

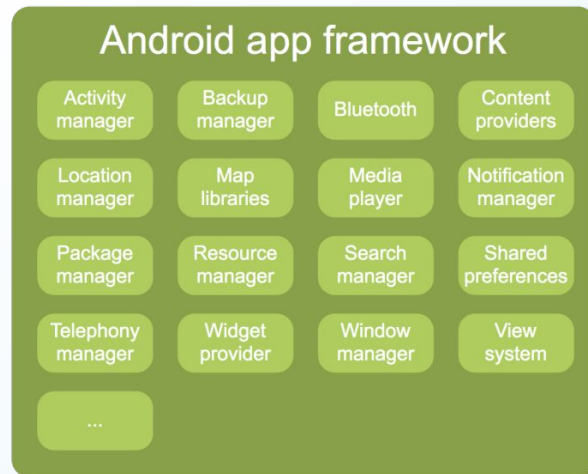
# Mobile Security: Framework Analysis



Yue Duan

# Outline

- Research paper:
  - EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework
  - Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework



# EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework

Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele,  
Christopher Kruegel, Giovanni Vigna, and Yan Chen

NDSS 2015

# Introduction

- Static analysis has been used for security and privacy
- Many analyses rely on the control flow graph
- Challenge:
  - ignoring the framework  $\Rightarrow$  incorrect control flow
    - common cause for imprecision: 'callbacks', e.g., onClick
  - analyzing the framework  $\Rightarrow$  8.6 million lines of code
    - no existing research

# Motivating Example

```
1 public class MainClass {  
2     static String url;  
3     public static void main(String[] args) {  
4         MalComparator mal = new MalComparator();  
5         MainClass.value = '42'  
6         Collections.sort(list, mal);  
7         sendToInternet(MainClass.value);  
8     }  
9 }  
10 public class MalComparator implements Comparable<Object> {  
11     public int compare(Object arg0, Object arg1) {  
12         MainClass.value = 'GPSCoords'  
13         return 0;  
14     }  
15 }
```

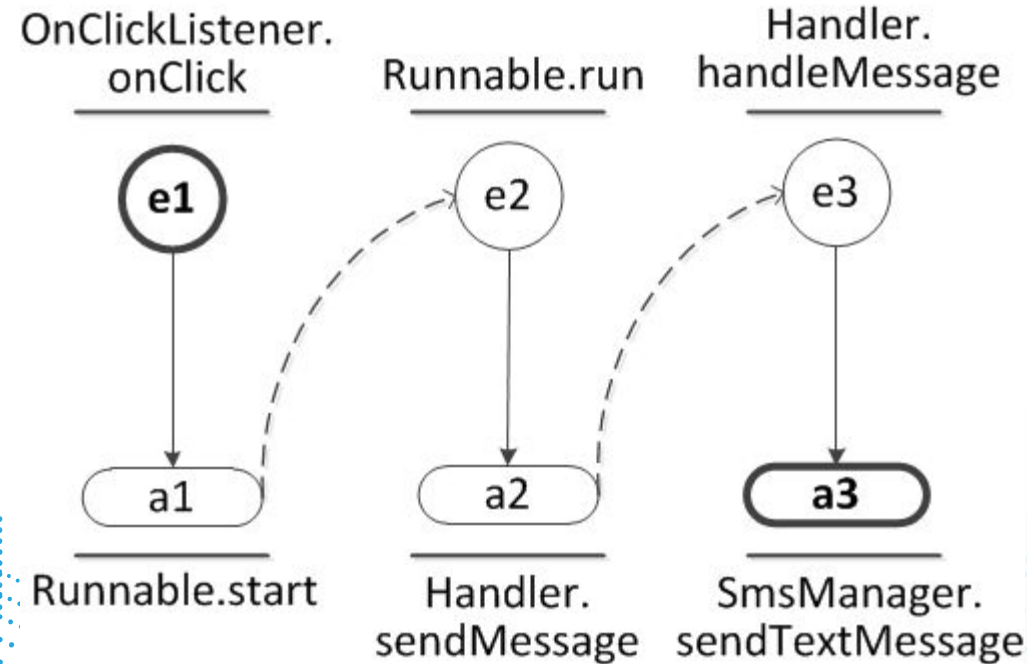
Privacy leakage is up to the value of MainClass.value.

The diagram illustrates a privacy leakage scenario. A grey arrow originates from the variable `MainClass.value` on line 5 of the `MainClass` and points to the `compare` method of `MalComparator` on line 12. Another grey arrow points from the `compare` method back to the `sendToInternet` method on line 7 of `MainClass`, which uses `MainClass.value`. This indicates that the value of `MainClass.value` is modified by the `compare` method and then used by `sendToInternet`, leading to a privacy leakage.

# Existing Approaches

- Whole program analysis
  - state explosion
  - redundant efforts (slow-down of static analysis)
- Summary-based analysis
  - extensive manual efforts
    - incomplete due to the high volume of callbacks
    - heuristic summarization: inaccurate

# Existing Approaches



```
//Ljava/util/LinkedList;.add
{
    signature = "<java.util.LinkedList: boolean add(java.lang.Object)>";

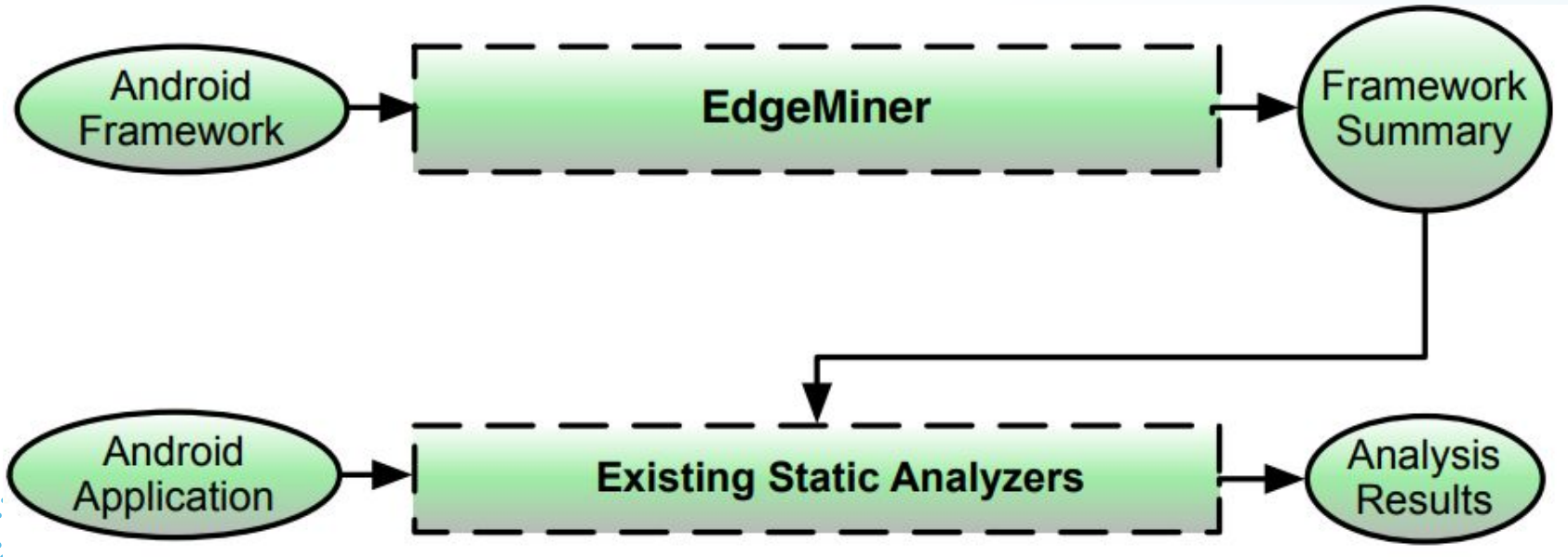
    stod = new LinkedHashMap<Integer, List<Integer>>();

    source = new Integer(0);
    dests = new ArrayList<Integer>();
    dests.add(new Integer(thisObject));
    stod.put(source, dests);

    summary.put(signature, stod);
}
```

# EdgeMiner

- Summarize framework: list of registration-callback pairs





# Concepts

- Callbacks
  - necessary condition: a framework method that can be overridden by an application method
- Registration
  - necessary condition: a framework method that is invokable from the application space

# A Data Flow

```
1 public class Collections {  
2     public static void sort(List list, Comparator comparator) {  
3         ...  
4         comparator.compare(e1, e2);  
5     }  
6 }
```

**Registration**

**Argument**

**Data Flow**

**Backward Data Flow Analysis**

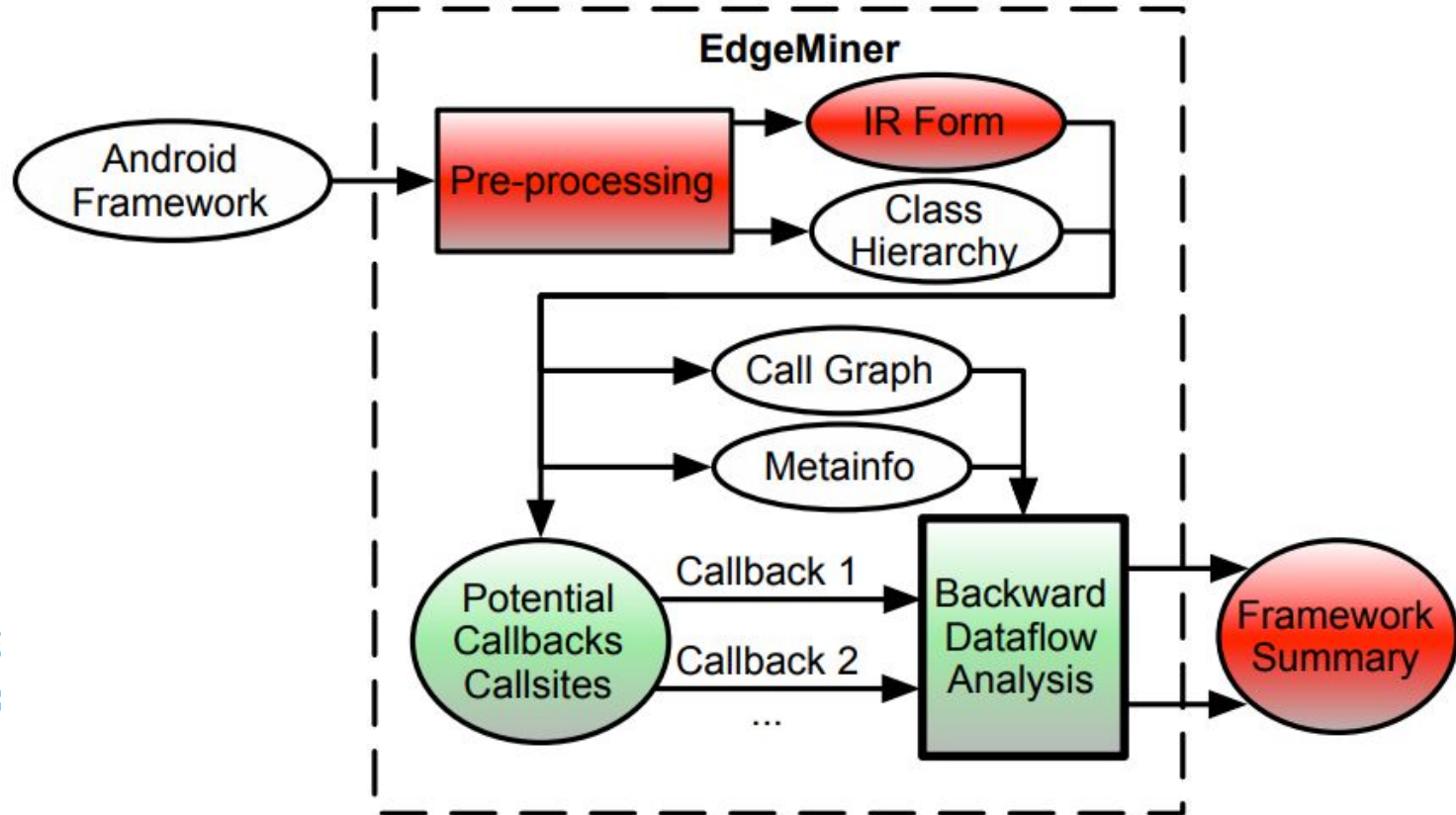
**Callback**

**An object with the callback**

# Implementation

- ROP intermediate representation (IR)
  - Well-suited for static analysis
  - SSA format
  - Integral part of Android SDK
- EdgeMiner
  - built on top of ROP
  - performs backward dataflow analysis
  - summarizes implicit control flows through framework

# System Architecture



# Implementation

- Preprocessing
  - transforms individual methods into SSA format
  - extracts class hierarchy
    - need for generating an over-approximation of the call graph
  - generates call graph
    - necessary for data-flow analysis
- Potential callback callsites

- *The method is public or protected.*
- *The class in which the method is declared has a public or protected modifier.*
- *The method is not final or static.*
- *The class in which the method is declared does not have the final modifier.*
- *The class in which the method is declared is an interface or has at least one explicitly or implicitly declared, public or protected constructor.*

# Evaluation

- Tested on Android 2.3, 3.0 and 4.2 frameworks
- Number of registrations and callbacks

Android Version	# Registrations	# Callbacks	# Pairs
<b>2.3 (API 10)</b>	10,998	11,044	1,926,543
<b>3.0 (API 11)</b>	12,019	13,391	2,606,763
<b>4.2 (API 17)</b>	21,388	19,647	5,125,472

# Accuracy

- False negative
  - compare with dynamic approach
    - incomplete but accurate
  - 8,195 randomly selected applications
  - 6,906 registration-callback pairs
  - EdgeMiner finds all pairs
- False positive
  - against manual inspection
  - 8 FP out of 200 pairs

# Improving Static Analyzer

- Integration with FlowDroid
  - synchronous callbacks: inline invocation
    - e.e., `collections.sort` and `Comparator.compare`
  - asynchronous callbacks: delayed invocation
    - e.g., `setOnClickListener` and `onClick`
- Patterns of callbacks used by FlowDroid and identified by

Pattern	# FlowDroid	# EdgeMiner
<b>*Listener*</b>	155	576
<b>*Callback*</b>	19	319
<b>*On*</b>	3	509
<b>None of the above</b>	4	18,243
<b>Total</b>	181	19,647



# Improving Static Analyzer

- Run 9 new apps in TaintDroid
  - 4 verified, 2 crash and 3 no leak
- Incorrect call graph → missed privacy leaks
- Performance
  - 34.7 seconds one-time loading
  - only 1.85% overhead added to Flowdroid

Tool	FlowDroid	FlowDroid + EdgeMiner
# Apps with $\geq 1$ privacy leak	285	294 (285 + 9)
# Privacy leaks (in total)	46,586	51,418
# Apps timeout	48	48

# Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework

Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, Z. Morley Mao

NDSS 2016

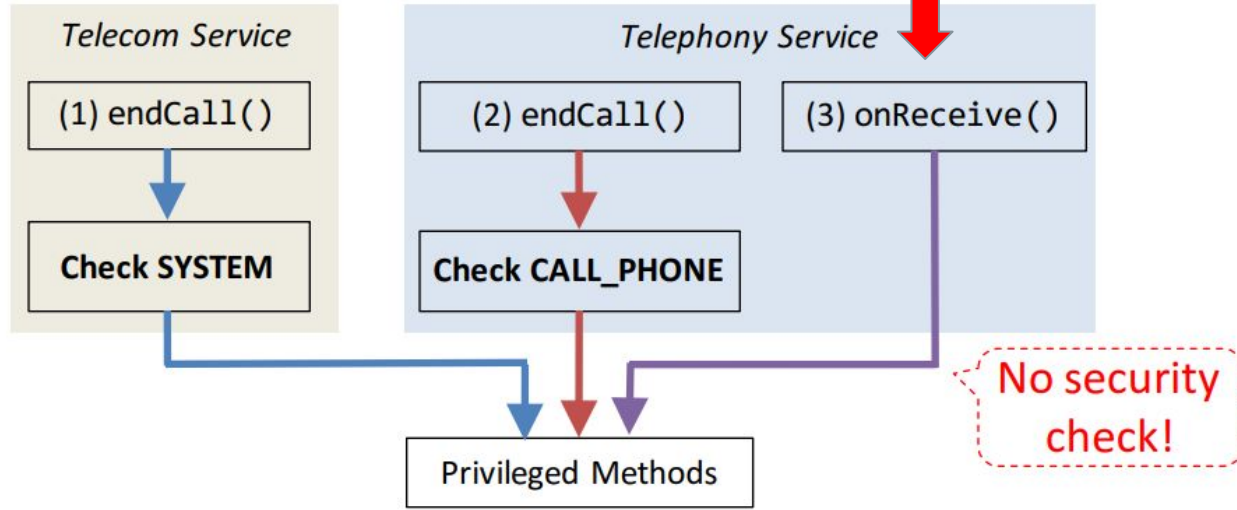
# Security Policy Enforcement

- Security policies regulate access to
  - sensitive data
  - system resources
  - privileged operations
- Policies need to be correctly enforced

# Motivation

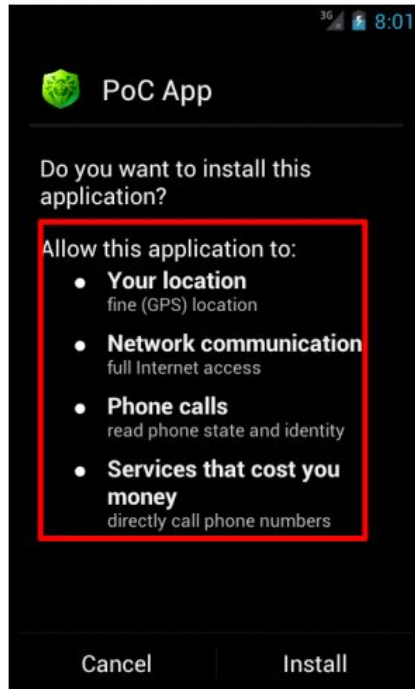
The enforcement of a security policy on different code paths can be inconsistent

- Inconsistencies do exist
- According to the Android documentation
  - apps that hold a CALL\_PHONE permission can end phone calls

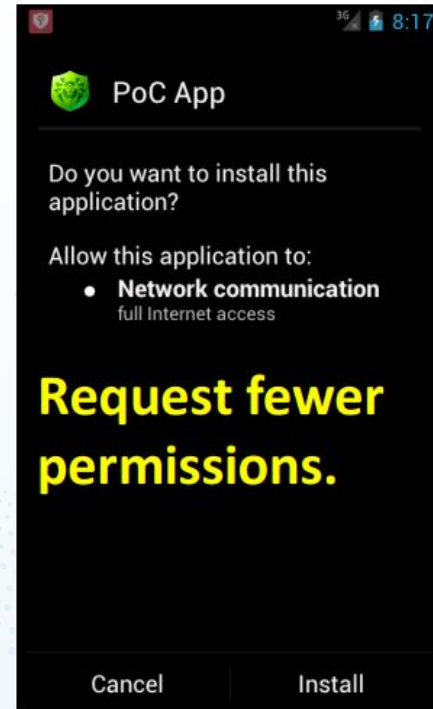


# Security Implication

- Privilege escalation



**Exploiting  
inconsistencies**



# Existing Research

- Inconsistent security policy enforcement is also found in SELinux and Xen [[AutoISES Usenix Sec'08](#)]
  - unauthorized user account access
  - permanent data loss
- No solution for Android framework
  - prior work is OS specific
  - Android has no explicitly defined policies

# Problem Statement

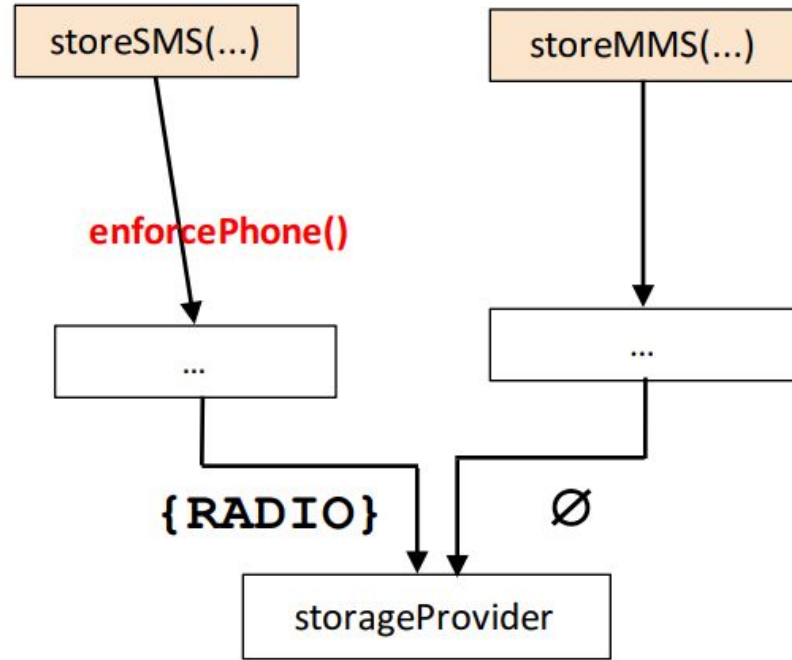
- Focus on the Android framework
- Seek to answer the following question:
  - How can we systematically detect inconsistent security policy enforcement *without any knowledge of the policies?*

# Approach

- Discover app-accessible service interfaces that have overlaps in functionality
  - expected to have consistent security enforcement
- Perform a differential analysis on security checks that two overlapping interfaces employ

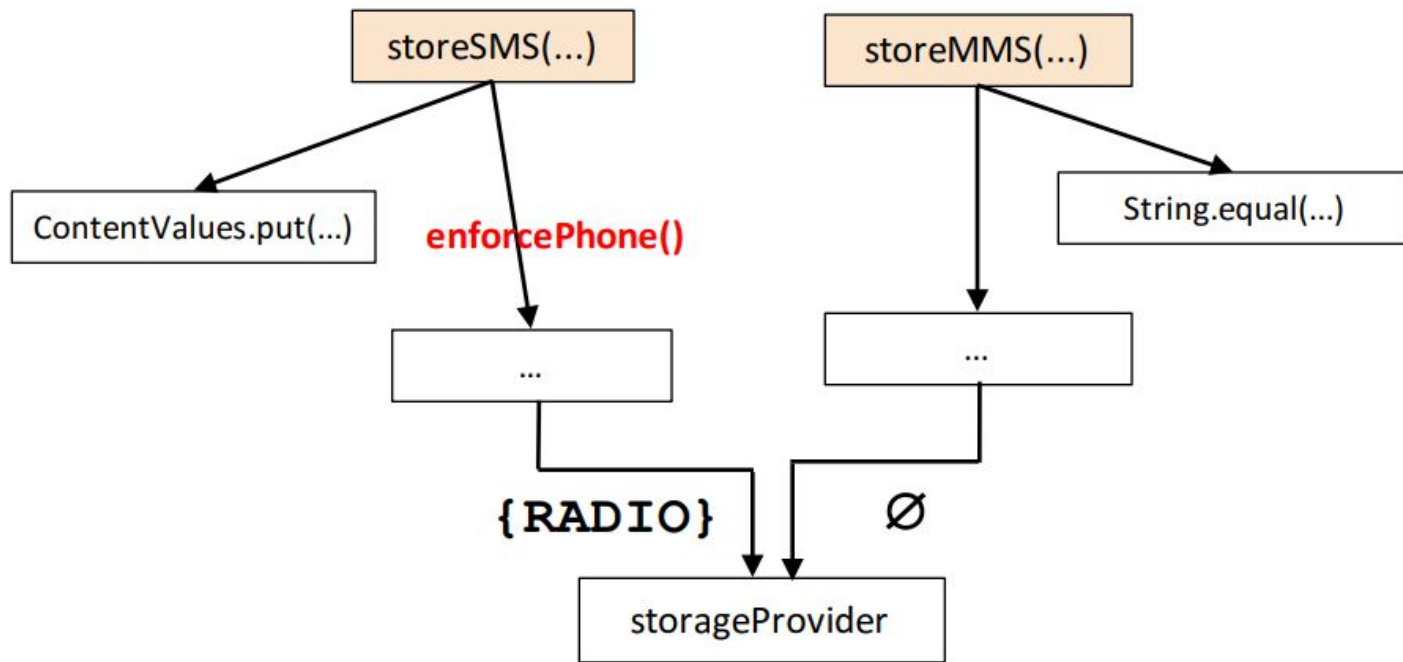


# Differential Analysis



`enforcePhone()` checks if the caller's UID is 1001 (RADIO)

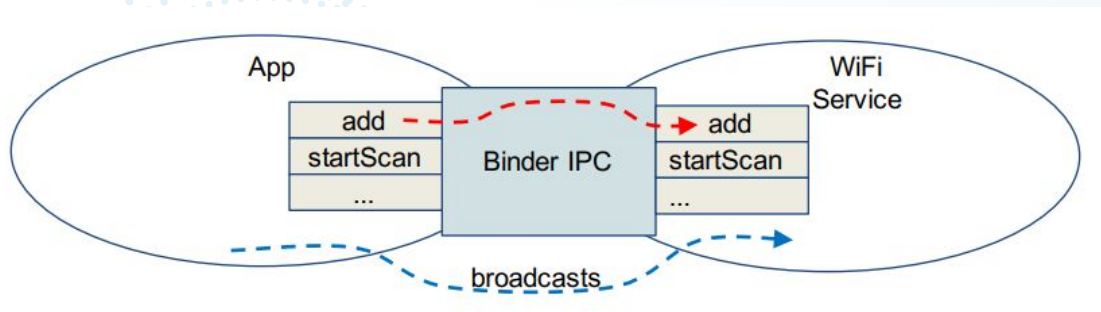
# Pruning



`enforcePhone()` checks if the caller's UID is 1001 (RADIO)

# App-accessible Service Interfaces

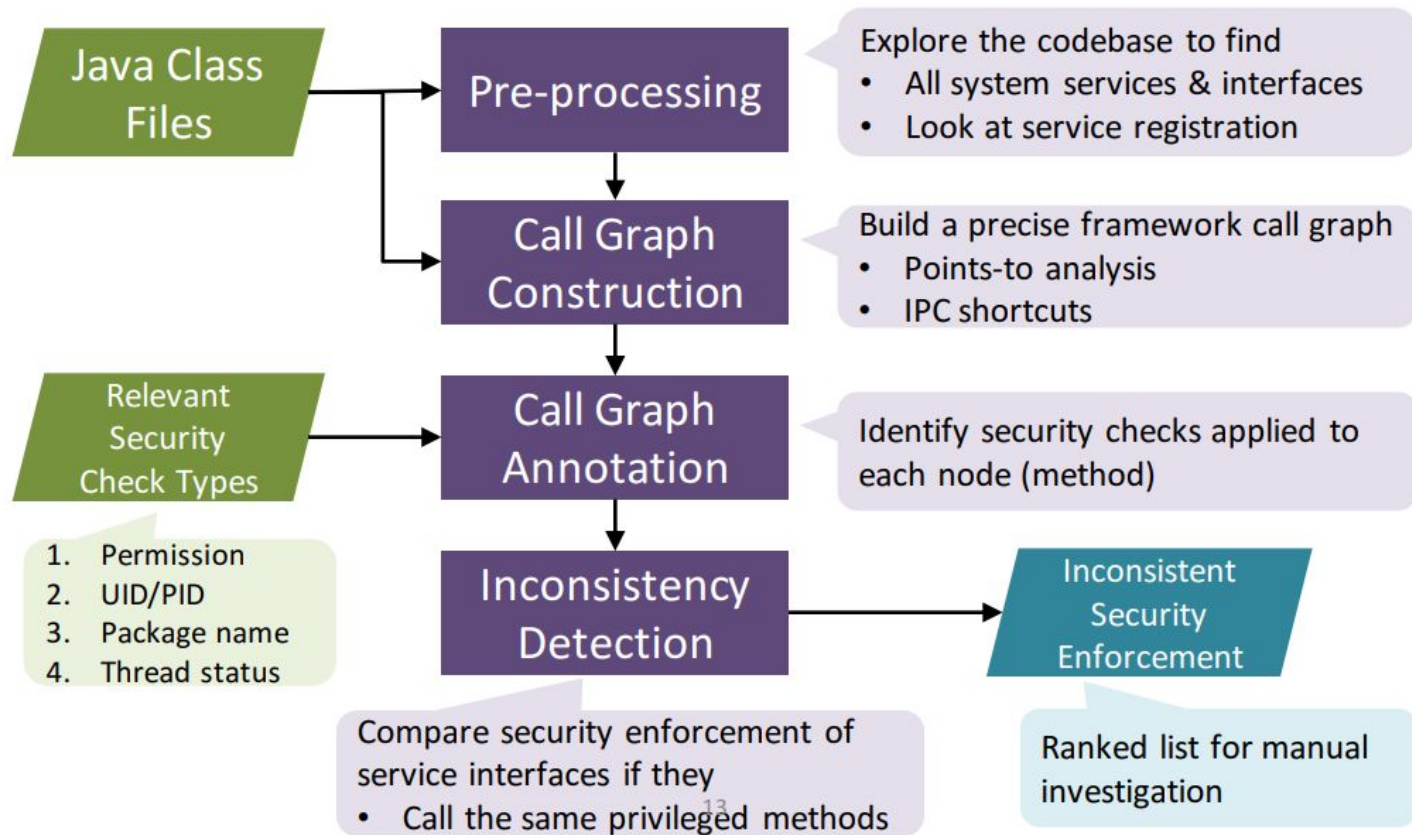
- Analysis scope:
  - system services perform enforcement
- Service interfaces
  - AIDL (Android interface definition language) methods
  - broadcast receivers



# Security Checks

- Security enforcement:
  - a set of security checks
- Kratos formulates 4 types of checks
  - permission check
  - UID/PID check
  - package name check
  - thread status check

# Kratos Design



# Implementation

- Support AOSP and customized frameworks
  - obtain java classes from
    - intermediate building output (AOSP)
    - decompiled dex files (customized)
- Build a precise framework call graph
  - points-to analysis using Spark
  - create an artificial and static entry point including app app-accessible service interfaces
- Perform data-flow analysis
  - identify security check methods
  - collect system services

# Evaluation

- 6 different Android codebases
  - AOSP 4.4, 5.0, 5.1 and M preview
  - HTC one, Samsung Galaxy Note 3
- Accuracy

Codebase	# Inconsistencies	# TP	# FP	Precision	# Exploitable
Android 4.4	21	16	5	76.2%	8
Android 5.0	61	50	11	82.0%	11
Android 5.1	63	49	14	77.8%	10
M Preview	73	58	15	79.5%	8
AT&T HTC One	29	20	9	69.0%	8
T-Mobile Samsung Galaxy Note 3	128	102	26	79.7%	10

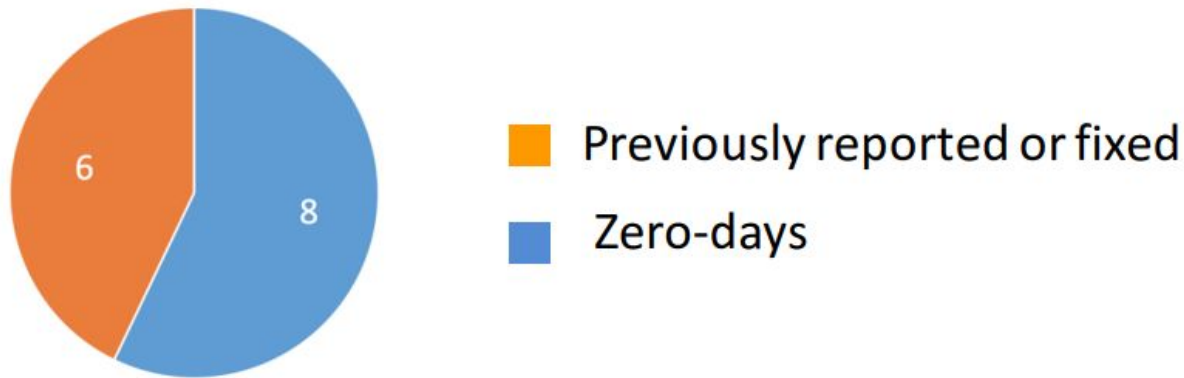
# Evaluation

- False positive
  - two interfaces are not equivalent in functionality
  - points-to analysis  $\Rightarrow$  over-approximated results
- not all inconsistencies are exploitable
  - difficult to construct valid arguments
  - difficult to trigger particular privileged methods



# Evaluation

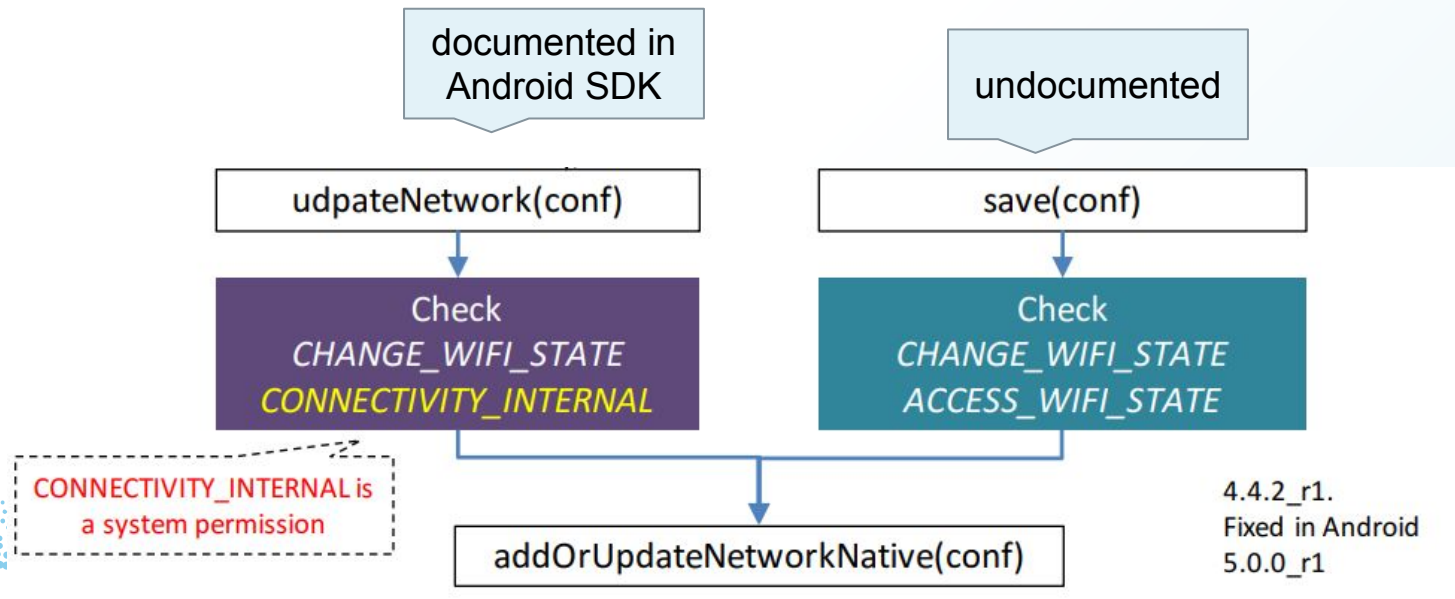
- Found 14 vulnerabilities



- 5 out of 14 affect all codebases
- bug reports confirmed by Google

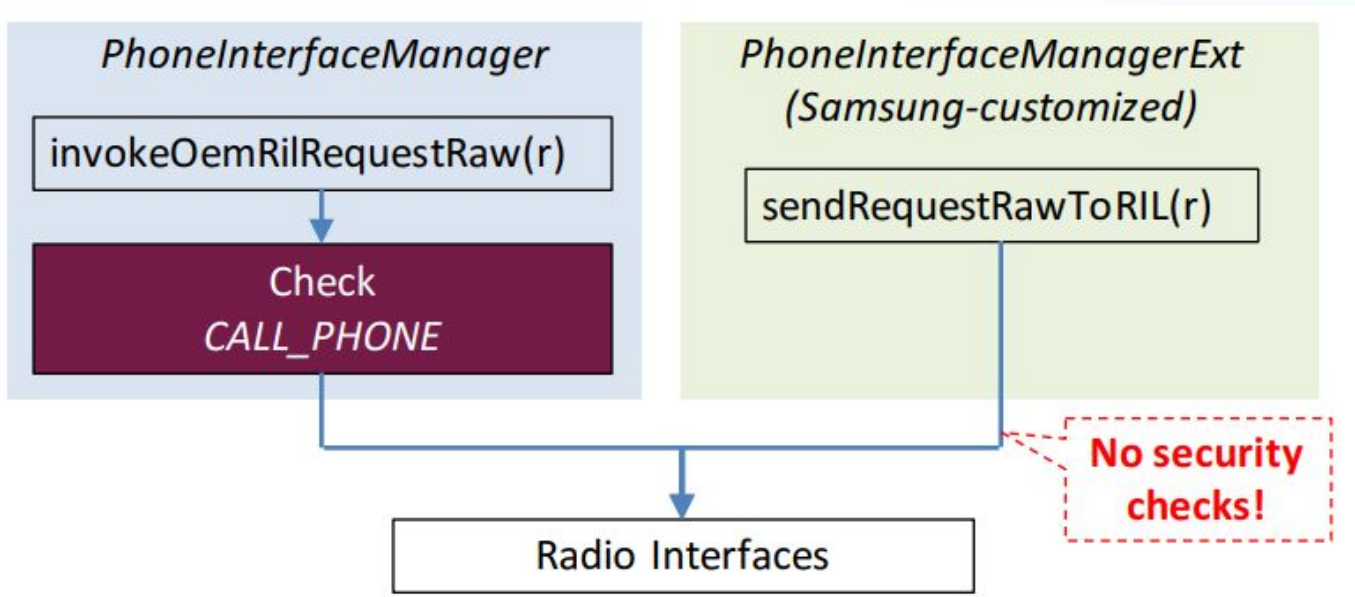
# Case Study 1

- Bypass system permission to change HTTP proxy settings



# Case Study 3

- Send arbitrary requests to the radio hardware without any permission



# Other Observations

- 11 vulnerable interfaces are hidden to apps
  - not available in the Android SDK
  - invoke using Java reflection
- AOSP frameworks
  - new system services introduce new inconsistencies
  - lead to new vulnerabilities
- Customized frameworks
  - Samsung added many system services
    - introduced 2 additional vulnerabilities
    - 1 vulnerability in AOSP was fixed

**Thank you!**

**Questions?**