On Demystifying the Android Application Framework:
Re-Visiting Android Permission Specification Analysis

Michael Backes, Sven Bugiel, **Erik Derr**, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber

USENIX Security Symposium
Austin, August 12th, 2016

CISPA
Center for IT-Security, Privacy and Accountability

Motivation

- Application framework internals still largely a black box
  - How do internals influence platform security and user-privacy
- Every security analysis requires a solid foundation
  - How to analyze the target in the first place?
  - Any platform-specific peculiarities
    that impede a static analysis?

C|ISPA Center for IT-Security, Privacy and Accountability

Motivation

- Lot of work established such knowledge for apps
  - Entry points (Chex, FlowDroid)
  - Generation of a static runtime model (FlowDroid, R-Droid, Epicc)
  - Sources/sinks (SuSi)
- Yet, such a knowledge base is missing for the application framework
  - System services provide core functionality
  - Existing knowledge from app analysis can **not** be transferred

Contributions

- Systematic methodology on how to statically analyze the application framework
  - How to enumerate framework entry points
  - How to generate a precise static runtime model
- Re-Visiting permission specification analysis
  - More precise permission mappings for SDK / framework
- Study internals of Android‘s permission system
  - How to classify sensitive operations guarded by permission checks
  - Where are permissions checked?

How to analyze the framework

Analysis Ingredients

How to enumerate framework entry points?

#1

How to generate a static model that approximates runtime behavior?

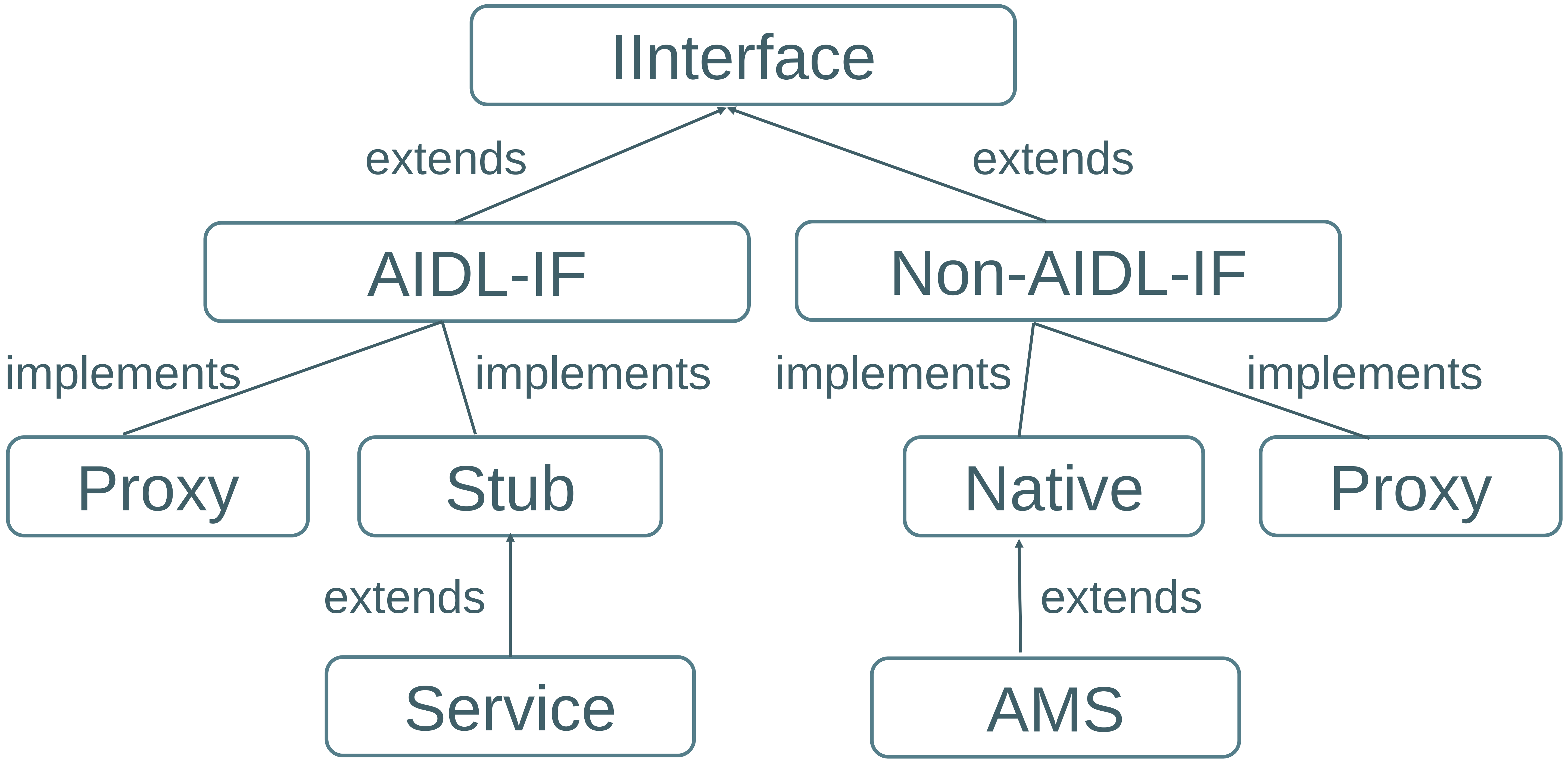
#2

What are the sensitive sinks within the framework?

#3

Framework Entry Points (#1)

- What functionality is exposed to app layer?
  - Key observation: Functionality only exposed via Binder-IPC
  - Entry class enumeration via class hierarchy analysis



Static Runtime Model (#2)

- Framework services follow the principle of separation of duty
- Highly responsive to process simultaneous queries from multiple clients (apps)
- Various concurrency pattern that complicate static analysis
  - Handler
  - AsyncChannel (framework only)
  - StateMachines (framework only)

Static Runtime Model - Handler

- Many services have a dedicated handler to process messages in a separate thread

```
public void enable() {
  Message msg = mHandler.obtainMessage(MESSAGE_ENABLE)
  mHandler.sendMessage(msg);
}
```

Runtime type

Message code

```
class BluetoothHandler extends Handler {
  public void handleMessage(Message msg) {
    switch (msg.what) {
      case MESSAGE_ENABLE:   // do_enable
      case MESSAGE_DISABLE:  // do_disable
      // other cases
}}}
```

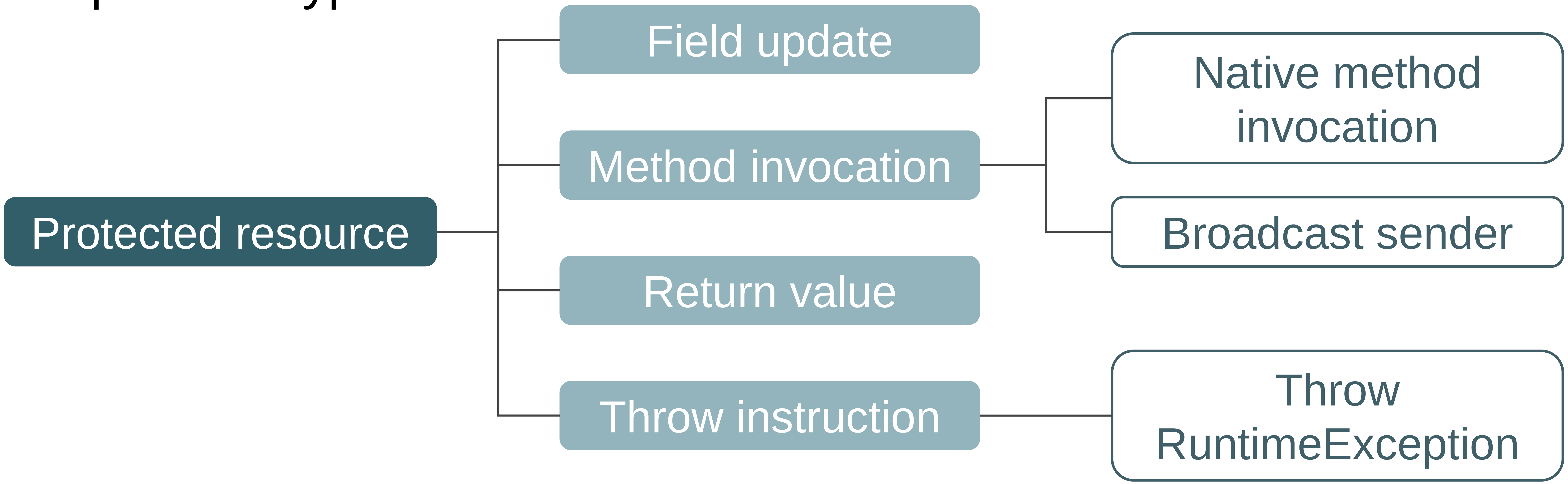
Path sensitivity

Protected Resources (#3)

- Concept of sources/sinks a list of APIs is no longer applicable
  - Analysis now shifts into the framework API
- How do we classify sensitive functionality?
  - Consider permission checks as guards of sensitive operations
- **Protected resources** are security-sensitive operations that have a tangible side-effect on
  - the system state or
  - use of privacy

Taxonomy of Protected Resource Types

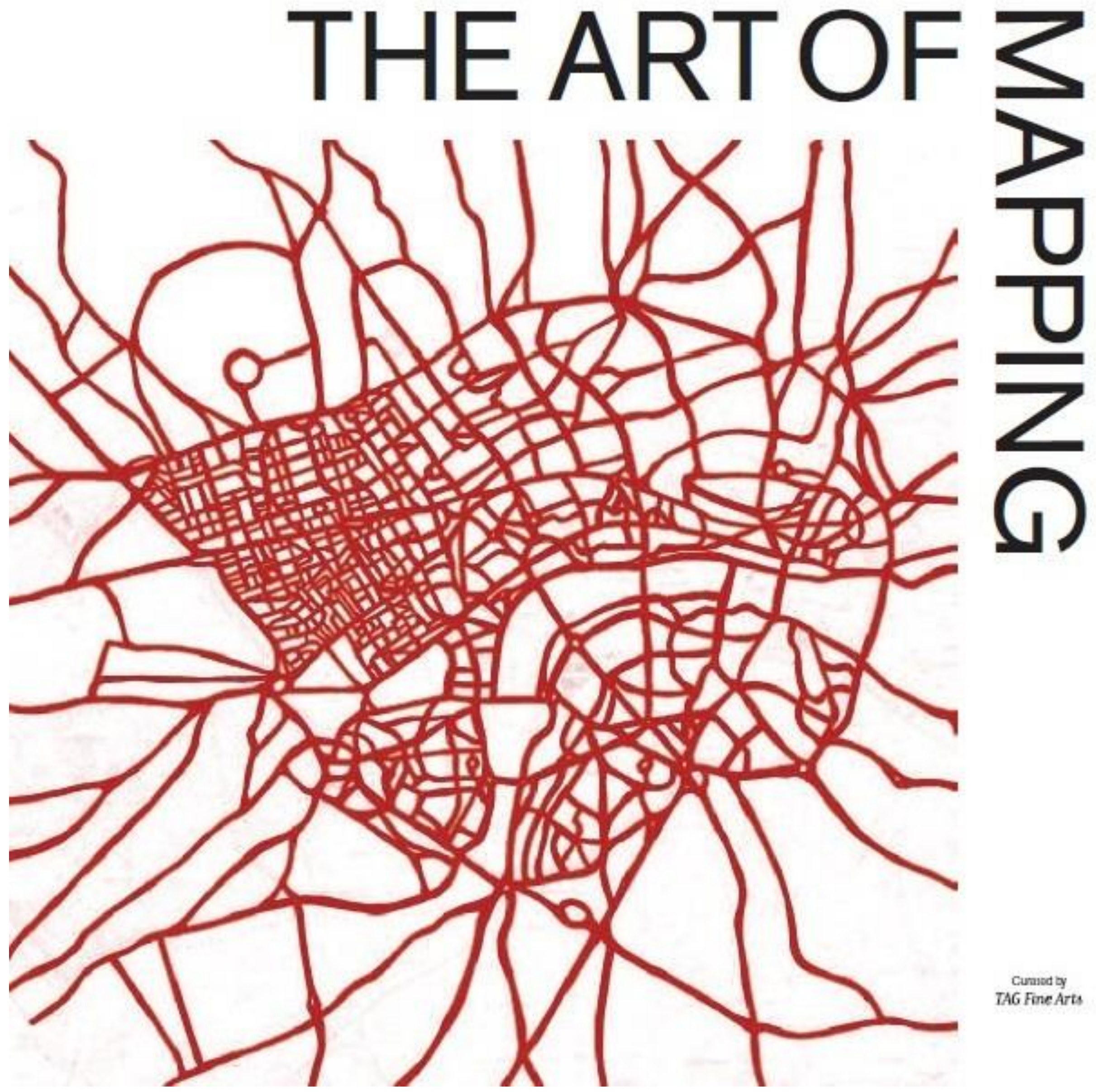
- No ground truth so far, thus we manually investigated 35 entry points from different services
- Diversity of operations forced us to create higher-level classification on operation types



Use-Case: Permission Analysis

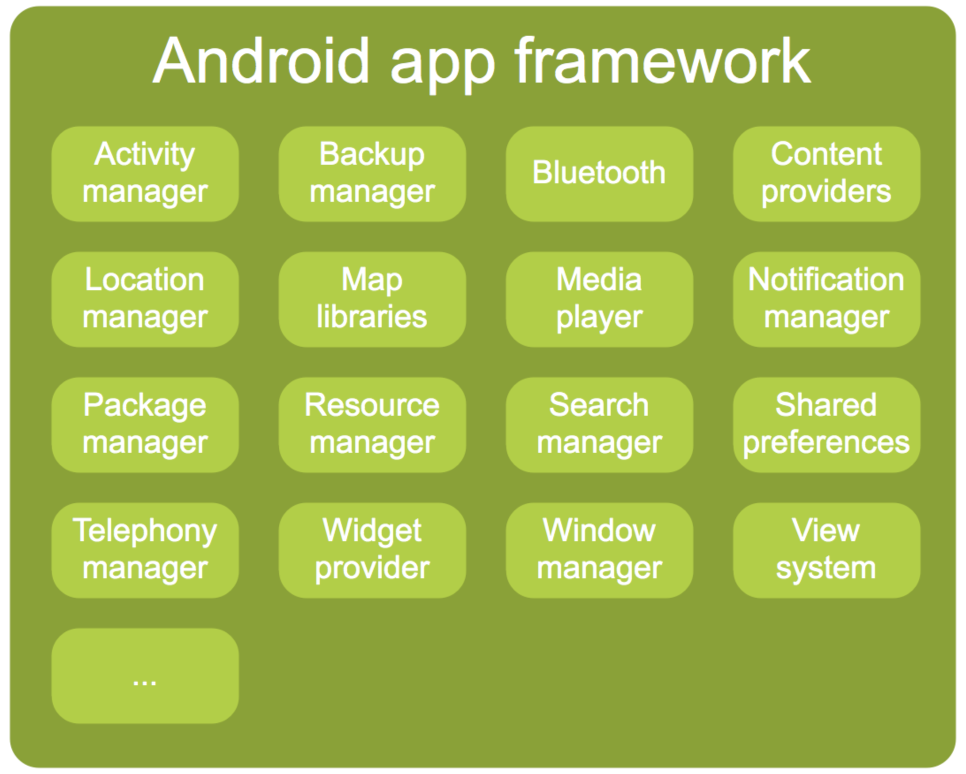
More Effort = Better Results?

- Generating precise graphs requires a lot of resources
- Do we perform better than existing work?
- Re-visit Android permission mappings!
  - Why? Still, one of the major security mechansim
  - Important for app developers & security research
  - Compare with state-of-the-art tool **PScout** (API 16)



Android Permission Mappings - Framework

- Map framework entry points to required permissions
- Approach: Forward control-flow slicing
- String analysis to resolve permission strings



Data-flow →

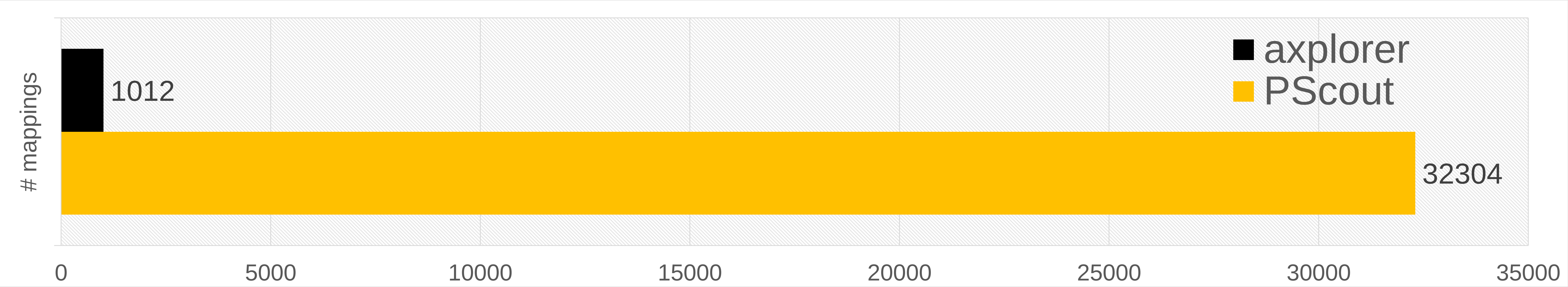
Permission checks

Framework / undocumented map

Framework entry point	→	**List of required permissions**
com.android.phone.PhoneInterfaceManager.getDeviceId()	→	android.permission.READ_PHONE_STATE

Framework API Mapping

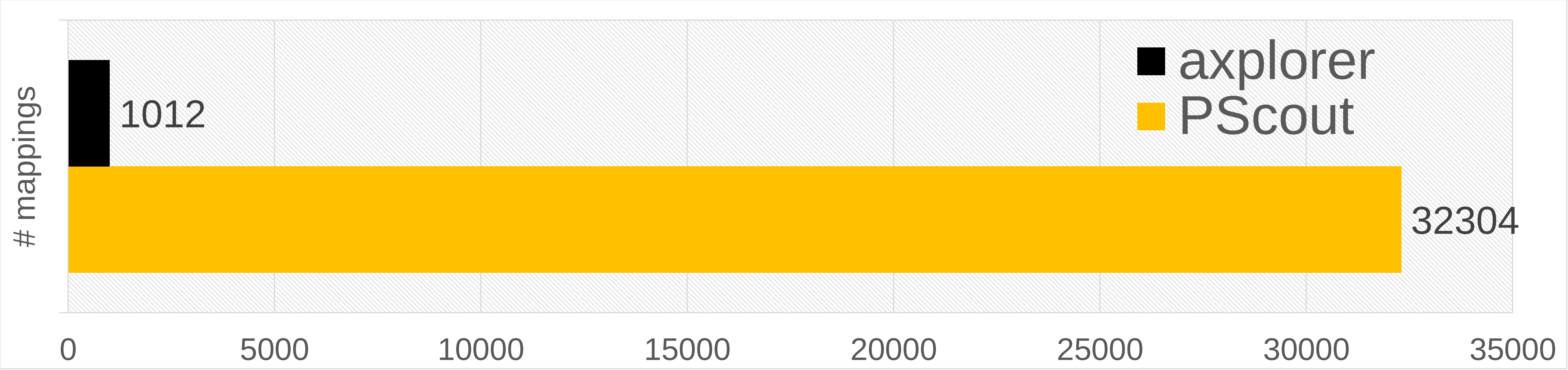
of API to permission mappings



- PScout includes normal + dangerous permissions
- axplorer additionally includes system + systemOrSignatures permissions

Framework API Mapping

- Less false mappings
- Reduced over-approximation through more precise call-graphs
- Entrypoint definition ensures valid mappings

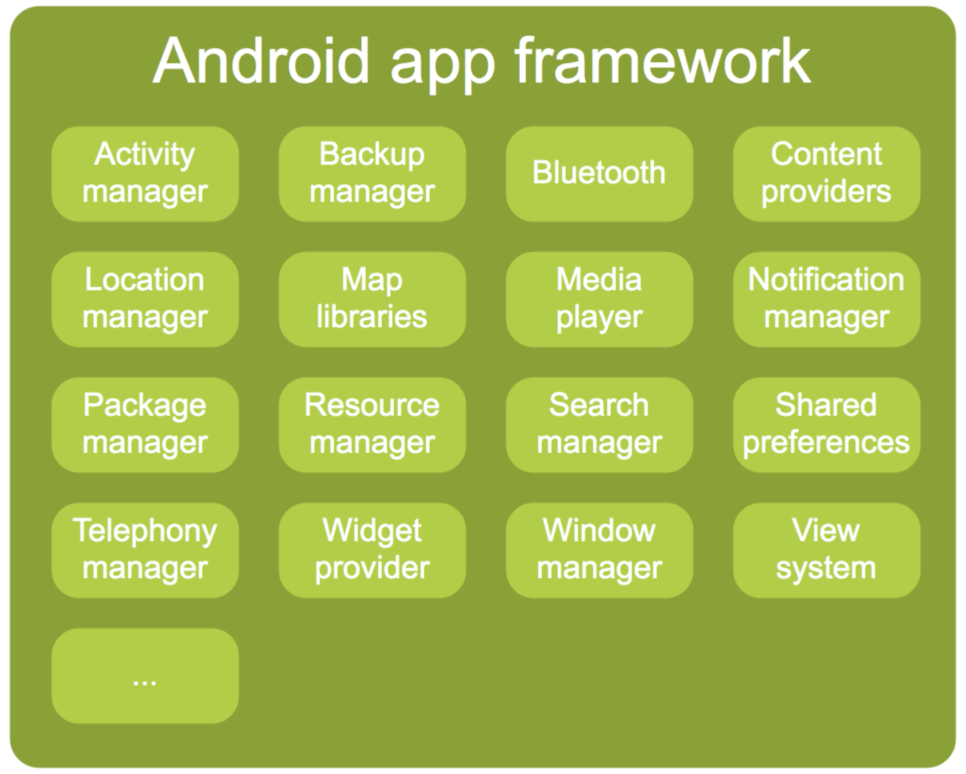


Android Permission Mappings - SDK

CISPA
Center for IT-Security, Privacy and Accountability

SDK

IPC



Data-flow

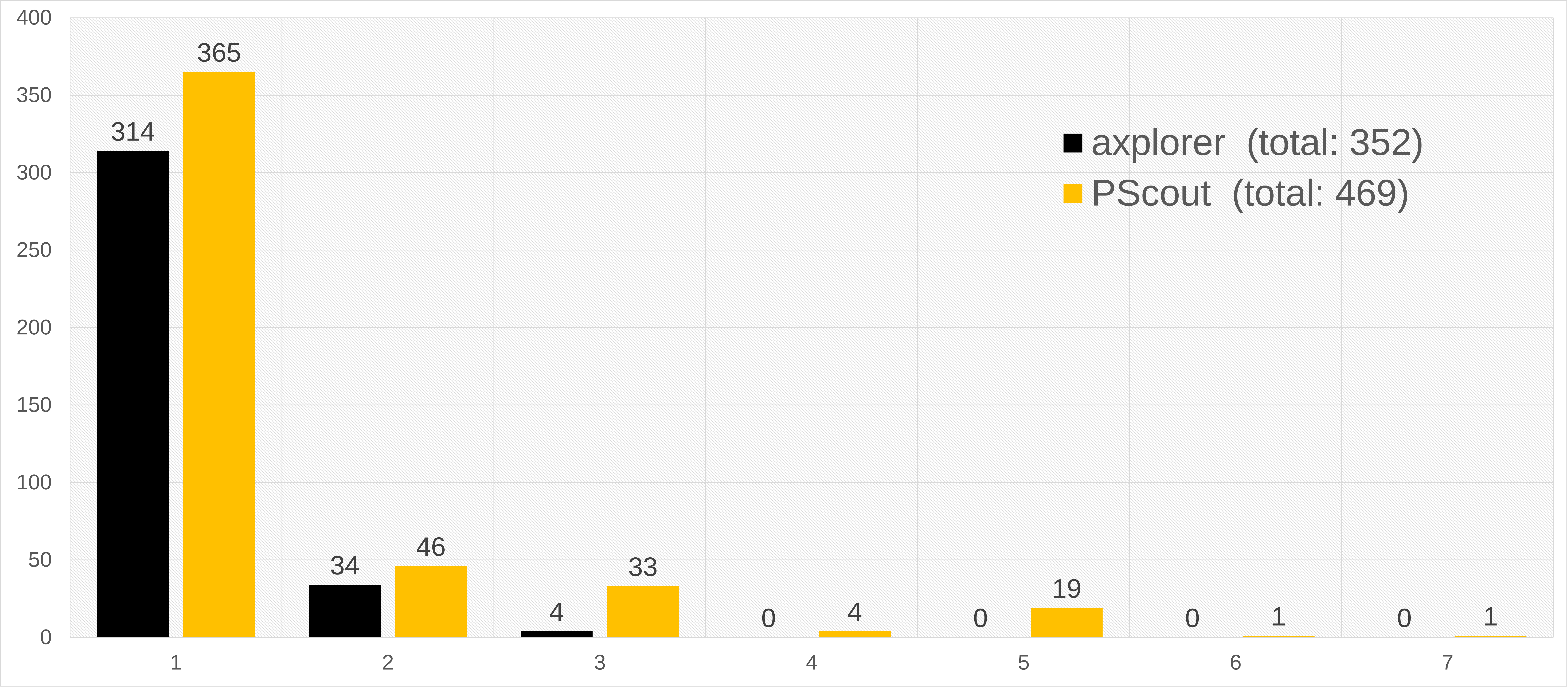
Permission checks

Framework / undocumented map

SDK / documented map

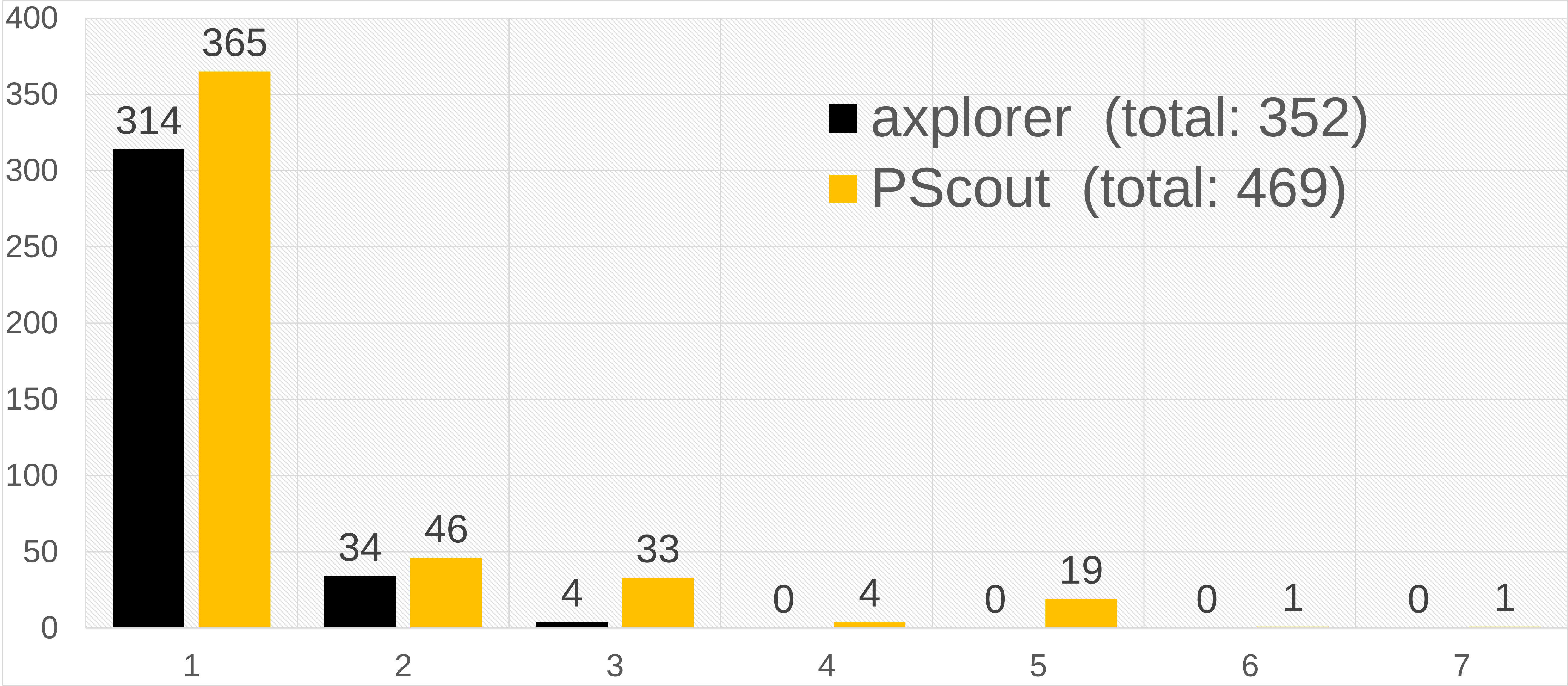
SDK Mapping (1)

Number of permissions required by documented APIs



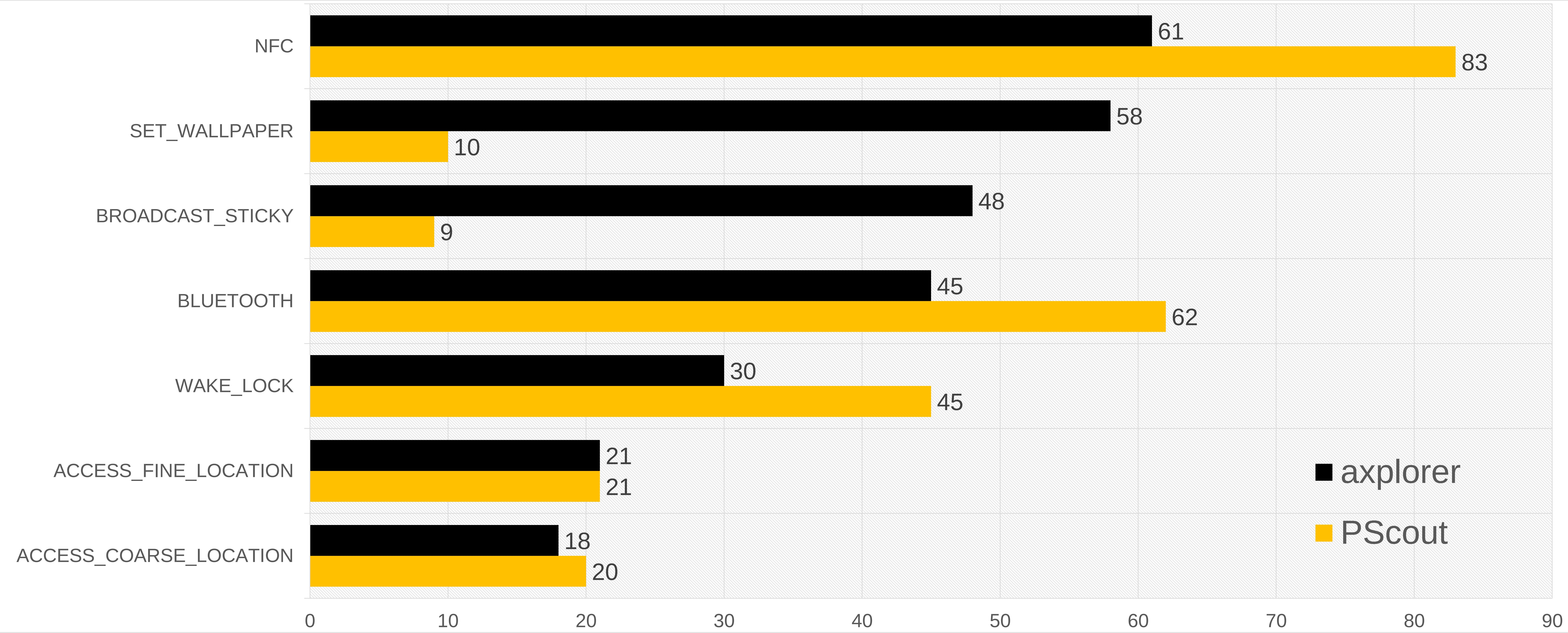
SDK Mapping (1)

- Connecting SDK to framework eliminates false-mappings
- Mappings with non-entry methods are ruled out
- Path-sensitivity in Handler eliminates outliers



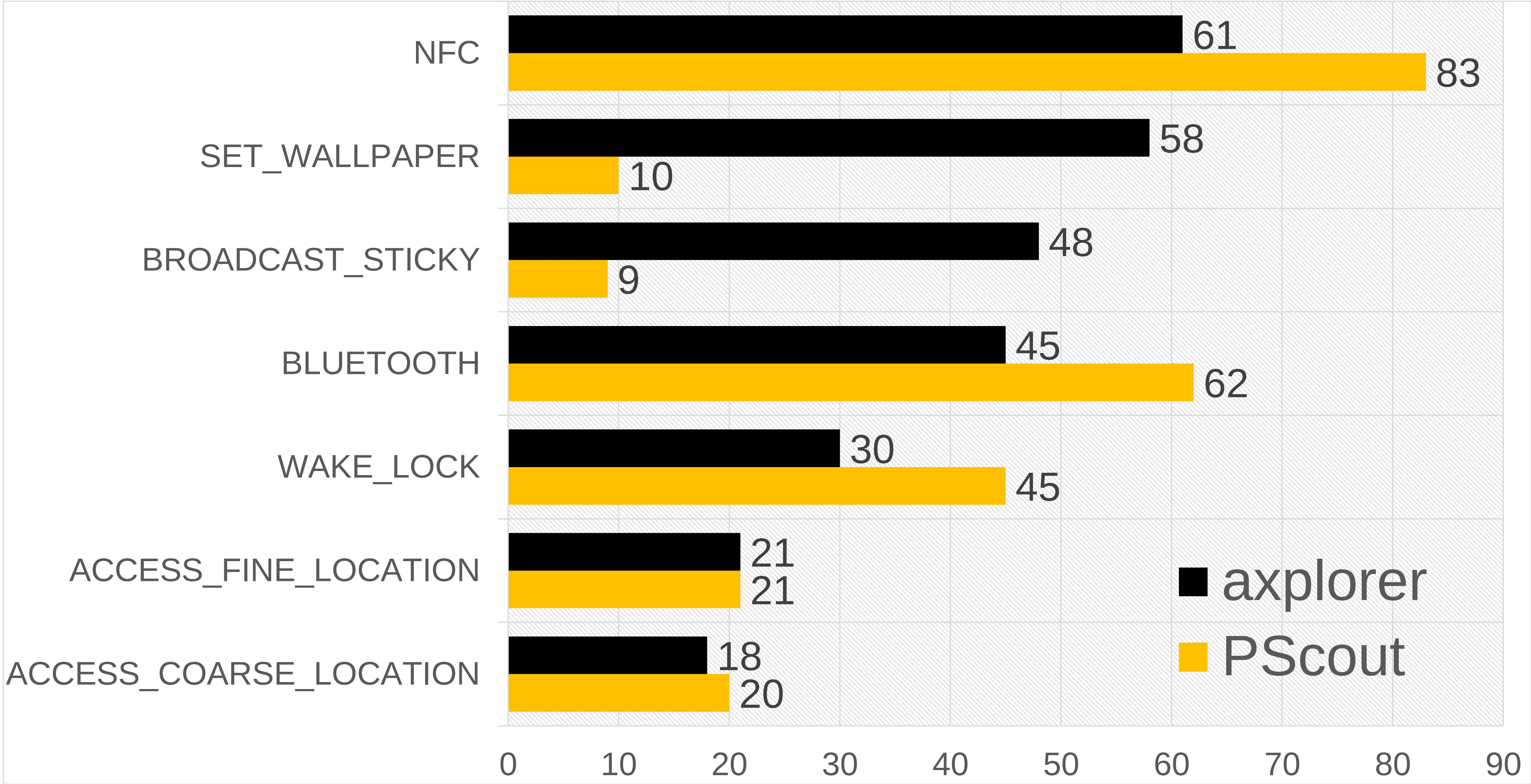
SDK Mapping (2)

Number of documented APIs that require a specific permission



SDK Mapping (2)

- Manually validated the top 4 permissions
- Differences due to SDK analysis
- Context class difficult to get right (>100 direct and indirect subclasses)



Permission Locality

- Services follow the principle of separation of duty
  - How are permission checks distributed?
- Across API versions ~20% of permissions are checked in >1 class and at most in 10 classes
  - This equally affects all protection levels (dangerous, system,..)
- There is a trend towards more checks in more classes in newer Android versions

Permission Locality

- Locality measured in terms of number of distinct classes that check a given permission
- **High** permission locality
  Permission is checked/enforced at a single service
  - SET_WALLPAPER is only enforced at WallpaperManagerService
- **Low** permission locality
  Permission is enforced at different (possibly unrelated) services

CISPA
Center for IT-Security, Privacy and Accountability

Low Permission Locality

- Framework API 16 (4.1.1)
  - Permission: **READ_PHONE_STATE**
  - Level: **dangerous**

System app

`internal.telephony.PhoneSubInfo`

`phone.PhoneInterfaceManager`

`server.TelephonyRegistry`

`server.net.NetworkPolicyManagerService`

Low Permission Locality

- Framework API 22 (5.1)
  - Permission: **READ_PHONE_STATE**
  - Level: **dangerous**

`internal.telephony.PhoneSubInfoProxy`

`internal.telephony.SubscriptionController`

`phone.PhoneInterfaceManager`

`server.TelephonyRegistry`

`server.net.NetworkPolicyManagerService`

Permission Locality

- Locality steadily decreases between new Android versions
- Impedes understanding the big picture of Android permissions
- Single enforcement point for permissions?
  - Facilitates policy generation for access control frameworks (ASM/ASF)
- How to establish?
  - Identify owning class/service for each permission
  - Dedicated permission check method that is exposed via IInterface

Conclusion

- Comprehensive and systematic methodology on how to analyze Android‘s application framework
- First high-level classification of protected resource types
- Re-Visited permission analysis
  - Improved on prior results of SDK / framework mappings
  - Permission locality improves understanding of permission system
- Check out **www.axplorer.org**