support more issue types, reduce false positives)

Thanks for Listening!

Aper is open-sourced at: https://github.com/sqlab-sustech/APER-tool/

Contact us: liuyp1@sustech.edu.cn