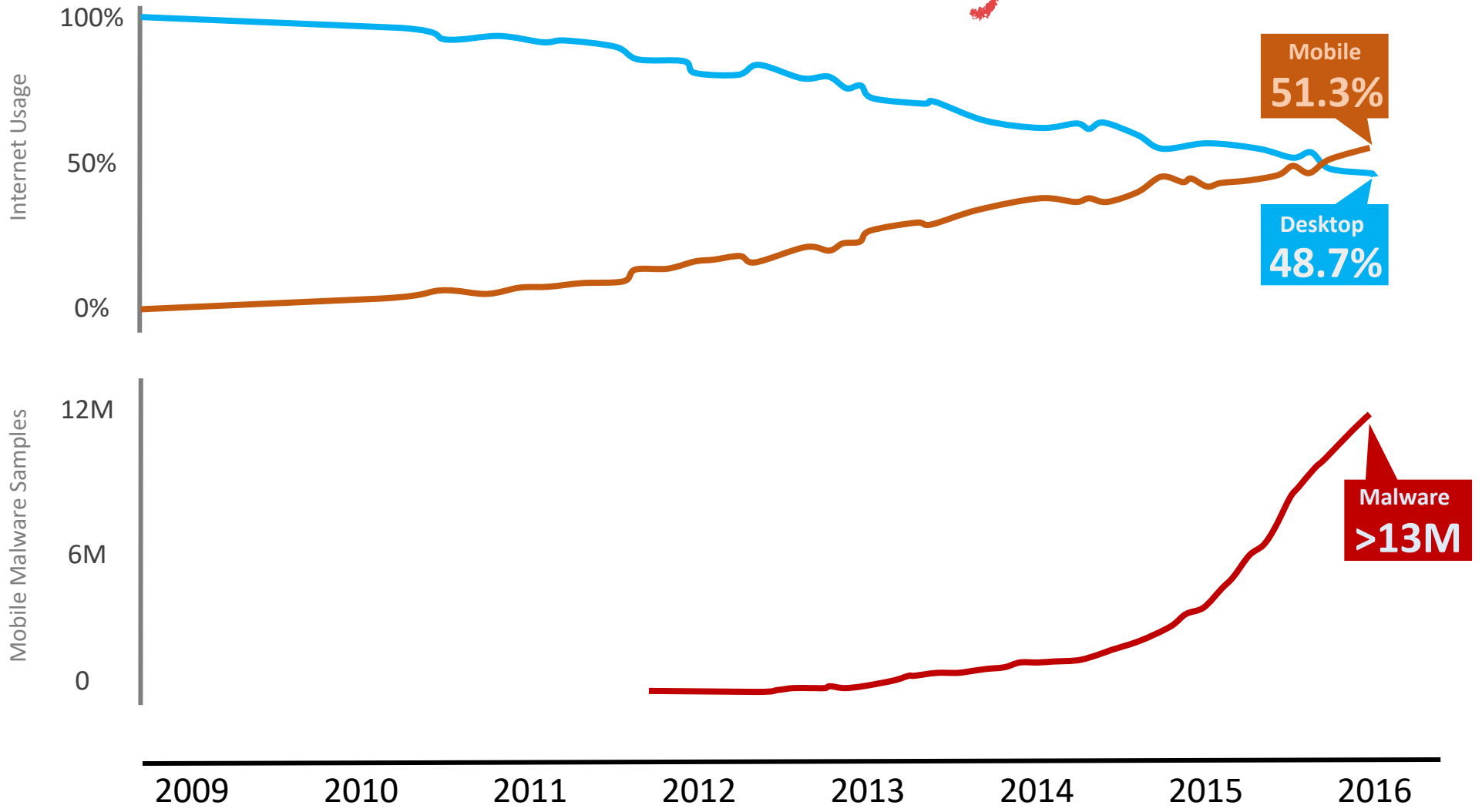
Smartphone Software Security: Guest Lecture

Joshua Garcia

Assistant Professor

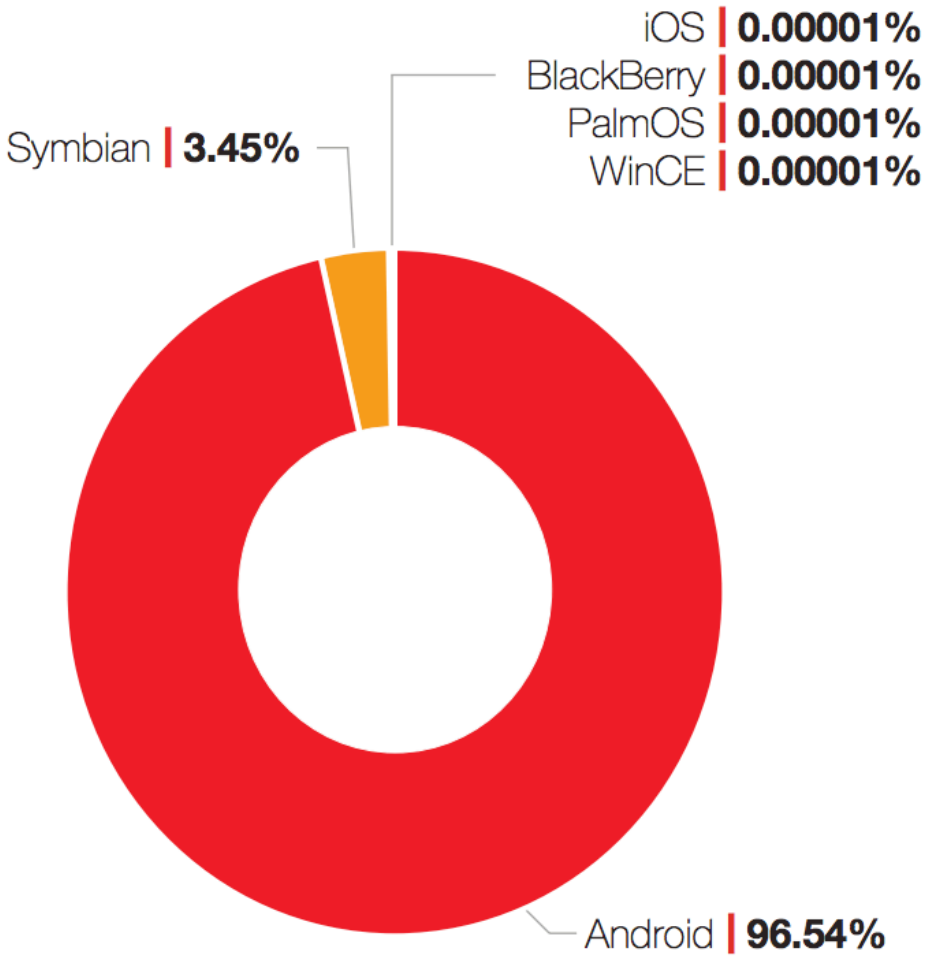
University of California, Irvine

The Rise of ~~Mobile~~ Security Threats



Sources: StatCounter GlobalStats, McAfee

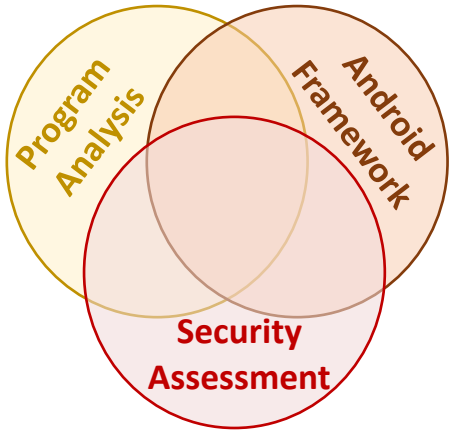
Android is the Primary Target



Source: FORTINET

Structure of Today's Lecture

- A Taxonomy of Program Analysis Techniques for Security Assessment of Android Software
- Automatic Exploit Generation of Android Apps
- Lightweight, Obfuscation-Resilient Android Malware Classification
- Self-Protection of Android Systems from Inter-Component Communication Attacks



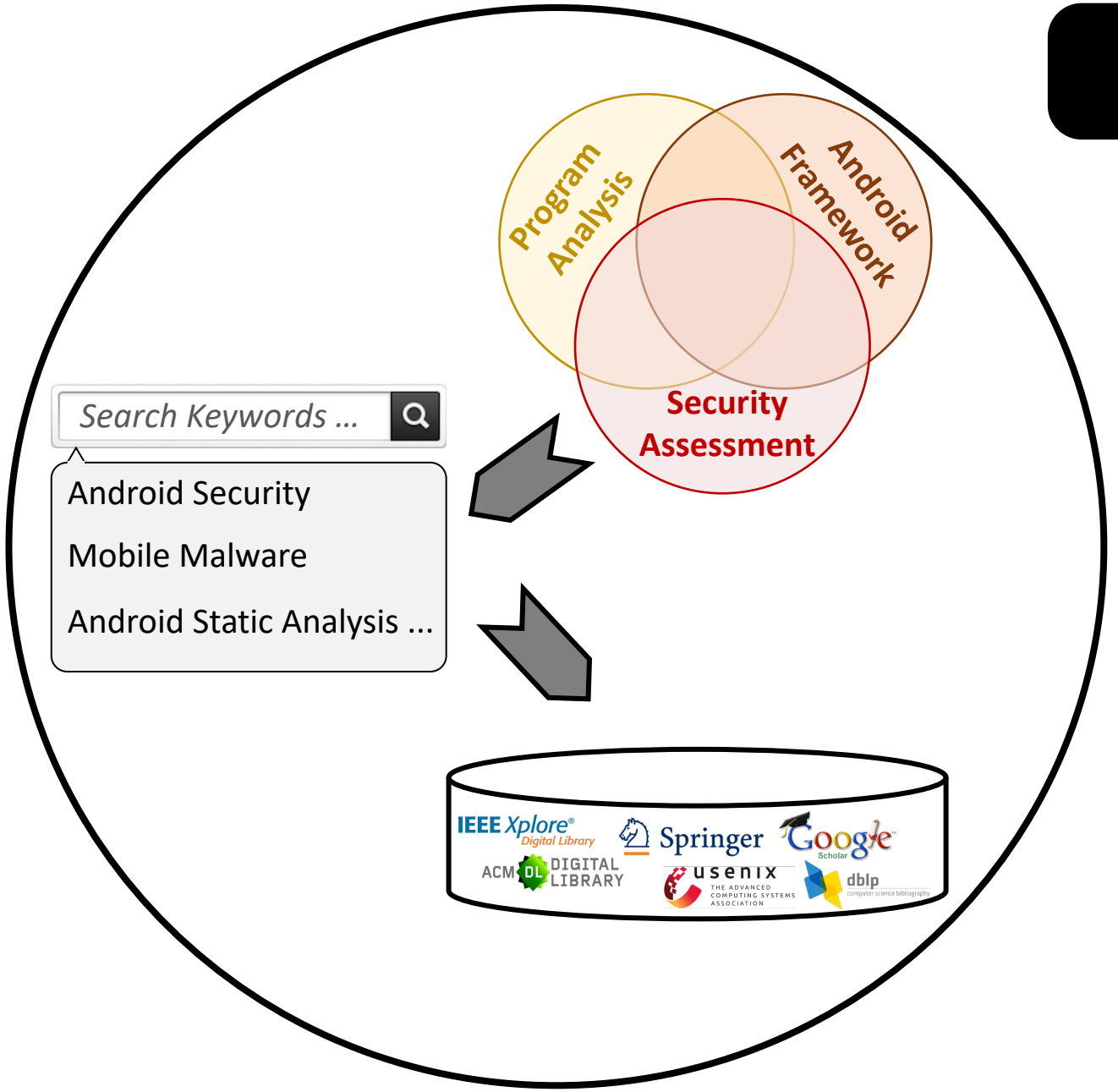
Systematic Literature Review (SLR)

RQ1 How to classify existing research?

RQ2 Current state of existing research?

RQ3 Patterns, trends, gaps, future research?

Iterative Research Process



Paper Collection

Selected Papers

RQ1
- Survey of surveys
- Initial paper review
- Formulate taxonomy
- Refine Taxonomy

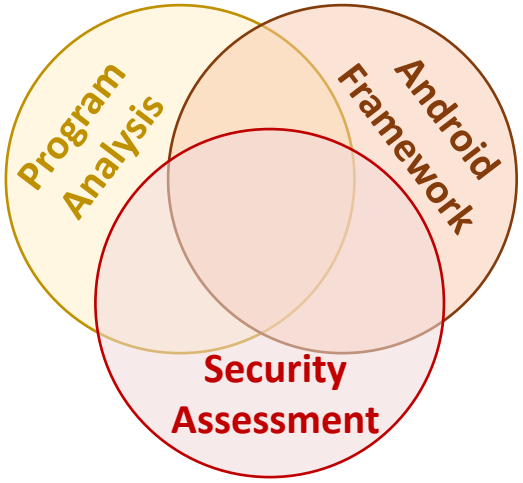
Initial Taxonomy

RQ2
- Detailed paper review
- Papers classification
- Survey analysis

RQ3
- Gap/Trends analysis
- Future research recommendation

Inclusion and Exclusion Criteria

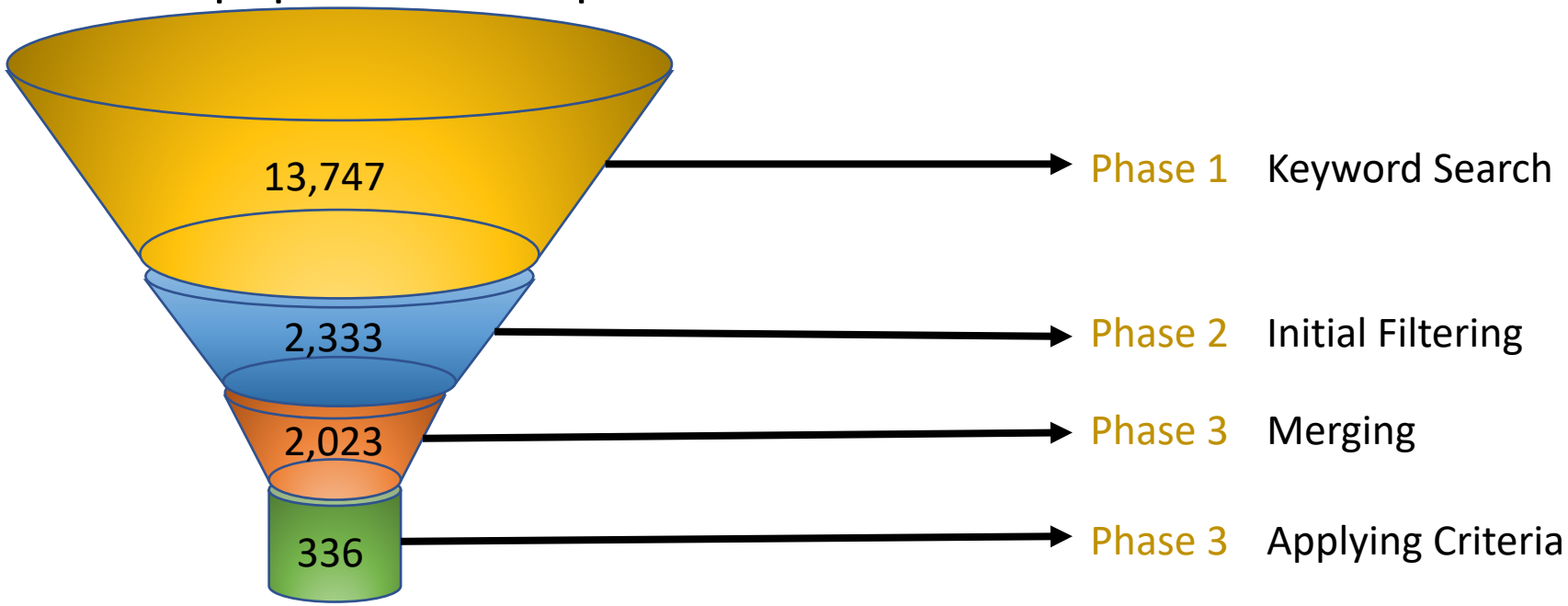
- Inclusion Criteria



- Exclusion Criteria
  1. Not involving the Android platform
  2. Only focusing on mitigation of security threats
  3. Focused on low-level monitoring and profiling techniques
  4. Analysis performed only on meta-data
  5. Only expanding and enhancing Java program analysis
  6. Attacks without detection

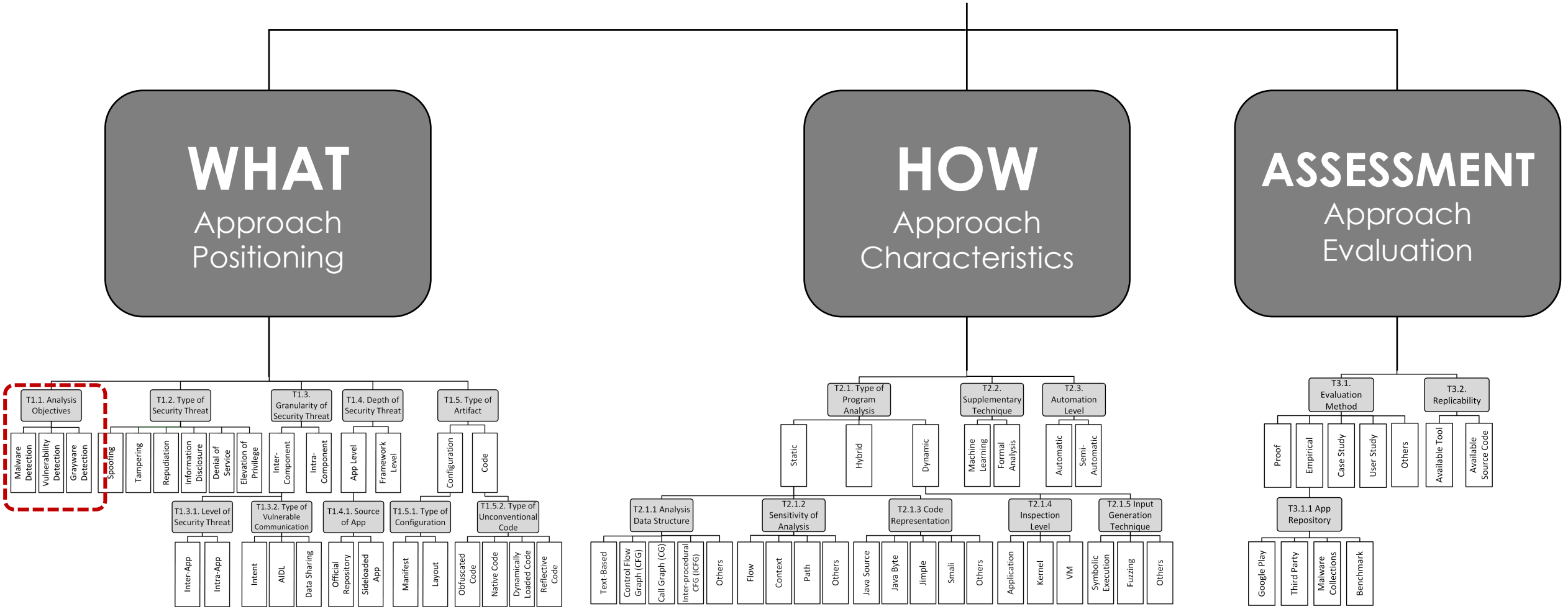
Paper Selection Phases

Number of papers at each phase

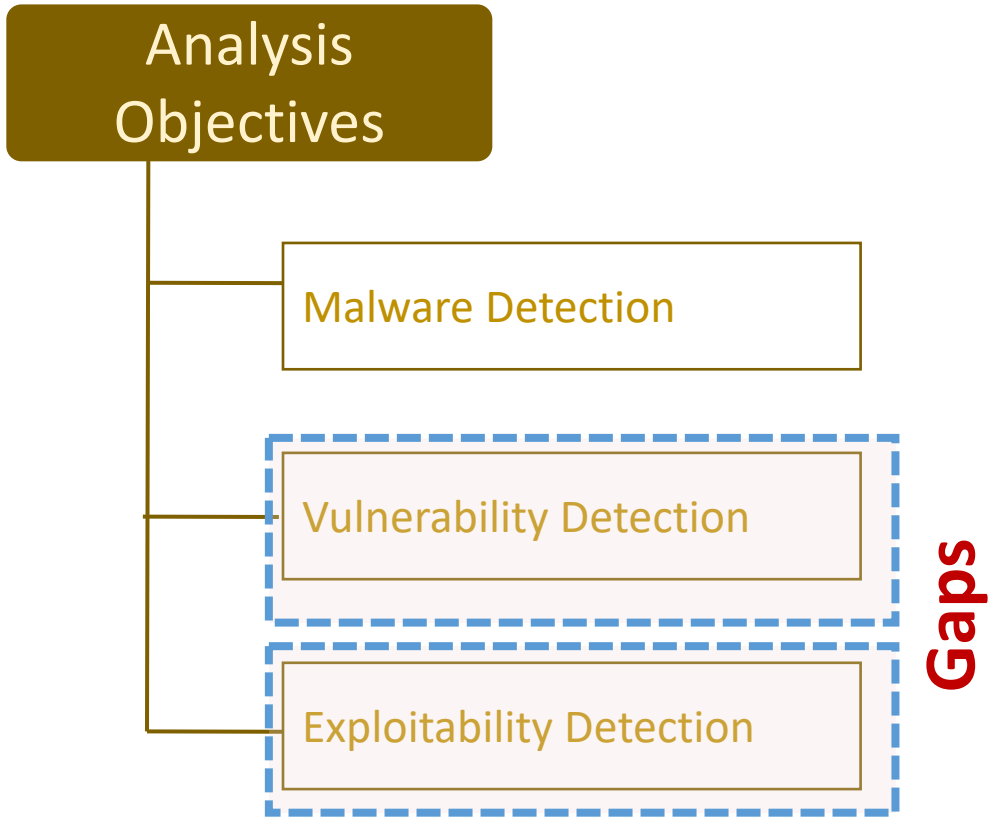


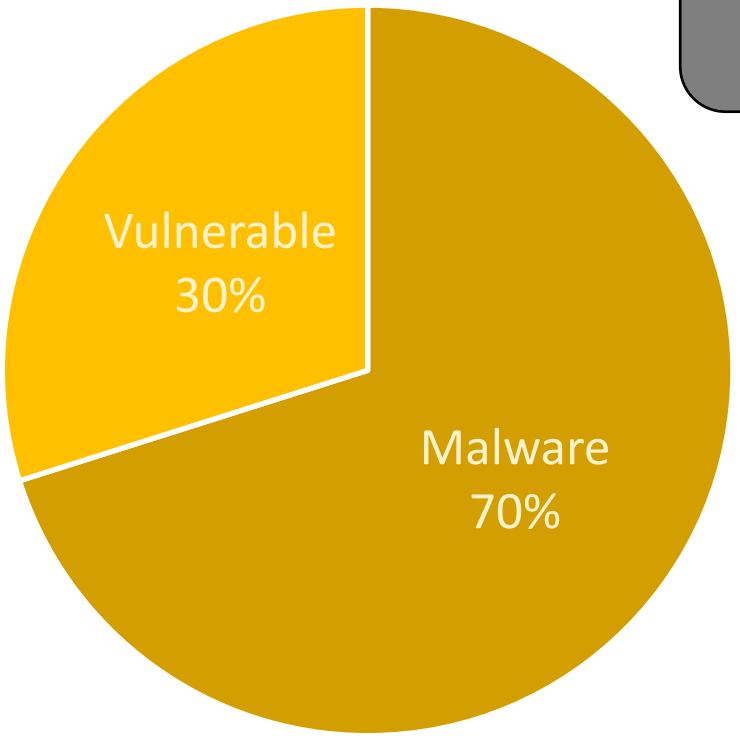
Published in over 100 different journals articles or conference proceedings in **software engineering** (and **PL**), **security**, and **mobile systems**.

Taxonomy


WHAT
Approach Positioning
HOW
Approach Characteristics
ASSESSMENT
Approach Evaluation
T1.1. Analysis Objectives
Malware Detection
Vulnerability Detection
Grayware Detection
T1.2. Type of Security Threat
Spoofing
Tampering
Repudiation
Information Disclosure
Denial of Service
Elevation of Privilege
T1.3. Granularity of Security Threat
Inter-Component
Intra-Component
T1.4. Depth of Security Threat
App Level
Framework Level
T1.5. Type of Artifact
Configuration
Code
T1.3.1. Level of Security Threat
Inter-App
Intra-App
T1.3.2. Type of Vulnerable Communication
Intent
AIDL
Data Sharing
T1.4.1. Source of App
Official Repository
Sideloaded App
T1.5.1. Type of Configuration
Manifest
Layout
T1.5.2. Type of Unconventional Code
Obfuscated Code
Native Code
Dynamically Loaded Code
Reflective Code
T2.1. Type of Program Analysis
Static
Hybrid
Dynamic
T2.2. Supplementary Technique
Machine Learning
Formal Analysis
T2.3. Automation Level
Automatic
Semi-Automatic
T2.1.1 Analysis Data Structure
Text-Based
Control Flow Graph (CFG)
Call Graph (CG)
Inter-procedural CFG (ICFG)
Others
T2.1.2 Sensitivity of Analysis
Flow
Context
Path
Others
T2.1.3 Code Representation
Java Source
Java Byte
Jimple
Smali
Others
T2.1.4 Inspection Level
Application
Kernel
VM
T2.1.5 Input Generation Technique
Symbolic Execution
Fuzzing
Others
T3.1. Evaluation Method
Proof
Empirical
Case Study
User Study
Others
T3.2. Replicability
Available Tool
Available Source Code
T3.1.1 App Repository
Google Play
Third Party
Malware Collections
Benchmark

WHAT
Approach Positioning





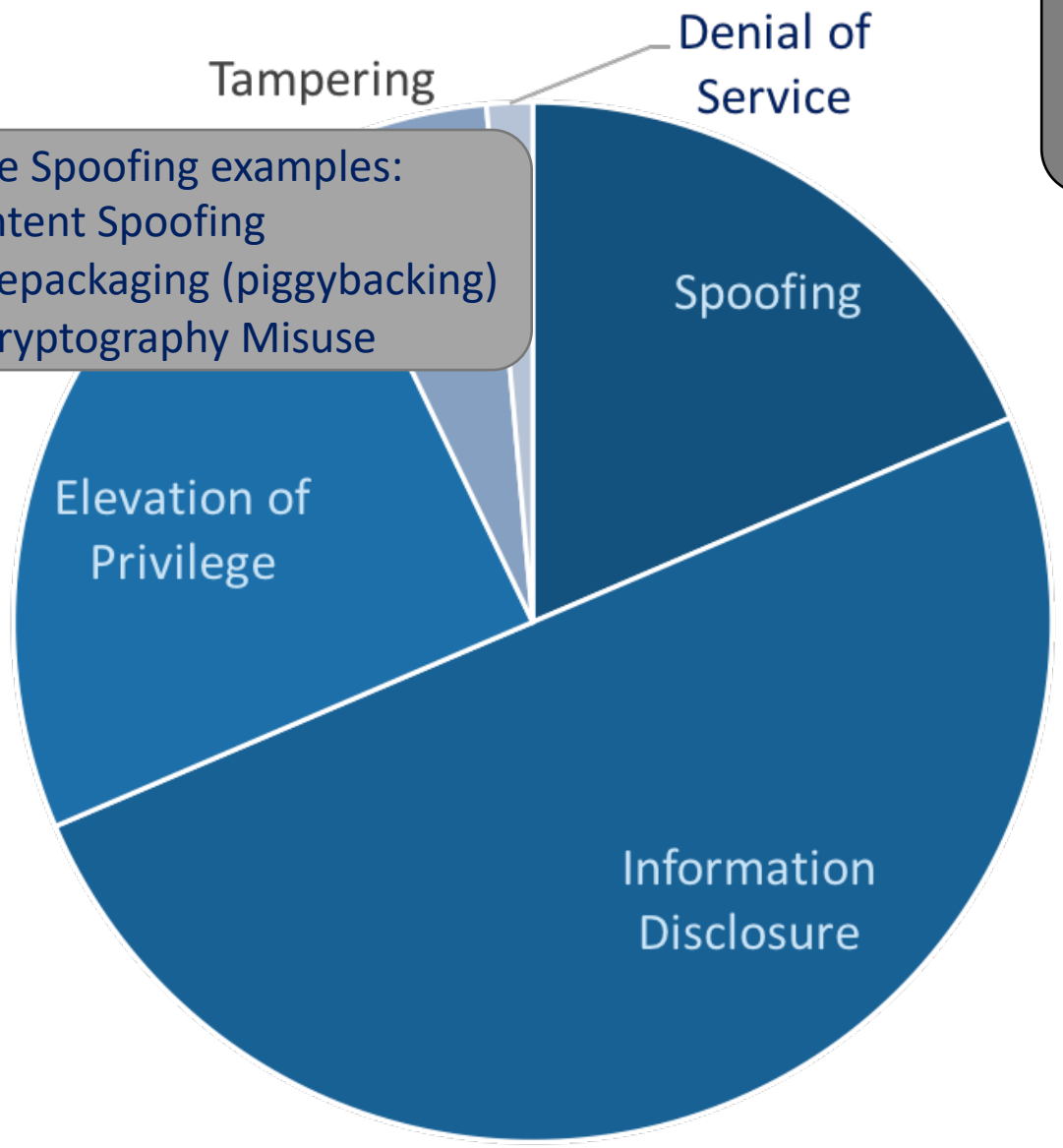
WHAT
Approach
Positioning

Type of Security Threat

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

Some Spoofing examples:
- Intent Spoofing
- Repackaging (piggybacking)
- Cryptography Misuse

Gaps

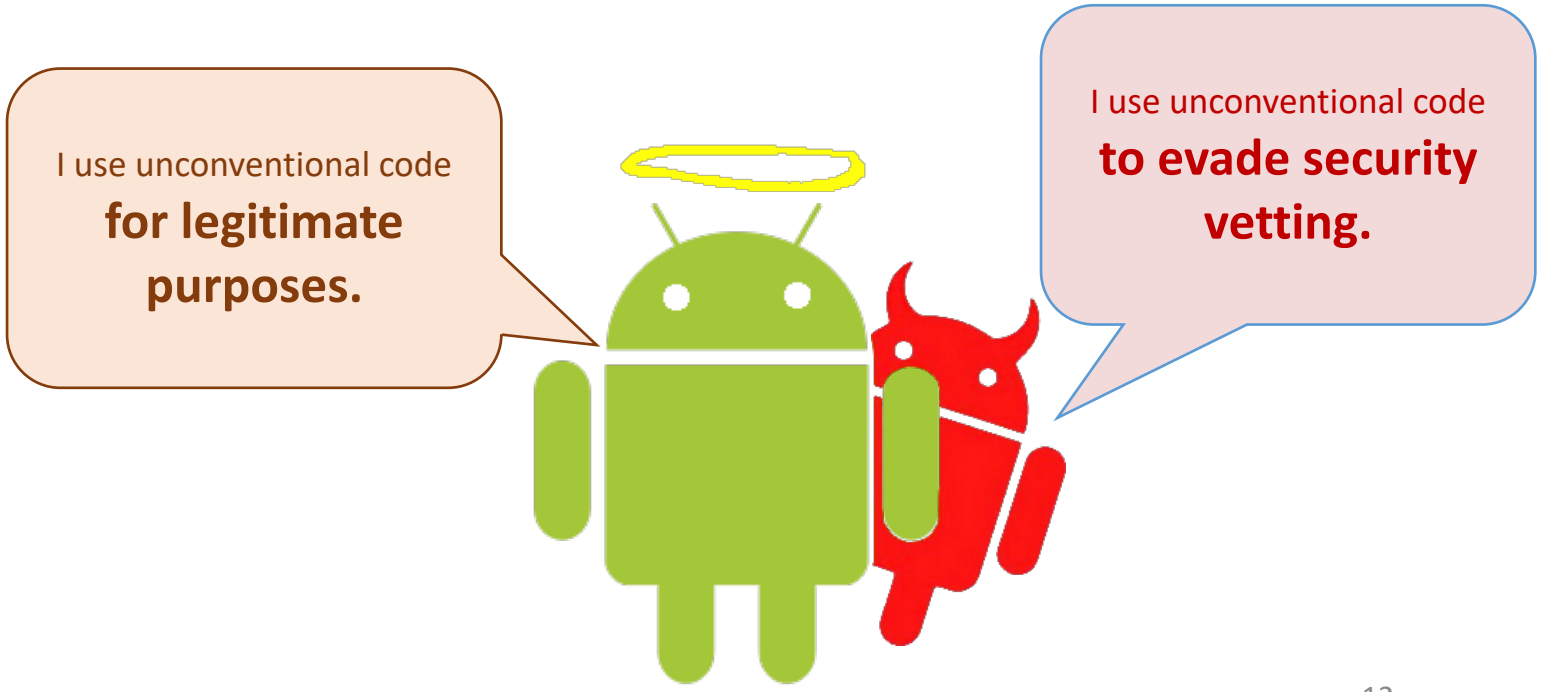


STRIDE
threat model

Type of Unconventional Code

- Obfuscated Code
- Native Code
- Dynamically Loaded Code
- Reflective Code

WHAT
Approach
Positioning

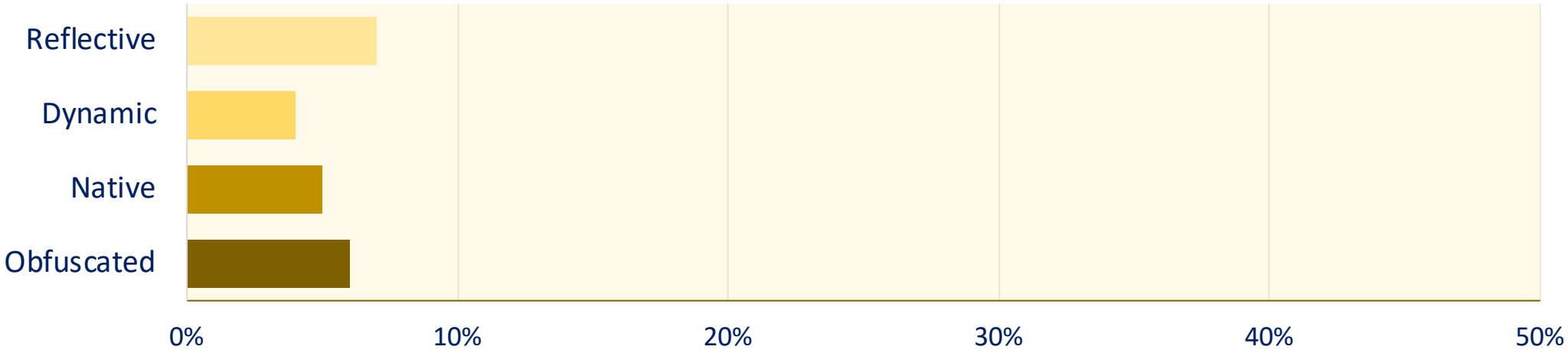


WHAT
Approach Positioning

Type of Unconventional Code

- Obfuscated Code
- Native Code
- Dynamically Loaded Code
- Reflective Code

Gaps



Mobile Malware Using Unconventional Code

AndroidOS/FakeInst Sends SMS uses reflection

2012

- Bootkit with malicious native code
- Over 350,000 devices

2014

- Botnet operating from native code
- 50,000-100,000 downloads

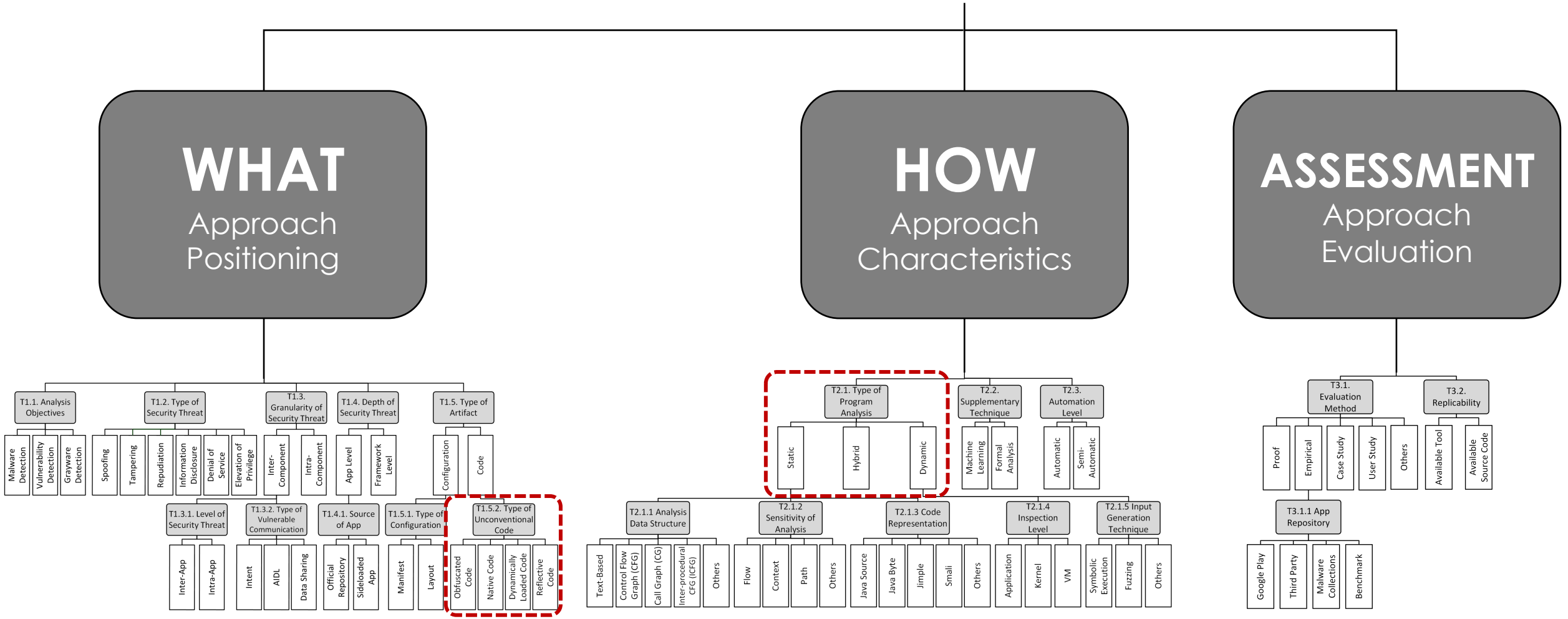
2016

?

2017

Sources: total defense, 360 www.360.cn, Check Point SOFTWARE TECHNOLOGIES LTD.

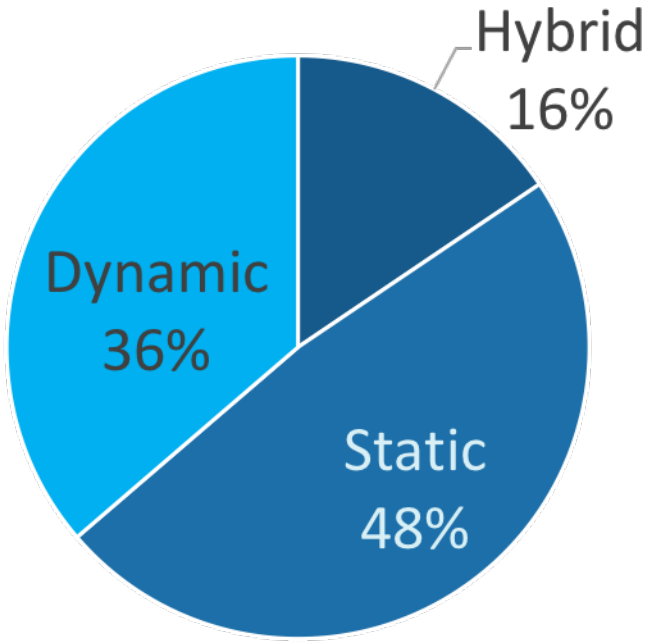
Taxonomy

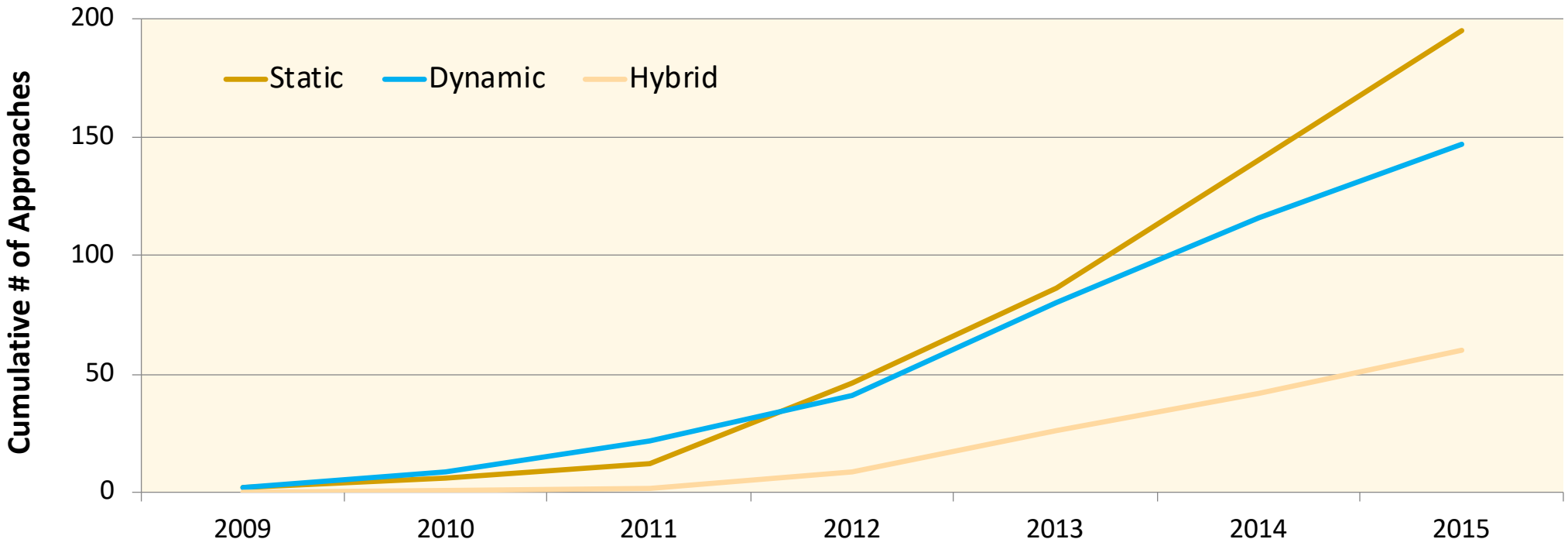

WHAT
Approach Positioning
HOW
Approach Characteristics
ASSESSMENT
Approach Evaluation
T1.1. Analysis Objectives
Malware Detection
Vulnerability Detection
Grayware Detection
T1.2. Type of Security Threat
Spoofing
Tampering
Repudiation
Information Disclosure
Denial of Service
Elevation of Privilege
T1.3. Granularity of Security Threat
Inter-Component
Intra-Component
T1.4. Depth of Security Threat
App Level
Framework Level
T1.5. Type of Artifact
Configuration
Code
T1.3.1. Level of Security Threat
Inter-App
Intra-App
T1.3.2. Type of Vulnerable Communication
Intent
AIDL
Data Sharing
T1.4.1. Source of App
Official Repository
Sideloaded App
T1.5.1. Type of Configuration
Manifest
Layout
T1.5.2. Type of Unconventional Code
Obfuscated Code
Native Code
Dynamically Loaded Code
Reflective Code
T2.1. Type of Program Analysis
Static
Hybrid
Dynamic
T2.2. Supplementary Technique
Machine Learning
Formal Analysis
T2.3. Automation Level
Automatic
Semi-Automatic
T2.1.1 Analysis Data Structure
Text-Based
Control Flow Graph (CFG)
Call Graph (CG)
Inter-procedural CFG (ICFG)
Others
T2.1.2 Sensitivity of Analysis
Flow
Context
Path
Others
T2.1.3 Code Representation
Java Source
Java Byte
Jimple
Smali
Others
T2.1.4 Inspection Level
Application
Kernel
VM
T2.1.5 Input Generation Technique
Symbolic Execution
Fuzzing
Others
T3.1. Evaluation Method
Proof
Empirical
Case Study
User Study
Others
T3.2. Replicability
Available Tool
Available Source Code
T3.1.1 App Repository
Google Play
Third Party
Malware Collections
Benchmark

HOW
Approach Characteristics

Type of Program Analysis

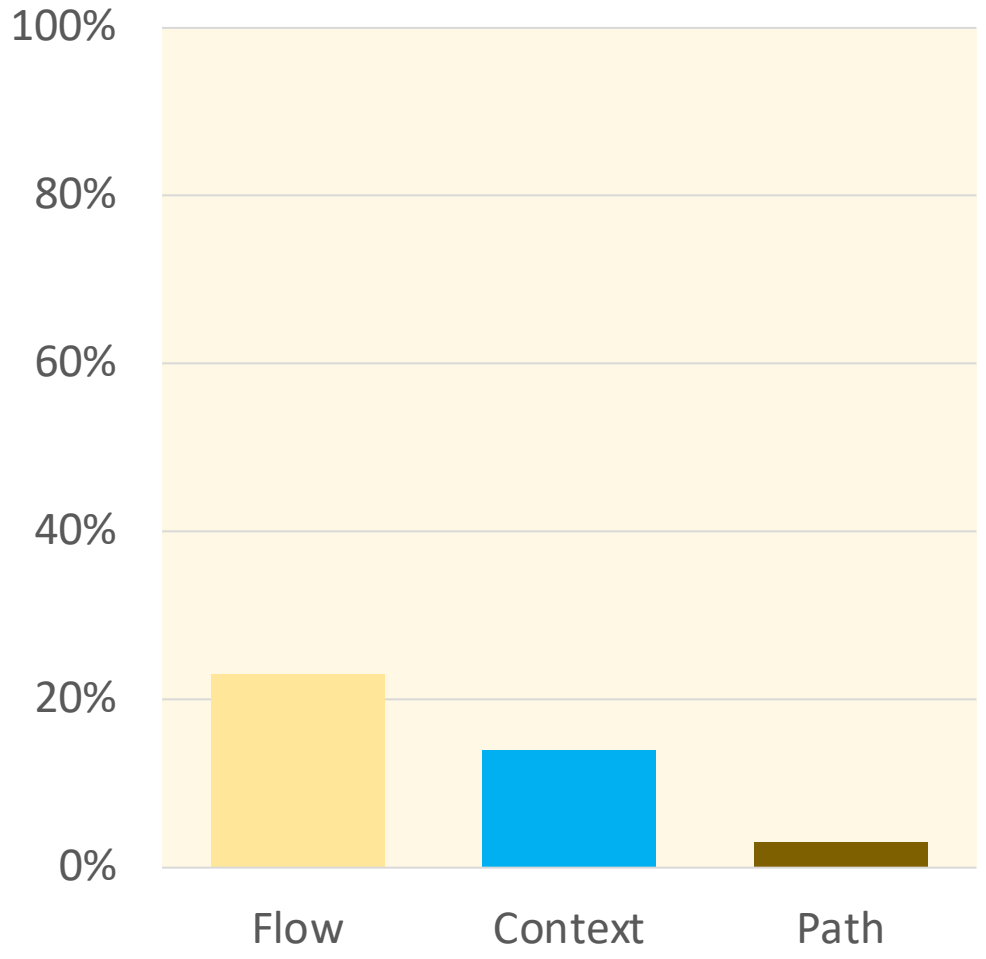
- Static
- Dynamic
- Hybrid





HOW
Approach Characteristics

- Type of Program Analysis
  - Static
  - Dynamic
  - Hybrid
- Sensitivity of Analysis
  - Flow
  - Context
  - Path
  - Others

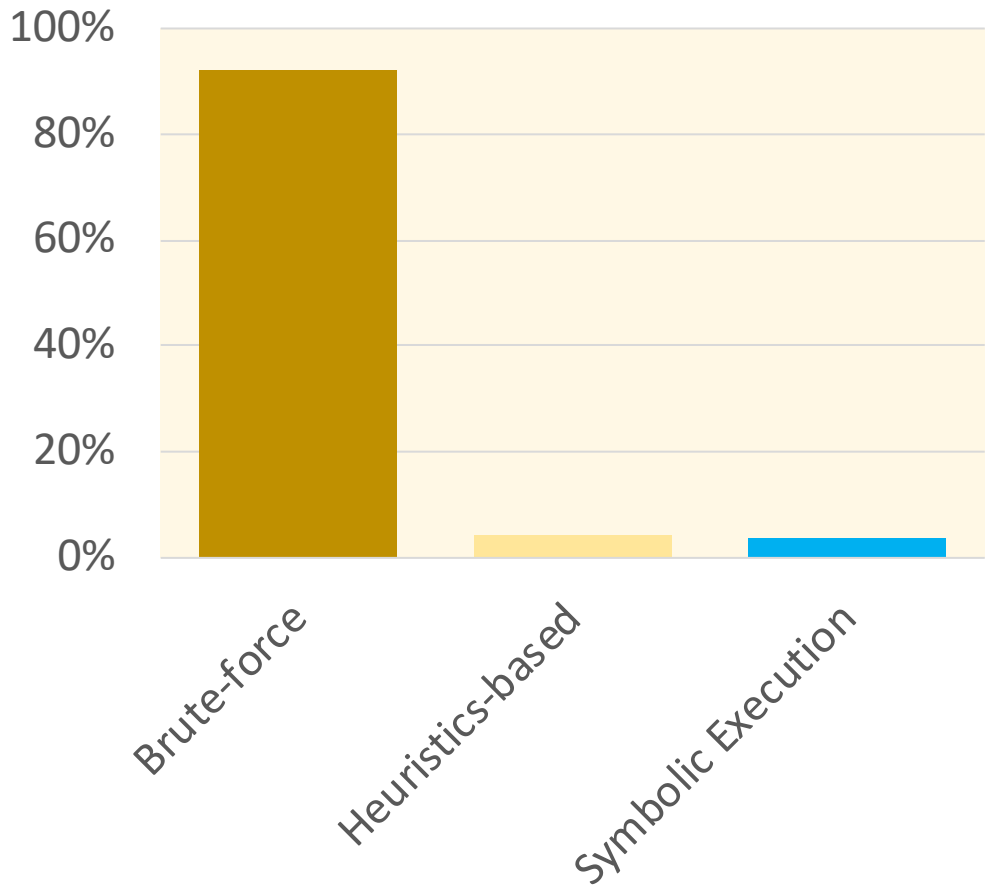


Sensitivity of Analysis

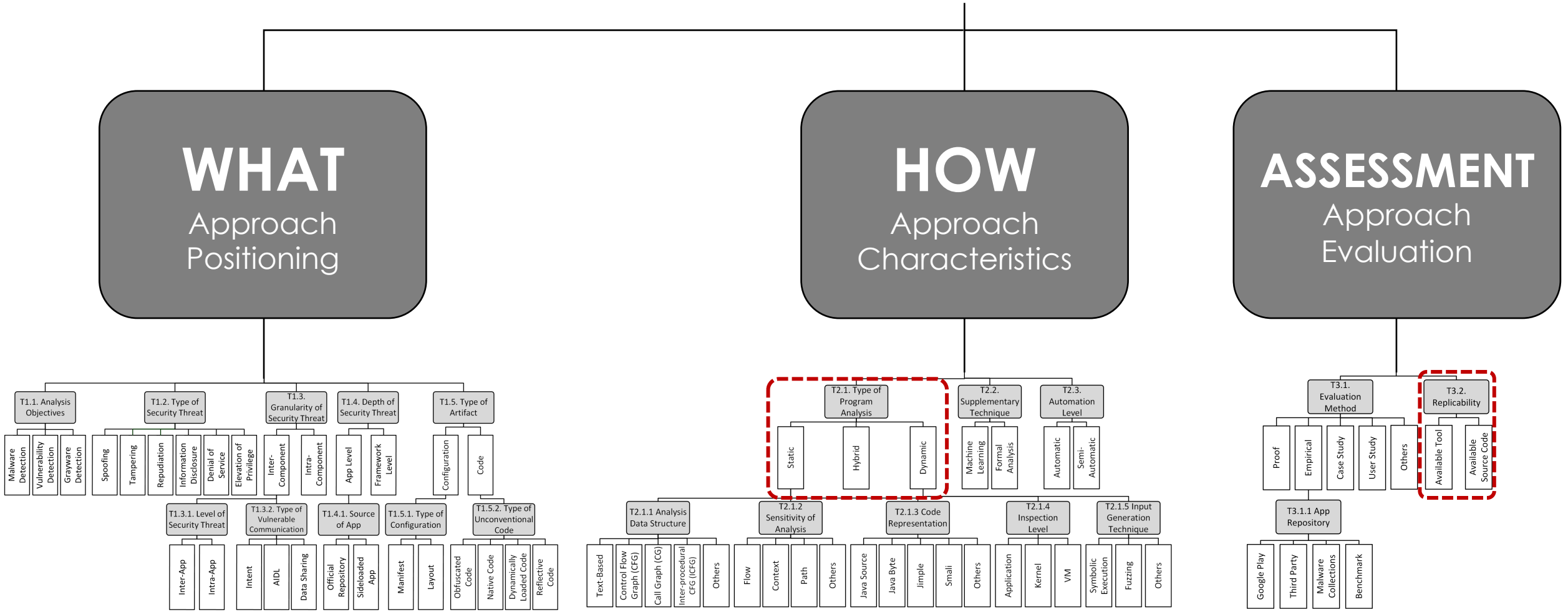
HOW
Approach Characteristics

- Type of Program Analysis
  - Static
  - Dynamic
  - Hybrid
- Input Generation Technique
  - Brute-force fuzzing
  - Heuristic-based fuzzing
  - Symbolic Execution

Systematic



Taxonomy


WHAT
Approach Positioning
HOW
Approach Characteristics
ASSESSMENT
Approach Evaluation
T1.1. Analysis Objectives
T1.2. Type of Security Threat
T1.3. Granularity of Security Threat
T1.4. Depth of Security Threat
T1.5. Type of Artifact
Malware Detection
Vulnerability Detection
Grayware Detection
Spoofing
Tampering
Repudiation
Information Disclosure
Denial of Service
Elevation of Privilege
Inter-Component
Intra-Component
App Level
Framework Level
Configuration
Code
T1.3.1. Level of Security Threat
T1.3.2. Type of Vulnerable Communication
T1.4.1. Source of App
T1.5.1. Type of Configuration
T1.5.2. Type of Unconventional Code
Inter-App
Intra-App
Intent
AIDL
Data Sharing
Official Repository
Sideloaded App
Manifest
Layout
Obfuscated Code
Native Code
Dynamically Loaded Code
Reflective Code
T2.1. Type of Program Analysis
T2.2. Supplementary Technique
T2.3. Automation Level
Static
Hybrid
Dynamic
Machine Learning
Formal Analysis
Automatic
Semi-Automatic
T2.1.1 Analysis Data Structure
T2.1.2 Sensitivity of Analysis
T2.1.3 Code Representation
T2.1.4 Inspection Level
T2.1.5 Input Generation Technique
Text-Based
Control Flow Graph (CFG)
Call Graph (CG)
Inter-procedural CFG (ICFG)
Others
Flow
Context
Path
Others
Java Source
Java Byte
Jimple
Smali
Others
Application
Kernel
VM
Symbolic Execution
Fuzzing
Others
T3.1. Evaluation Method
T3.2. Replicability
Proof
Empirical
Case Study
User Study
Others
Available Tool
Available Source Code
T3.1.1 App Repository
Google Play
Third Party
Malware Collections
Benchmark

A Gap in Approach Evaluation …

Sharing research artifacts

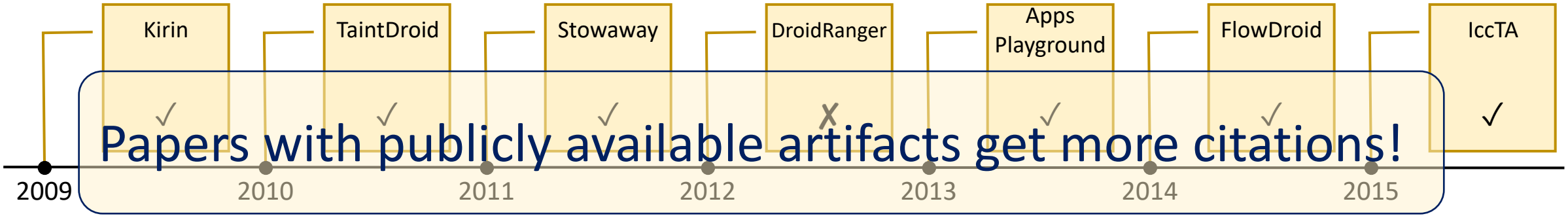
Only 17% of research artifacts are publicly available

Papers with publicly available artifacts get more citations!

Benefits of Sharing

All top five highly cited papers have their research artifacts publicly available:

1. TaintDroid [2010] : 1563
2. Stowaway [2011] : 745
3. Kirin [2009] : 625
4. Enck et al. [2011] : 599
5. ComDroid [2011] : 502



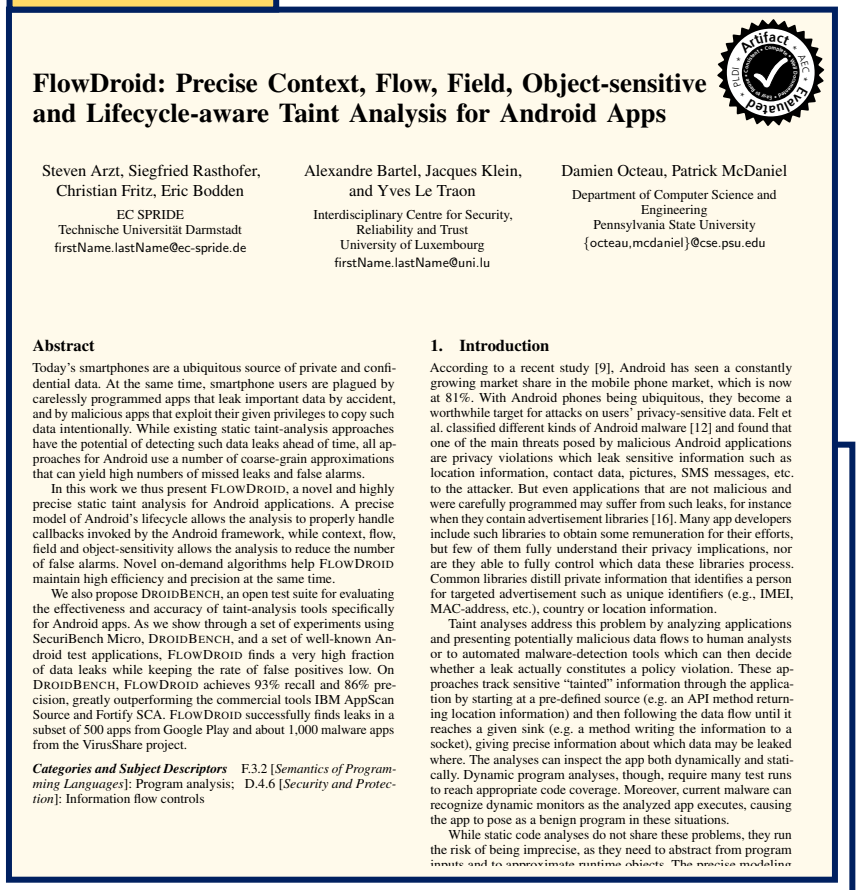
Used in the **evaluation** of:

AppAudit, IccTA, Apparecium, FUSE, WeChecker, DroidSafe, …

Used in the **implementation** of:

AppContext, COVERT, Pedal, DroidADDMiner, Harehunter, PCLeaks, DidFail, …

FlowDroid



FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden
EC SPRIDE
Technische Universität Darmstadt
firstName.lastName@ec-spride.de

Alexandre Bartel, Jacques Klein, and Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
firstName.lastName@uni.lu

Damien Octeau, Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
{octeau,mcdaniel}@cse.psu.edu

Abstract

Today's smartphones are a ubiquitous source of private and confidential data. At the same time, smartphone users are plagued by carelessly programmed apps that leak important data by accident, and by malicious apps that exploit their given privileges to copy such data intentionally. While existing static taint-analysis approaches have the potential of detecting such data leaks ahead of time, all approaches for Android use a number of coarse-grain approximations that can yield high numbers of missed leaks and false alarms.

In this work we thus present FLOWDROID, a novel and highly precise static taint analysis for Android applications. A precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and object-sensitivity allows the analysis to reduce the number of false alarms. Novel on-demand algorithms help FLOWDROID maintain high efficiency and precision at the same time.

We also propose DROIDBENCH, an open test suite for evaluating the effectiveness and accuracy of taint-analysis tools specifically for Android apps. As we show through a set of experiments using SecuriBench Micro, DROIDBENCH, and a set of well-known Android test applications, FLOWDROID finds a very high fraction of data leaks while keeping the rate of false positives low. On DROIDBENCH, FLOWDROID achieves 93% recall and 86% precision, greatly outperforming the commercial tools IBM AppScan Source and Fortify SCA. FLOWDROID successfully finds leaks in a subset of 500 apps from Google Play and about 1,000 malware apps from the VirusShare project.

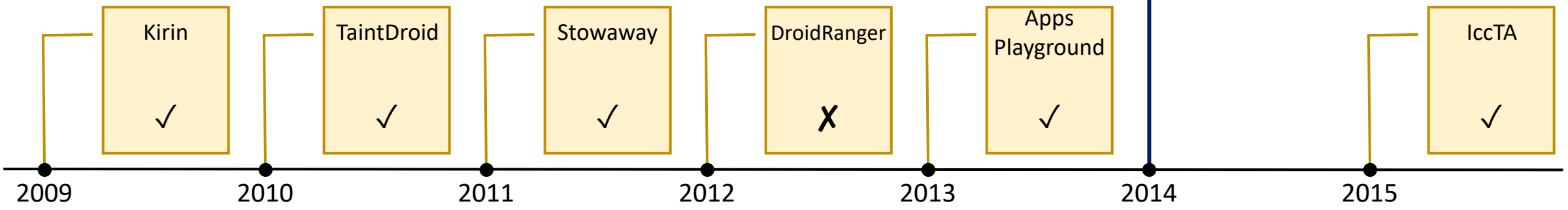
Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; D.4.6 [*Security and Protection*]: Information flow controls

1. Introduction

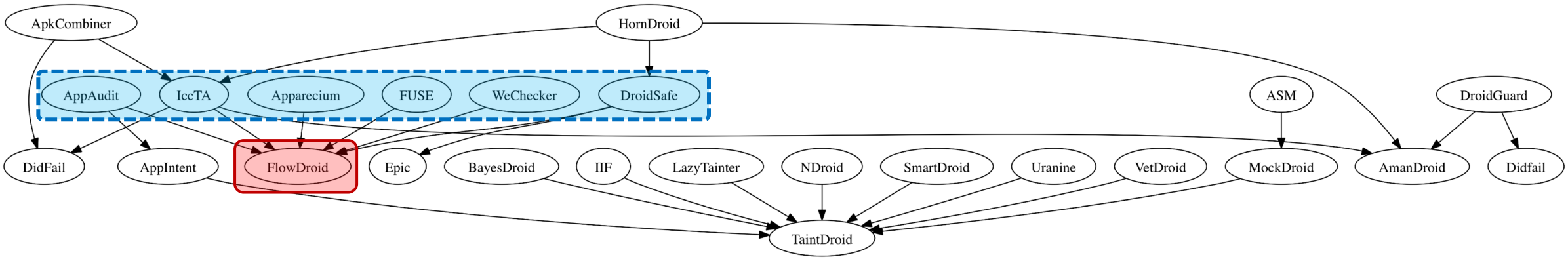
According to a recent study [9], Android has seen a constantly growing market share in the mobile phone market, which is now at 81%. With Android phones being ubiquitous, they become a worthwhile target for attacks on users' privacy-sensitive data. Felt et al. classified different kinds of Android malware [12] and found that one of the main threats posed by malicious Android applications are privacy violations which leak sensitive information such as location information, contact data, pictures, SMS messages, etc. to the attacker. But even applications that are not malicious and were carefully programmed may suffer from such leaks, for instance when they contain advertisement libraries [16]. Many app developers include such libraries to obtain some remuneration for their efforts, but few of them fully understand their privacy implications, nor are they able to fully control which data these libraries process. Common libraries distill private information that identifies a person for targeted advertisement such as unique identifiers (e.g., IMEI, MAC-address, etc.), country or location information.

Taint analyses address this problem by analyzing applications and presenting potentially malicious data flows to human analysts or to automated malware-detection tools which can then decide whether a leak actually constitutes a policy violation. These approaches track sensitive "tainted" information through the application by starting at a pre-defined source (e.g. an API method returning location information) and then following the data flow until it reaches a given sink (e.g. a method writing the information to a socket), giving precise information about which data may be leaked where. The analyses can inspect the app both dynamically and statically. Dynamic program analyses, though, require many test runs to reach appropriate code coverage. Moreover, current malware can recognize dynamic monitors as the analyzed app executes, causing the app to pose as a benign program in these situations.

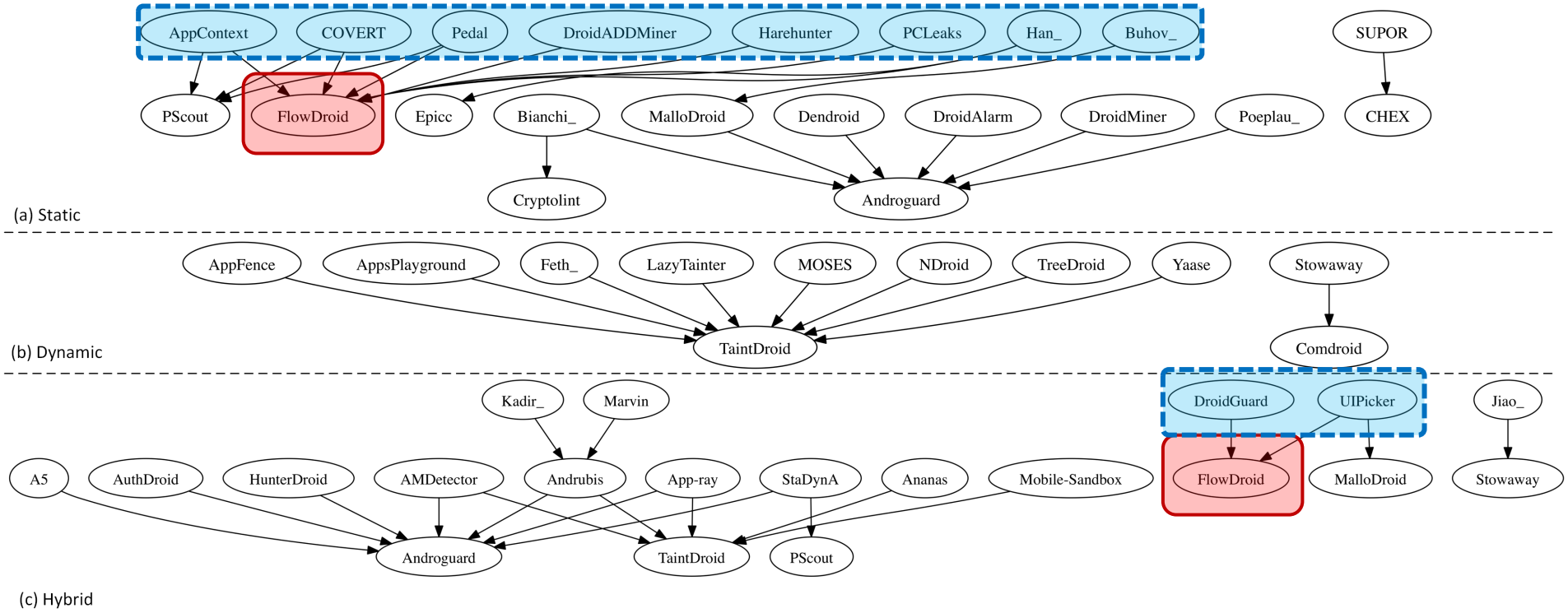
While static code analyses do not share these problems, they run the risk of being imprecise, as they need to abstract from program inputs and to approximate runtime objects. The precise modeling



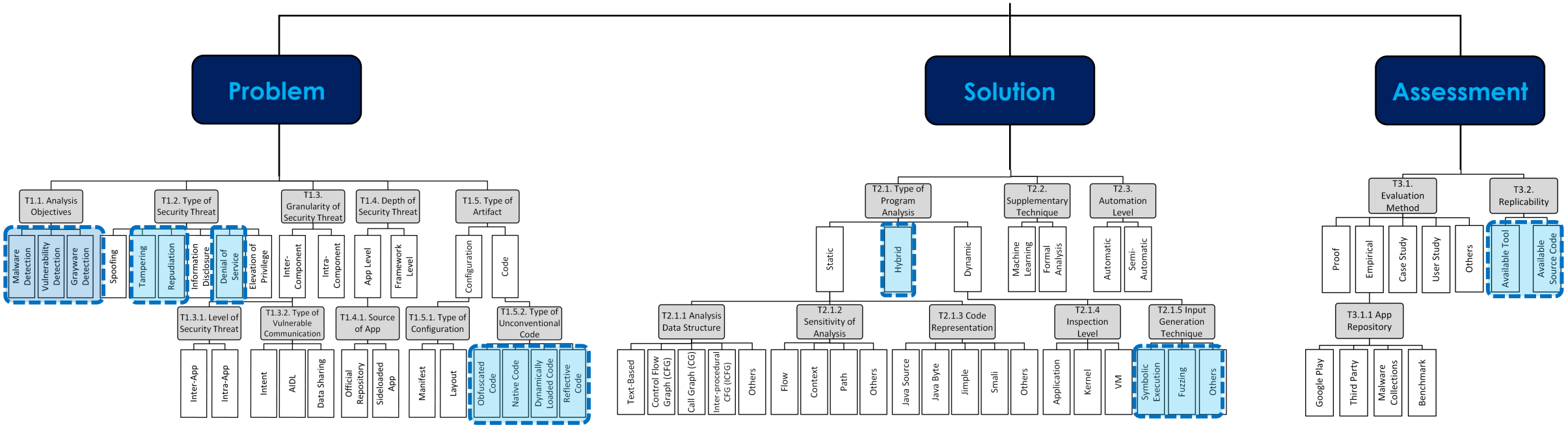
Compared Approaches in the Literature



Standing on Each Other's Shoulders



Areas for Future Research



- Analysis objectives
- Types of security threats
- Analysis of unconventional code

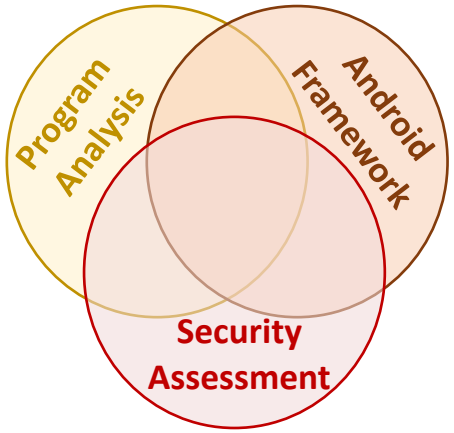
- Hybrid approaches
- Systematic input generation

- Research replicability

Survey catalog: http://www.ics.uci.edu/~seal/projects/droid-sec-taxonomy/index.html

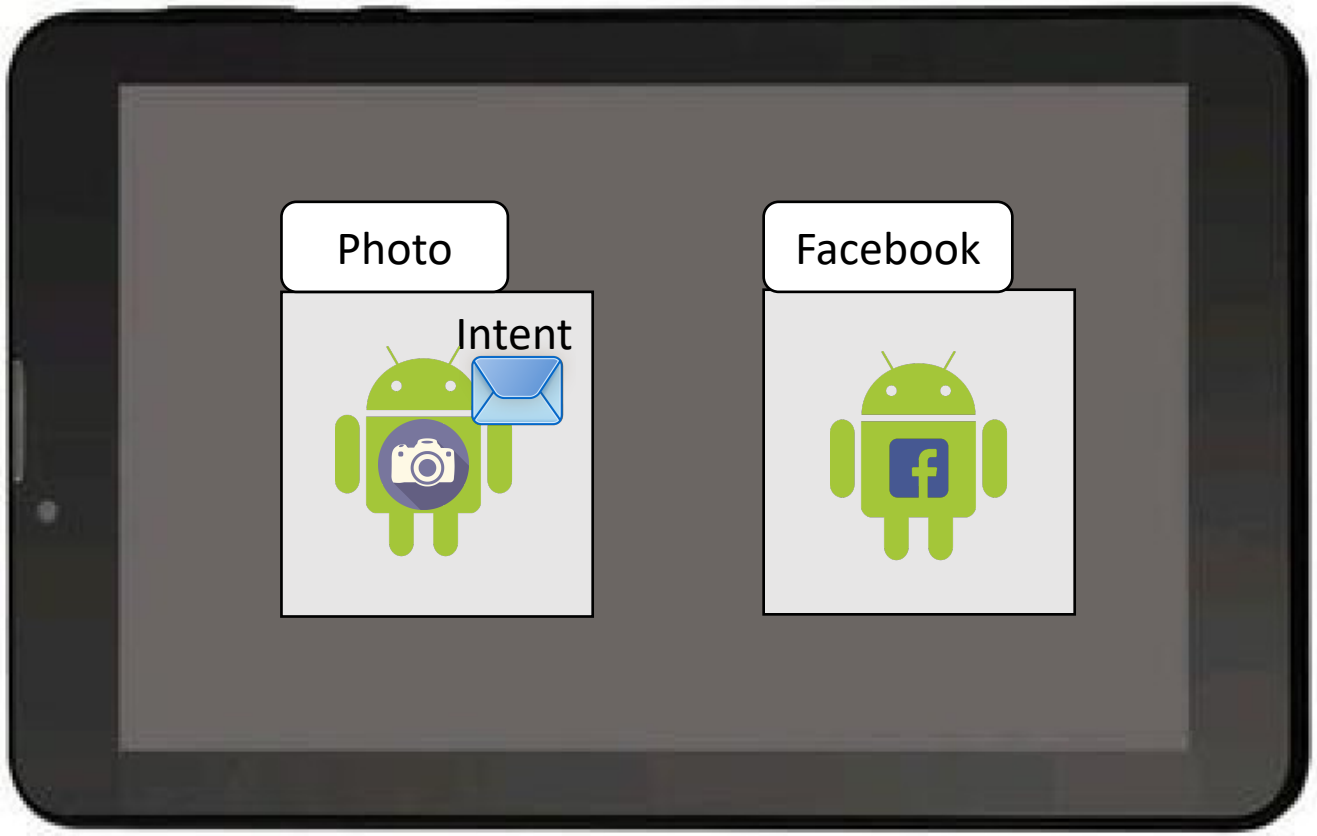
Structure of Today's Lecture

- A Taxonomy of Program Analysis Techniques for Security Assessment of Android Software
- Automatic Exploit Generation of Android Apps
- Lightweight, Obfuscation-Resilient Android Malware Classification
- Self-Protection of Android Systems from Inter-Component Communication Attacks



Inter-Component Communication in Android

- Intent: Android event message for inter-component communication (ICC)



Intent

action : SHARE

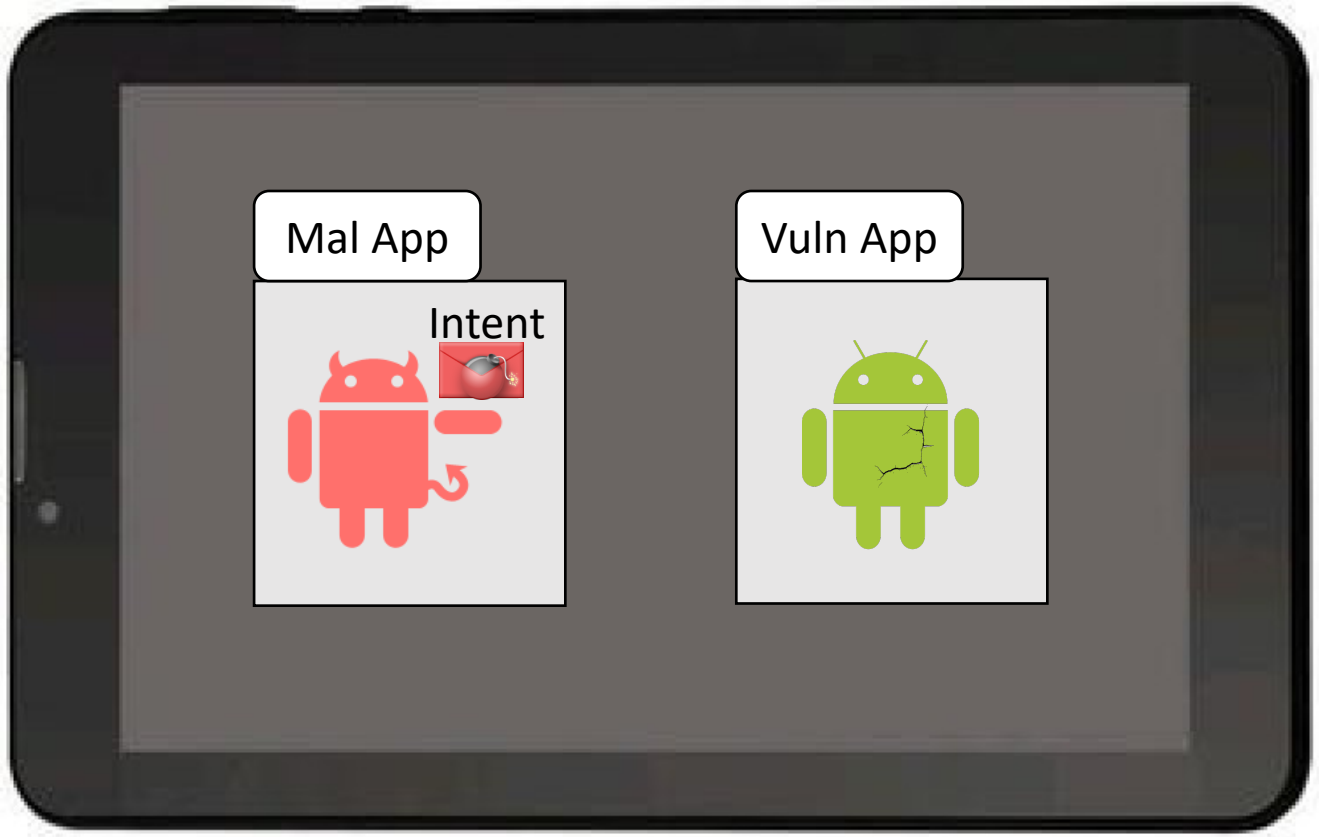
name: picture1

uri : media://picture1

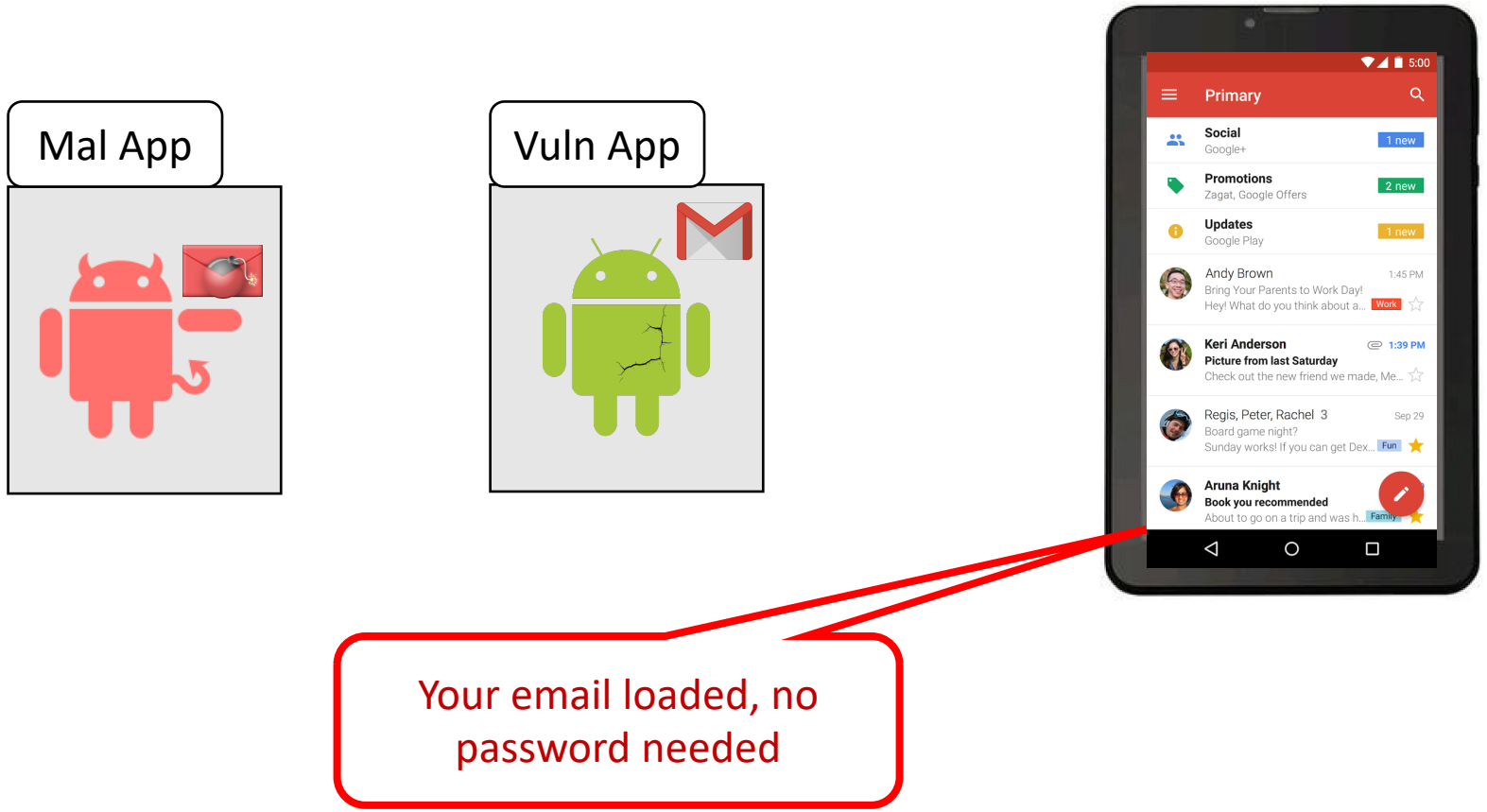
ICC-Based Android Vulnerability Types

- Inter-Process Denial of Service (IDOS)
- Fragment Injection (FI)
- Cross-Application Scripting (XAS)

Inter-Process Denial-of-Service Vulnerability



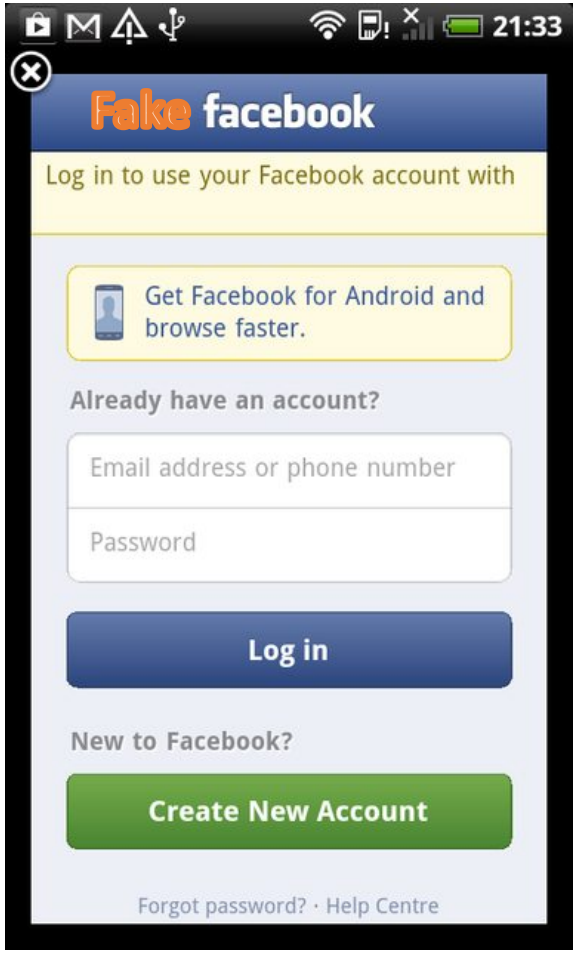
Fragment Injection Vulnerability



Cross-Application Scripting Vulnerability

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

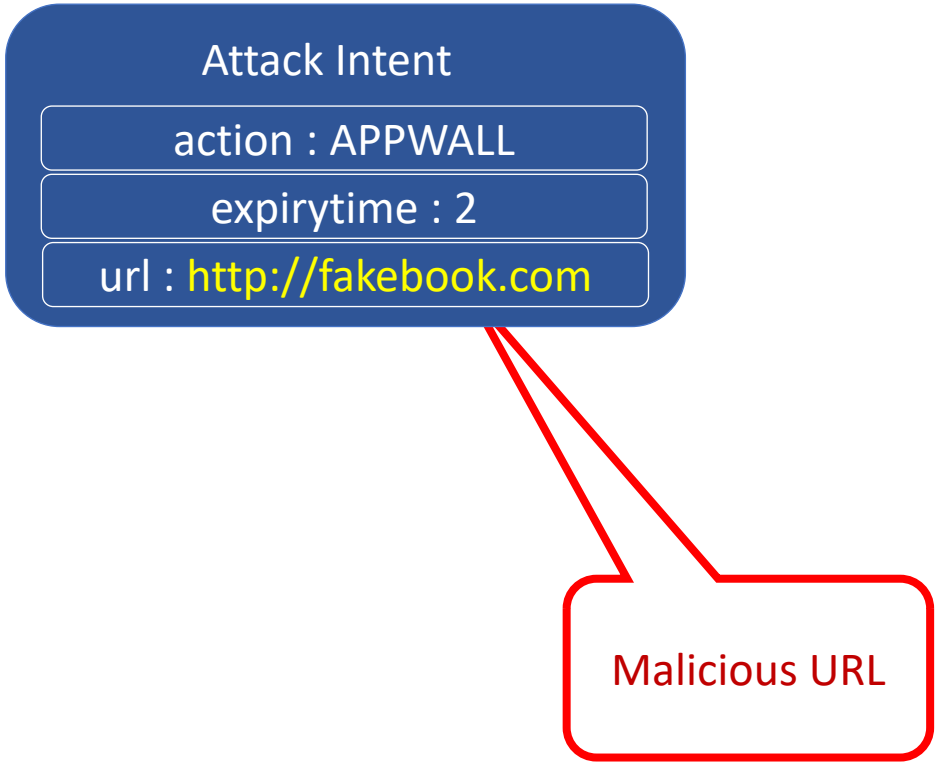
Statement vulnerable to cross-application scripting



Cross-Application Scripting Vulnerability

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

Statement vulnerable to cross-application scripting



Android Security Research on Inter-Component Communication

- Focus on vulnerabilities
- No research on **exploitability**

Vulnerable vs. Exploitable

- Vulnerability: a weakness in an application, system, device, or service that could lead to a failure to achieve security or privacy properties
- Exploitability: the extent to which a vulnerability can be successfully used by a malicious attacker
- Not all vulnerabilities are exploitable

Cross-Application Scripting Non-Exploitable

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        Bundle data = intent.getExtras();
        WebView webView = … ;
        String url = data.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

Attack Intent

action : APPWALL

expirytime : 2

url : http://fakebook.com

Statement vulnerable to cross-application scripting

Cross-Application Scripting Non-Exploitable

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        Bundle data = intent.getExtras();
        WebView webView = … ;
        String url = data.getStringExtra("url");
        if ( LocalDate.now().equals(“2015-01-23”) ) {
          webView.loadUrl( url );}
      }}}}
```

Non-exploitable vulnerability

Determining Exploitability

- Are the vulnerabilities that existing techniques detect actually exploitable?
- Manual and painstaking effort

- **Automatically** identify **exploitable** vulnerabilities
- Benefits
  - Reduce spurious vulnerabilities[1]
  - Prioritize bug fixes
  - Inputs to help fix security bugs
  - Stay ahead of zero-day vulnerabilities

1. Smith, Justin, et al. "Questions developers ask while diagnosing potential security vulnerabilities with static analysis." *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

Automatic Exploit Generation for Android

- No ICC-based Automatic Exploit Generation (AEG) for Android applications (apps)

- Challenges of ICC-based AEG for Android apps
  - Highly accurate model of Android ICC
  - Automatic assessment of vulnerability exploitation

Solution: LetterBomb

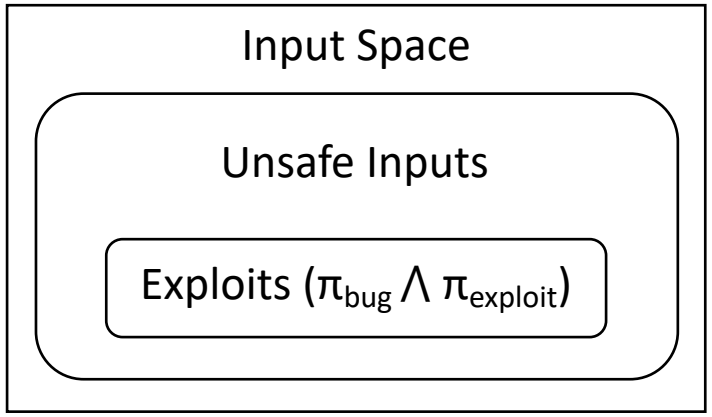
- Goal: Automatic generation of inter-component communication (ICC) exploits for Android apps
- Combined static and dynamic analysis consisting of the following:
  - **Path-sensitive backwards symbolic execution** along Android ICC interface
  - Test **input generation** to produce an Intent
  - Production of automated software test **oracles**

Automatic Exploit Generation

- Goal: automatically generate an input that satisfies the equation

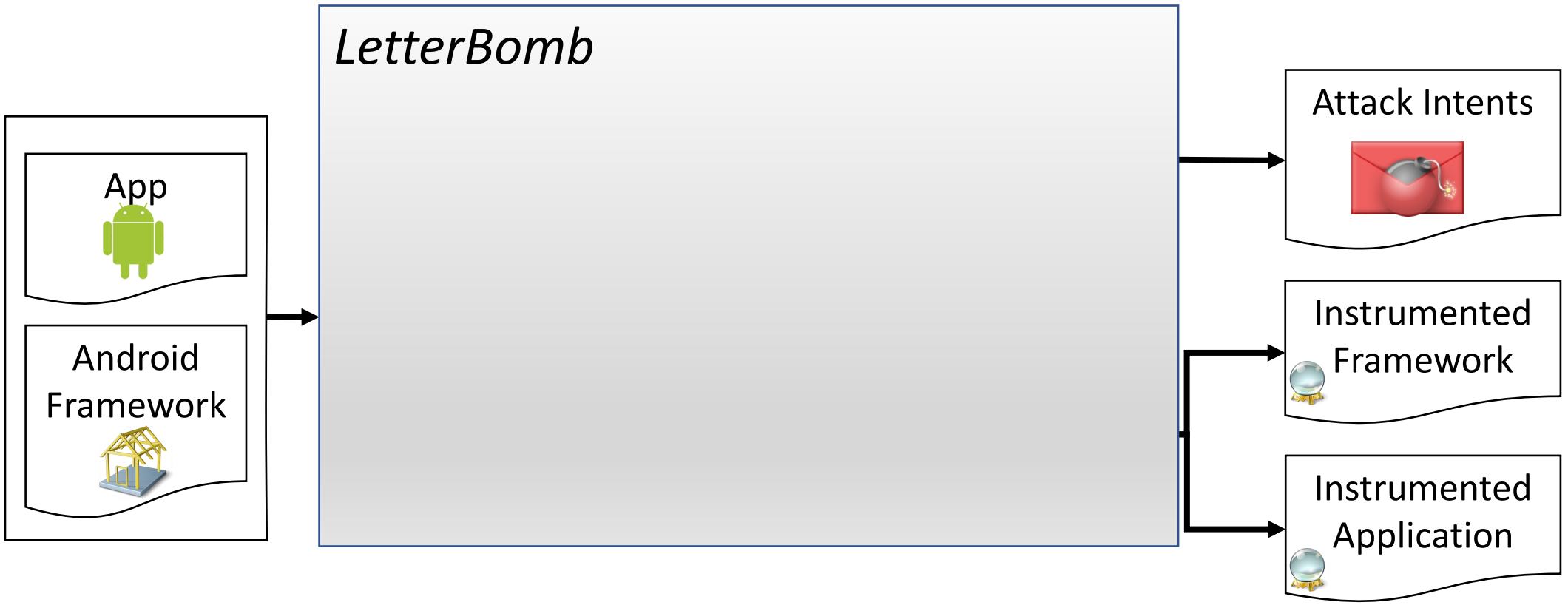
$$\pi_{bug} \wedge \pi_{exploit}$$

- π_{bug} is an unsafe path predicate
- $\pi_{exploit}$ is an exploit predicate
  - Attacker's logic
  - Successful exploitation

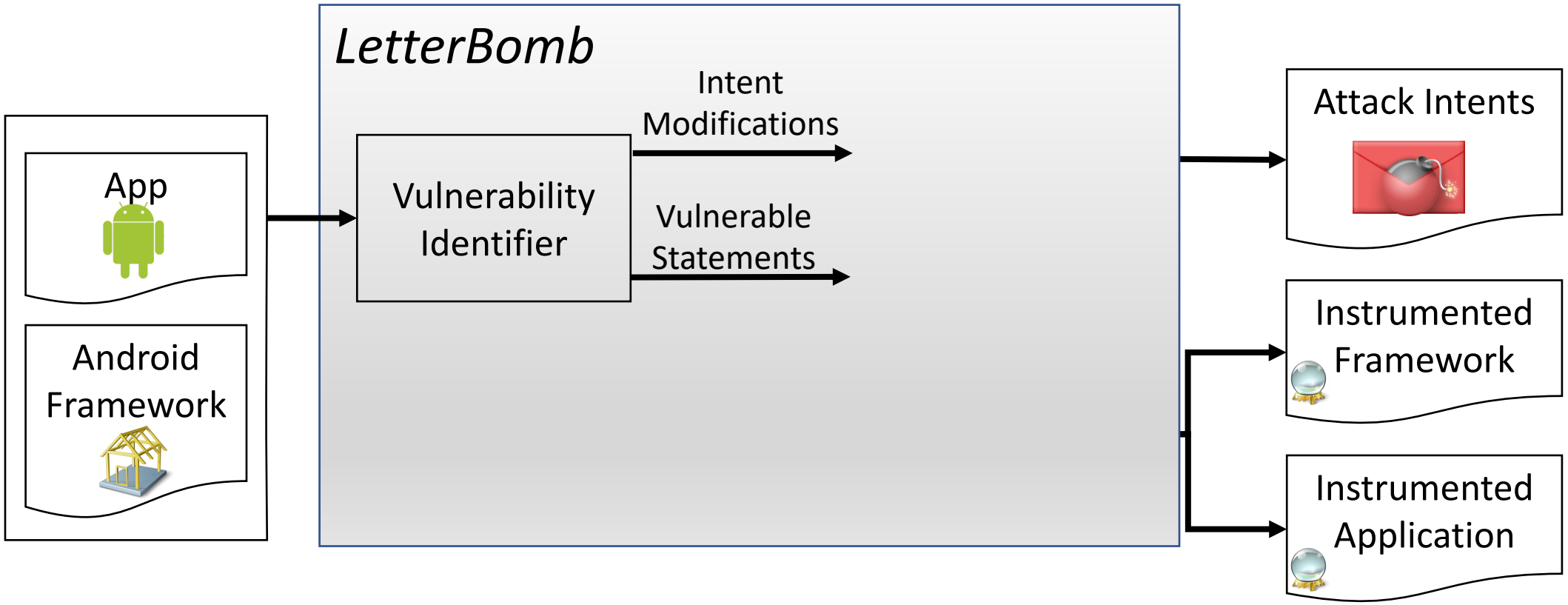


Thanassis, HBLT Avgerinos, Cha Sang Kil, and Brumley David. "Aeg: Automatic exploit generation." *ser. Network and Distributed System Security Symposium*. 2011.

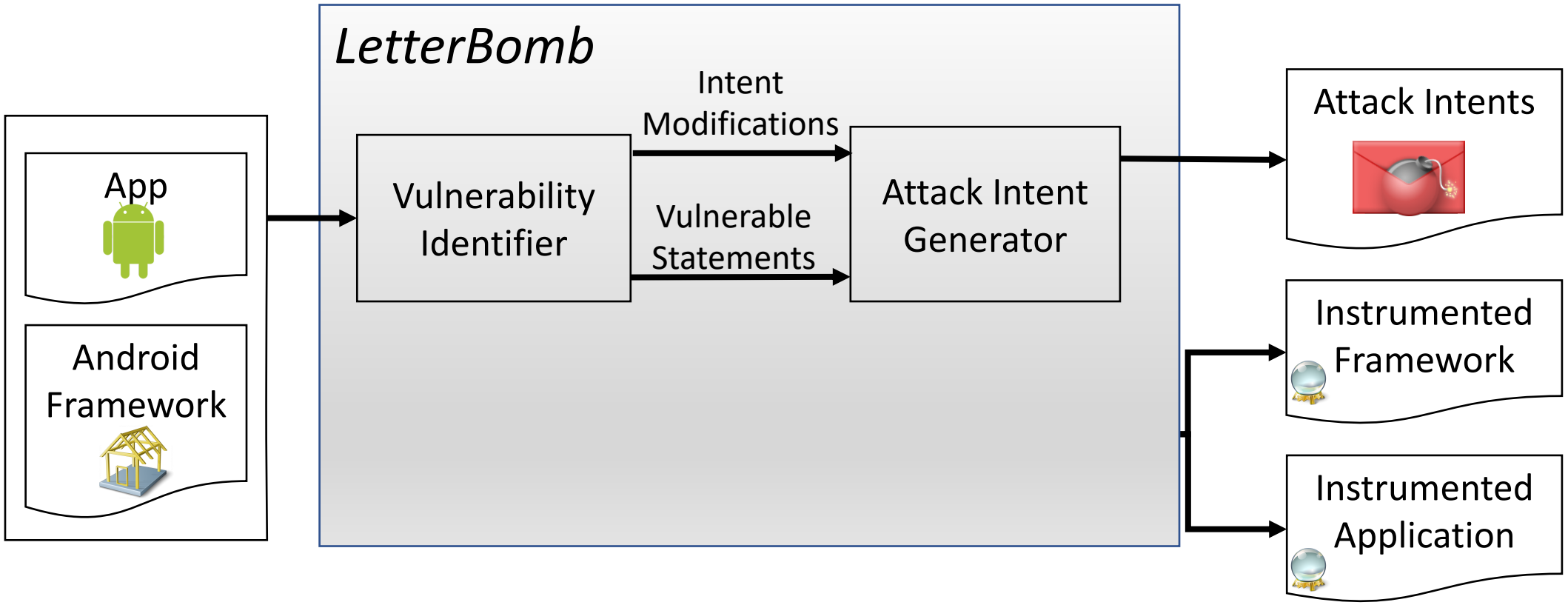
LetterBomb Overview



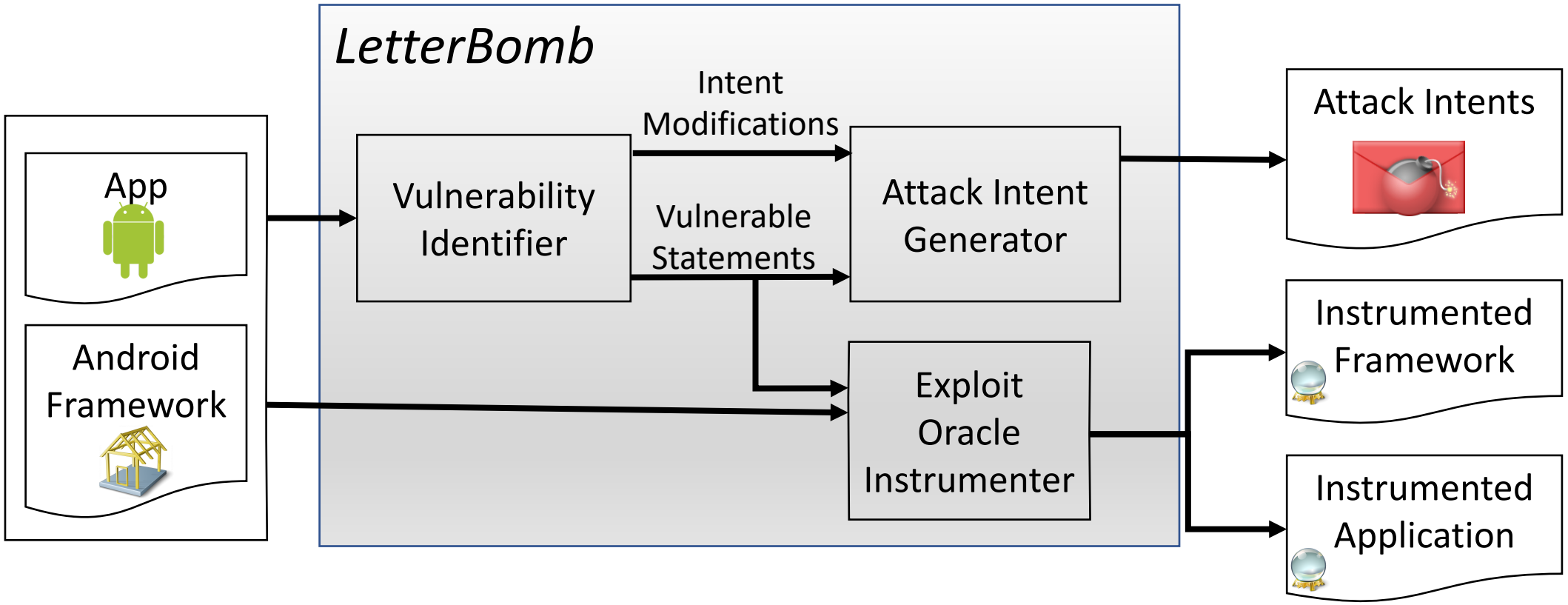
LetterBomb Overview



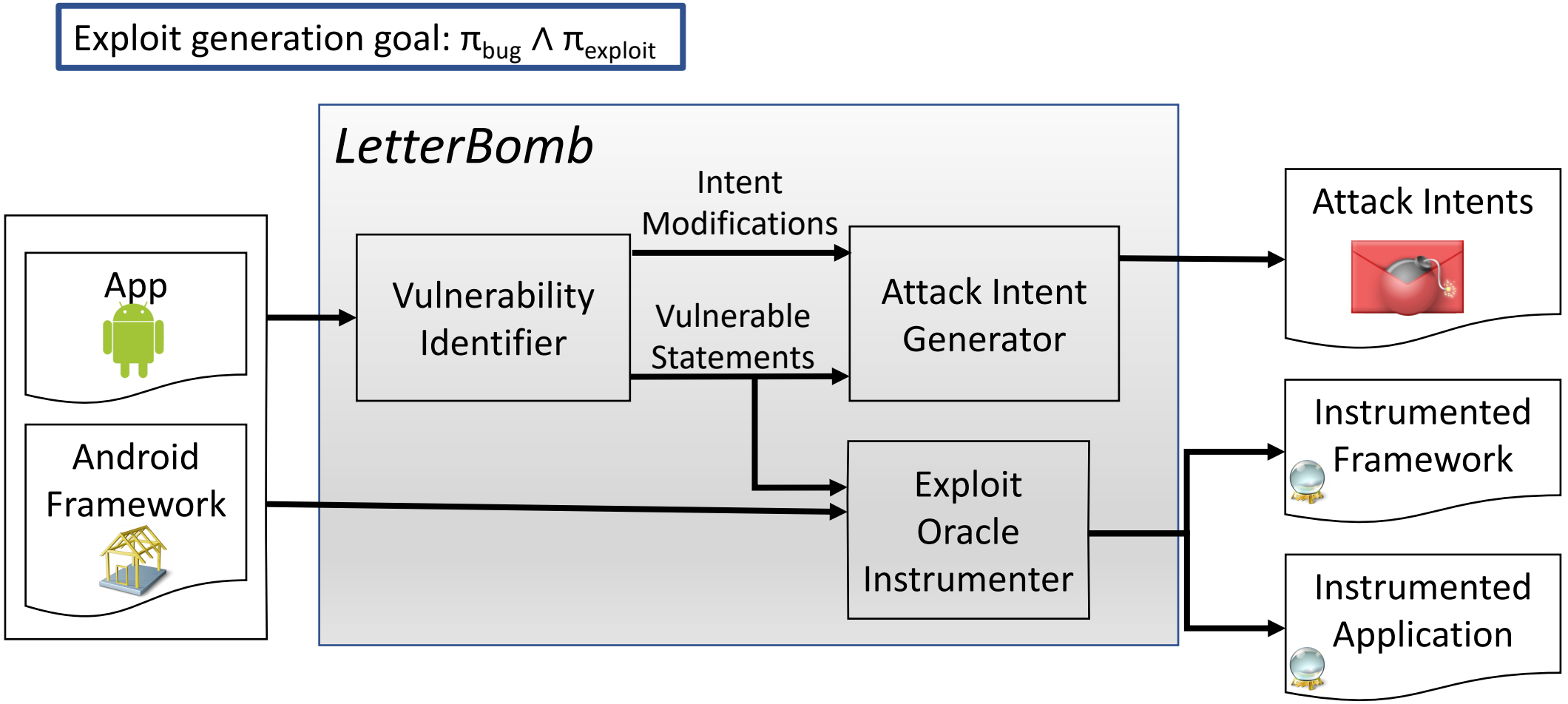
LetterBomb Overview



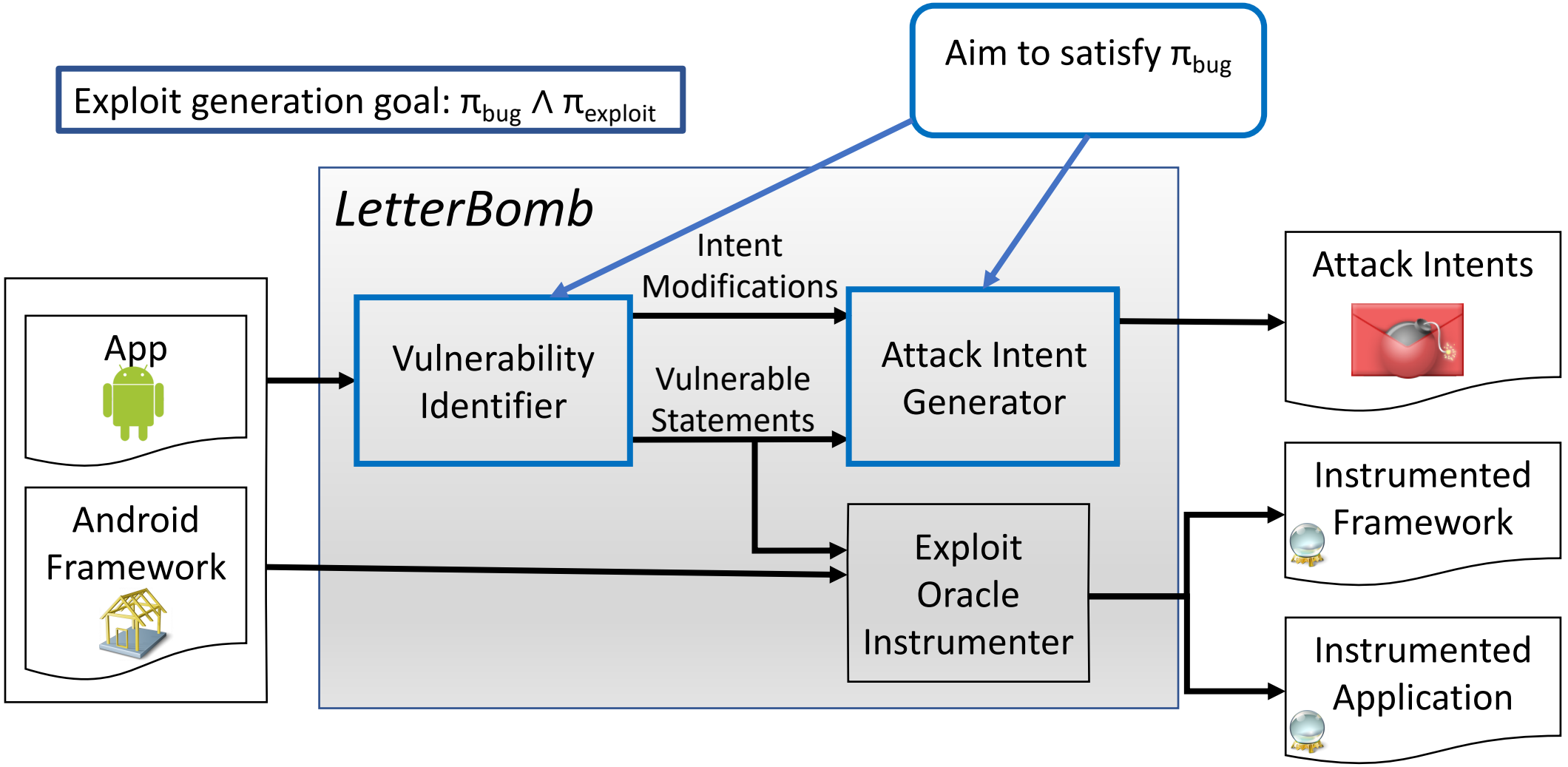
LetterBomb Overview



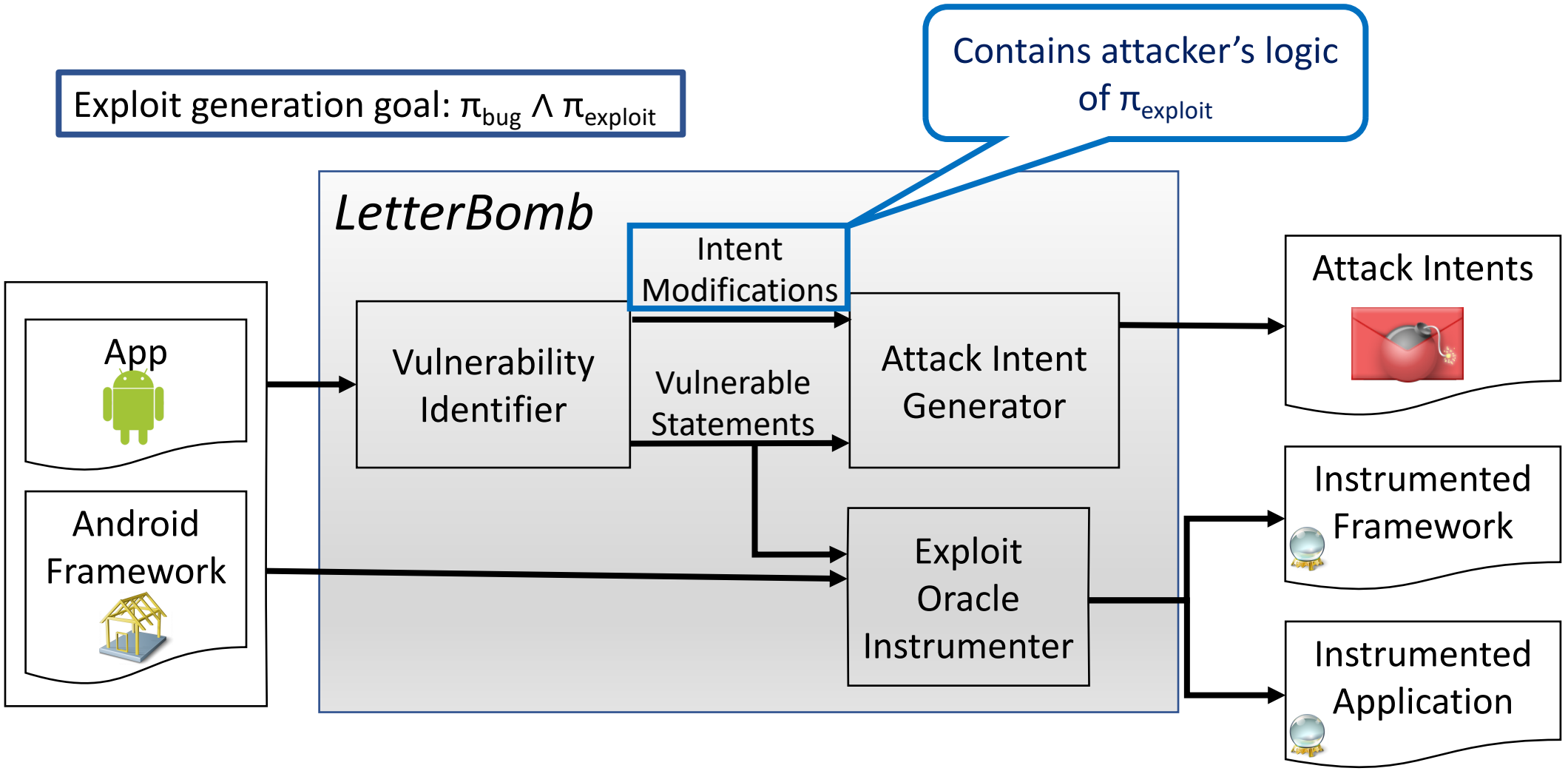
LetterBomb Overview



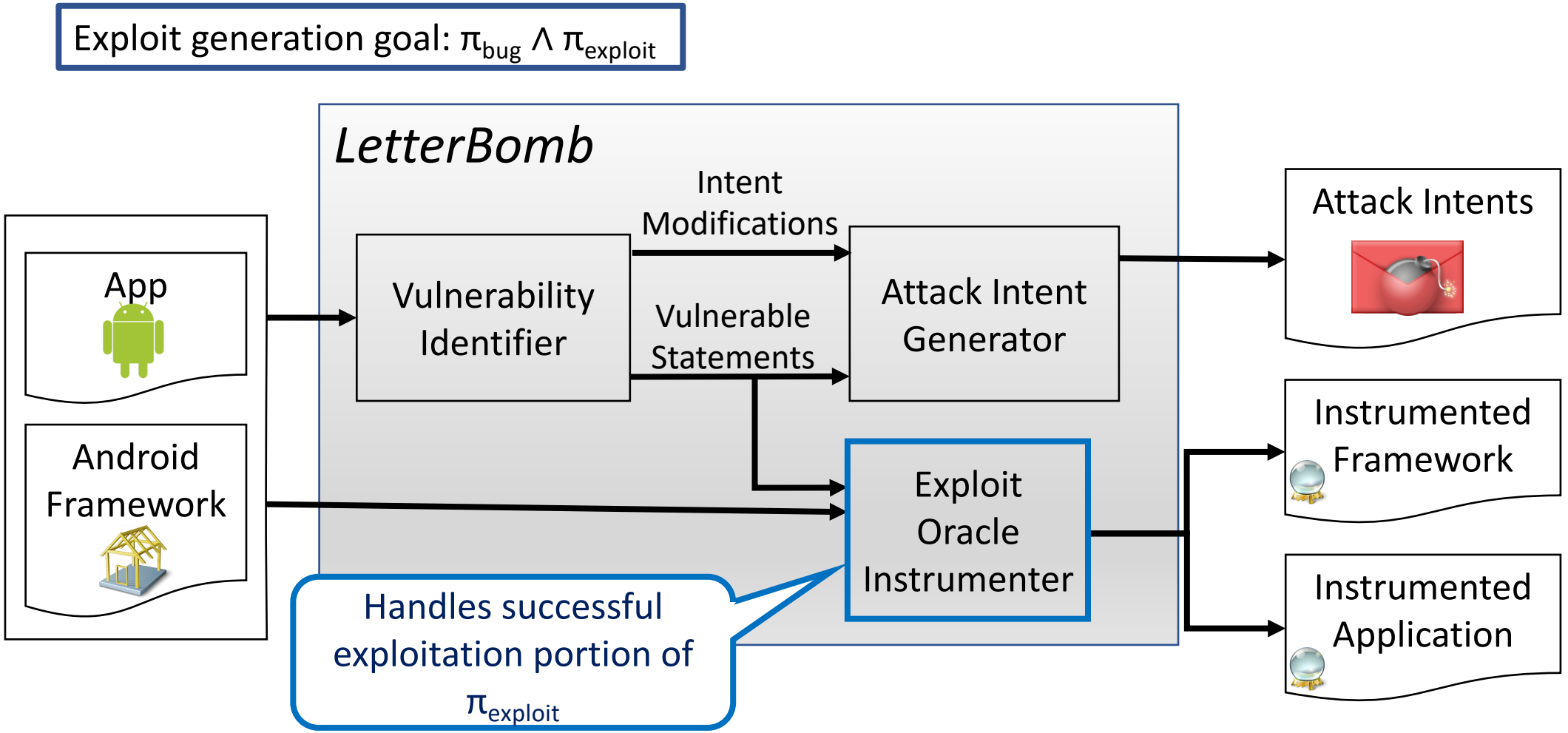
LetterBomb Overview



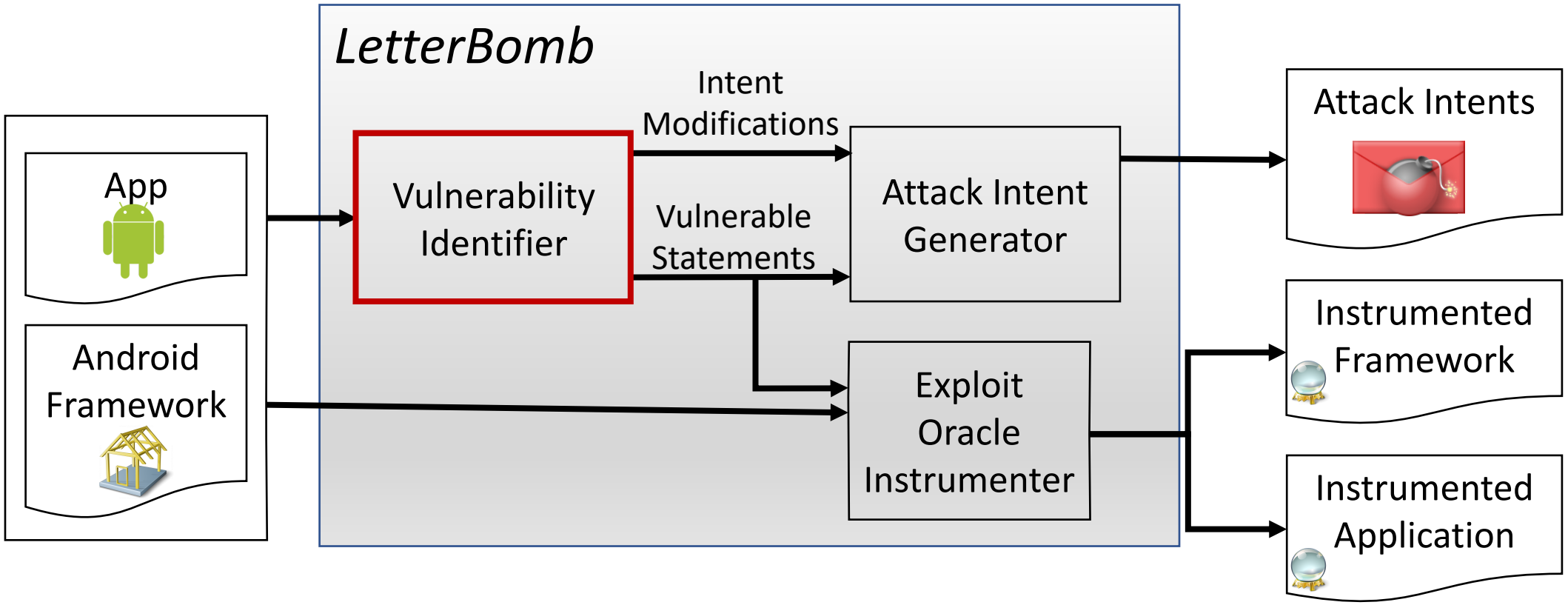
LetterBomb Overview



LetterBomb Overview



LetterBomb Overview



Selected Android Vulnerability Types

- Cross-Application Scripting (XAS)
- Inter-Process Denial of Service (IDOS)
- Fragment Injection (FI)

Vulnerability Identification

- Goal: Identify vulnerable statements
- Conservative analysis to be made more precise by later steps
- XAS Identification
  - Identify final injection point of the malicious URL
  - Track data passed to injection point to identify if it is from a received Intent

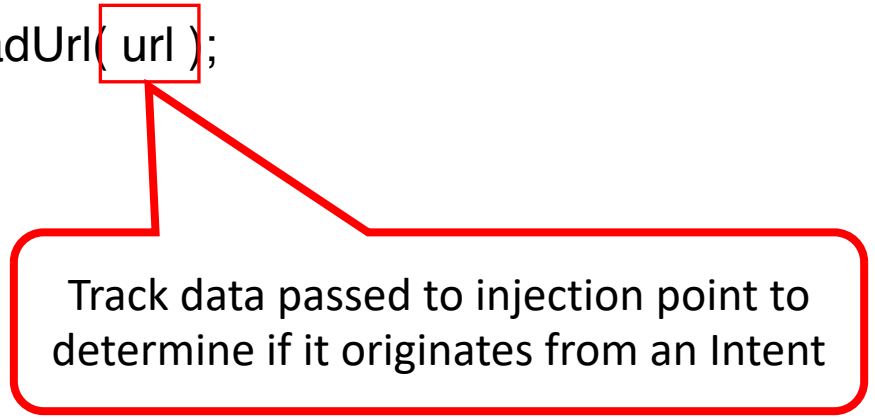
XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
    }}}}
```

Identify injection point for XAS

XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
    }}}}
```



XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

Identified extraction from Intent

XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
   Intent intent = getIntent();
   String action = intent.getAction();
   if ("APPWALL". equals(action)) {
    if (intent.getIntExtra ("expirytime" ,0) > 0) {
     WebView webView = … ;
     String url = intent.getStringExtra("url");
     webView.loadUrl( url );
   }}}}
```

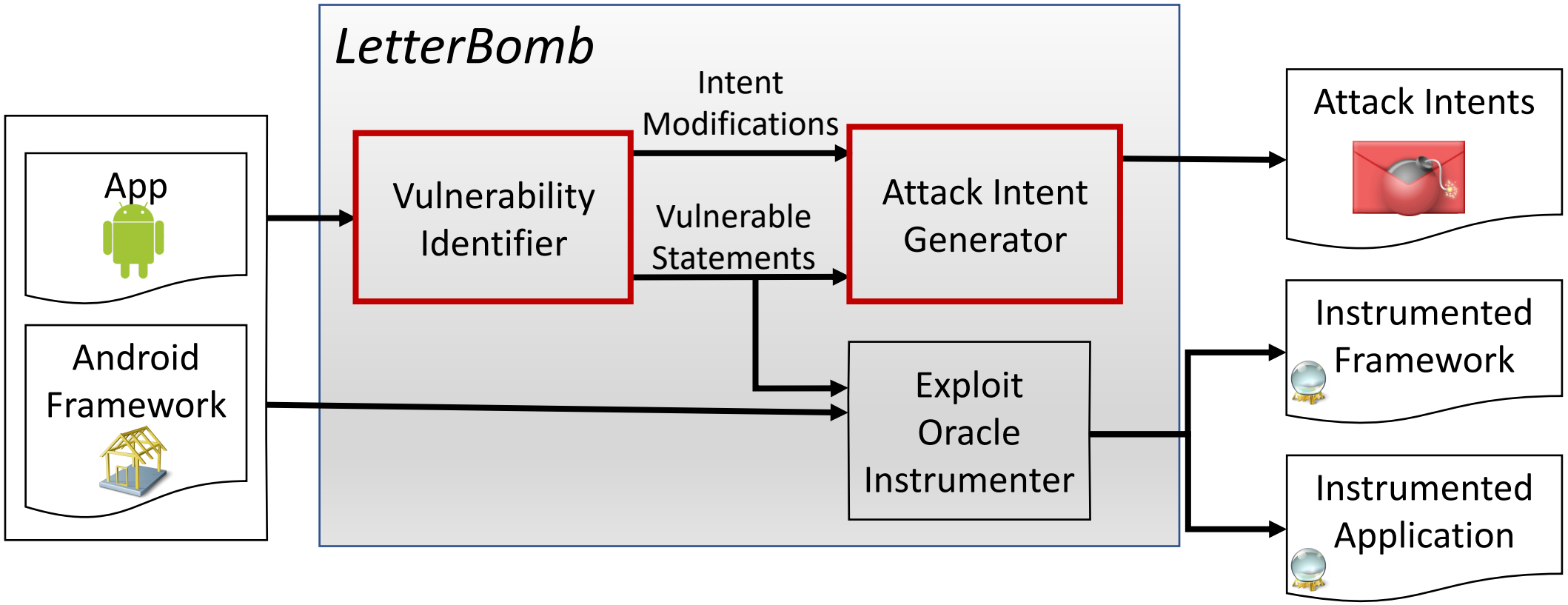
Verify that Intent was received externally

XAS Vulnerability Identification

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
    }}}}
```

Statement vulnerable to cross-application scripting

LetterBomb Overview



Attack Intent Generation

- Construct an Intent to attack an app
- Given a vulnerable statement,
  - Perform a path-sensitive backward symbolic execution starting at vulnerable statement
  - Identify attributes of Intent needed to execute the vulnerable program path
  - Supply logic of attack by modifying an Intent

XAS Attack Intent Generation Example

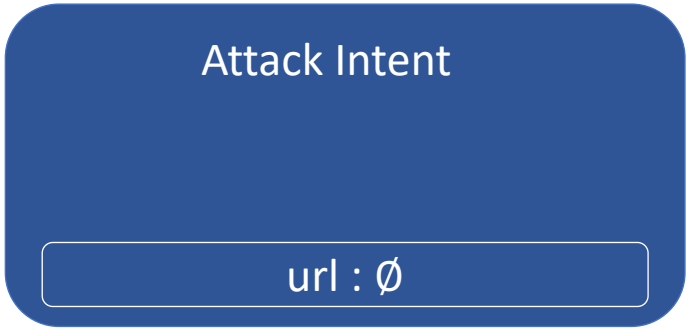
```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

Attack Intent

Statement vulnerable to cross-application scripting

XAS Attack Intent Generation Example

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

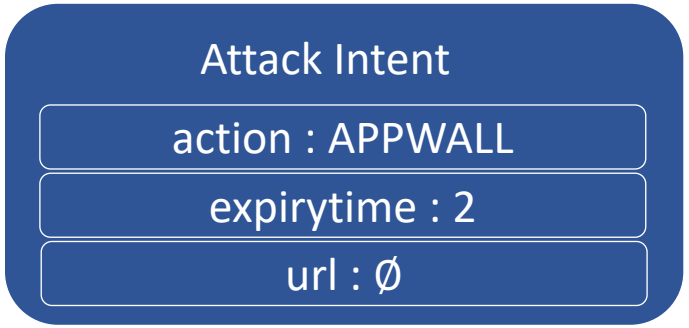


XAS Attack Intent Generation Example

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
     if (intent.getIntExtra ("expirytime" ,0) > 0) {
       WebView webView = … ;
       String url = intent.getStringExtra("url");
       webView.loadUrl( url );
     }}}}
```

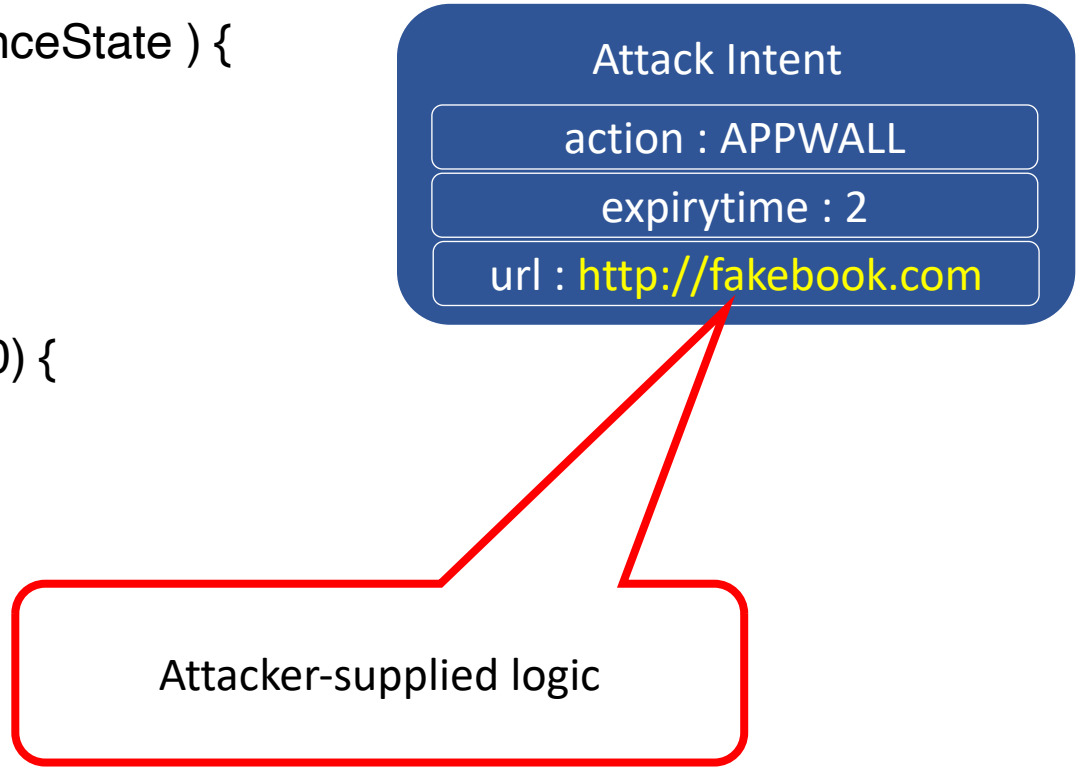
XAS Attack Intent Generation Example

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

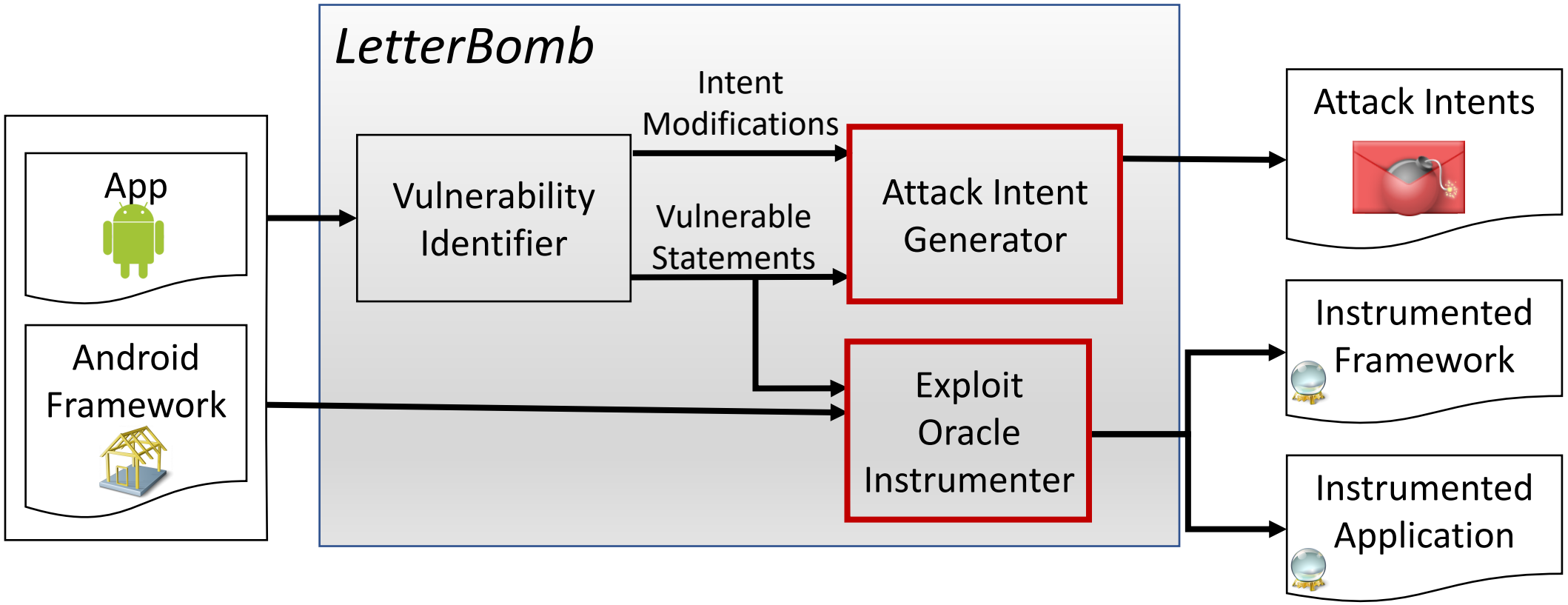


XAS Attack Intent Generation Example

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```



LetterBomb Overview



Exploit Oracle Instrumentation

- One-time specification or construction of oracle per vulnerability type
- Oracles are automated and reusable across all apps

- Two parts of customized oracle
  - Instrumentation of the vulnerable application or the Android framework
  - Post-processing of logged statements to determine if exploit was successful

XAS Exploit Oracle Instrumentation

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
      }}}}
```

XAS Exploit Oracle Instrumentation

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
        webView.setWebViewClient (new WebViewClient () {
          public void onPageFinished(WebView view ,String url ) {
            Log.i("Instrument "," loaded url:" + url );
            super.onPageFinished(view , url);}}) ;
      }}}}
```

Attack Intent

action : APPWALL

expirytime : 2

url : http://fakebook.com

Add instrumentation code

XAS Exploit Oracle Instrumentation

```
public class AdsActivity extends Activity {
  public void onCreate ( Bundle savedInstanceState ) {
    Intent intent = getIntent();
    String action = intent.getAction();
    if ("APPWALL". equals(action)) {
      if (intent.getIntExtra ("expirytime" ,0) > 0) {
        WebView webView = … ;
        String url = intent.getStringExtra("url");
        webView.loadUrl( url );
        webView.setWebViewClient (new WebViewClient () {
          public void onPageFinished(WebView view ,String url ) {
            Log.i("Instrument "," loaded url:" + url );
            super.onPageFinished(view , url);}}) ;
      }}}}
```

Attack Intent

action : APPWALL

expirytime : 2

url : http://fakebook.com

Assert that the log contains statement
“Instrument: loaded url: http://fakebook.com”

Attack Intent Generation: Symbolic Execution

- Backwards Path-Sensitive Static Symbolic Execution
  - Summary-based analysis starting from vulnerable statements
  - Context-sensitive, flow-sensitive, object-sensitive, and inter-procedural
  - Reverse topological order analysis over call graph
- Incremental Call-Graph Construction
  - Multiple entry points of Android lifecycle
- Output are summaries of analyzed method
  - $\Sigma : M \rightarrow$ targetExprs
  - Σ is a map from methods M to expressions describing Intents, i.e., targetExprs

Attack Intent Generation: SMT Expressions

- Construct path conditions as SMT expressions and provide to SMT solver to check for feasibility and to generate Intent data
- Expressed using SMT-Lib, a language for SMT solvers
- Intent model for SMT expressions
  - Extra data
  - Actions
  - Categories
  - String and Object Comparisons

Attack Intent Generation: Extra Data

Extra data: key-value pairs in Intents

e1) (assert (= (containsKey r_e r_k) true))

e2) (assert (= (fromIntent r_e) i))

Attack Intent Generation: Extra Data

Extra data: key-value pairs in Intents

e3) (assert (= (containsKey r_{et} “expirytime”) true))

e4) (assert (= (fromIntent r_{et}) r_{intent}))

Attack Intent Generation: Intent Actions

```
public class AdsActivity extends Activity {
 public void onCreate(Bundle savedInstanceState) {
  Intent intent = getIntent();
  String action = intent.getAction();
  if (action.equals("BANNER"))
    doBannerAd(intent);
  else if ("APPWALL".equals(action)) {
    if (intent.getIntExtra("expirytime",0)>0 && !intent.hasCategory("android.intent.category.DEFAULT"))
      doAppWallAd(intent); }}
```

Expressions for actions:

(assert (= r_a “APPWALL”))

(assert (not (= r_a “BANNER”)))

(assert (= (getAction i) r_a))

(assert (= (fromIntent r_a) i))

Attack Intent Generation: Intent Actions

```
public class AdsActivity extends Activity {
 public void onCreate(Bundle savedInstanceState) {
  Intent intent = getIntent();
  String action = intent.getAction();
  if (action.equals("BANNER"))
    doBannerAd(intent);
  else if ("APPWALL".equals(action)) {
    if (intent.getIntExtra("expirytime",0)>0 && !intent.hasCategory("android.intent.category.DEFAULT"))
      doAppWallAd(intent); }}
```

Existence:

(assert (exists((*idx* Int))(= (select $cats_{r_h}$ *idx*) r_c)))

Non-existence:

(assert (forall((*idx* Int))(not (= (select $cats_{r_h}$ *idx*) r_c))))

Attack Intent Generation: Intent Actions

```
public class AdsActivity extends Activity {
 public void onCreate(Bundle savedInstanceState) {
  Intent intent = getIntent();
  String action = intent.getAction();
  if (action.equals("BANNER"))
    doBannerAd(intent);
  else if ("APPWALL".equals(action)) {
    if (intent.getIntExtra("expirytime",0)>0 && !intent.hasCategory("android.intent.category.DEFAULT"))
      doAppWallAd(intent); }}
```

Existence:

(assert (exists((*idx* Int))(= (select $cats_{r_h}$ *idx*) r_c)))

Non-existence:

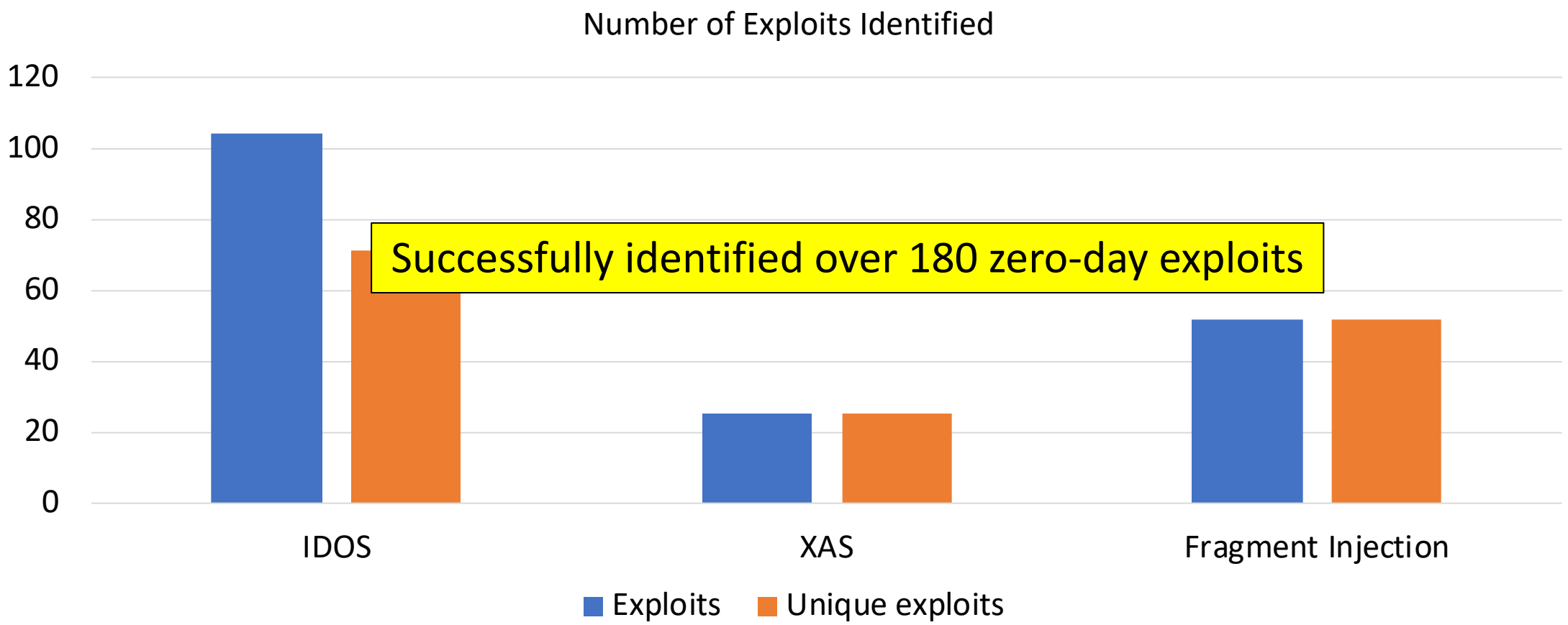
(assert (forall((*idx* Int))(not (= (select $cats_{r_h}$ *idx*) *"android.intent.category.DEFAULT"*)))))))

Empirical Evaluation

- Exploitability Detection
- Spurious Vulnerability Reduction
- Vulnerability Detection Comparison
- Runtime Efficiency Comparison
- Intent Accuracy

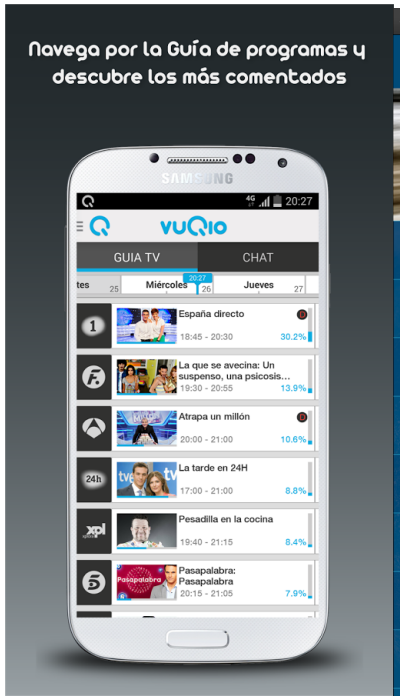
Exploitability Detection - Results

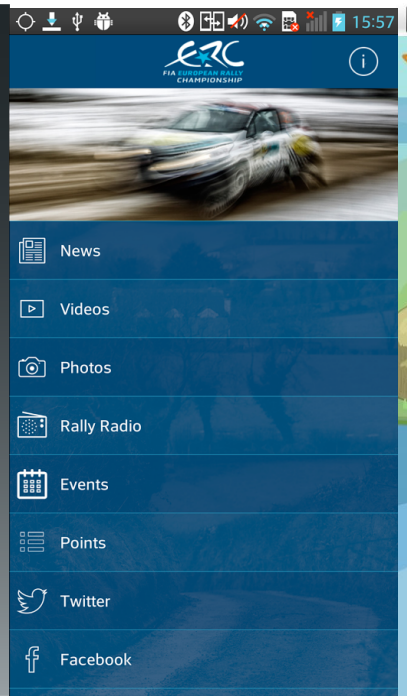
- 10,000 apps randomly selected from Google Play

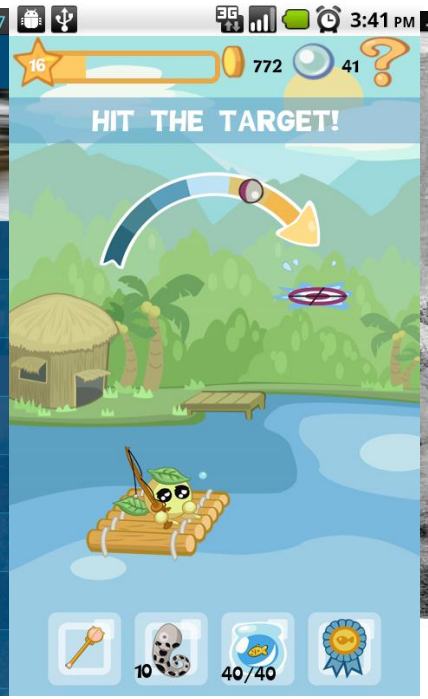


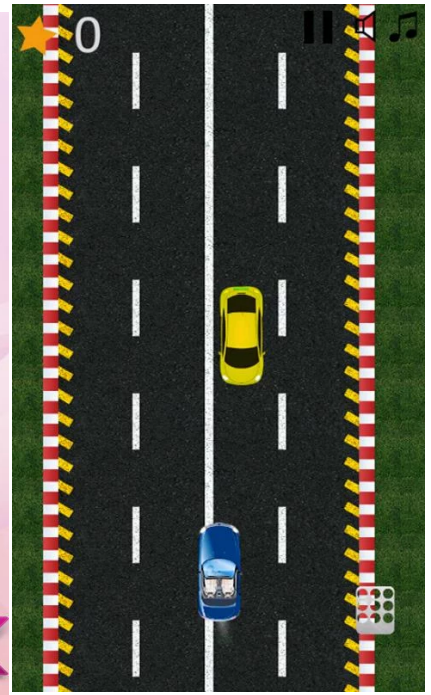
Some Popular Exploited Apps

IDOS **FI** **IDOS** **XAS** **IDOS** **XAS**



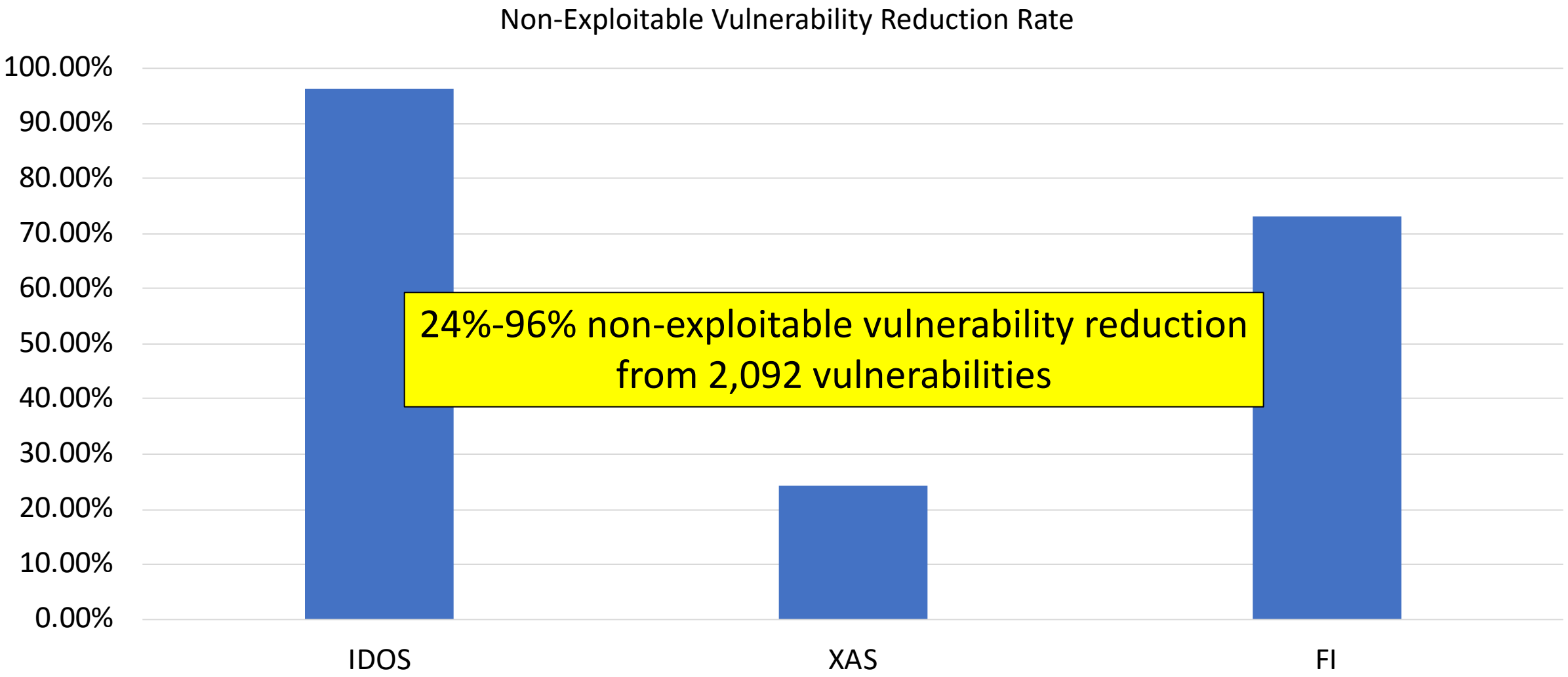




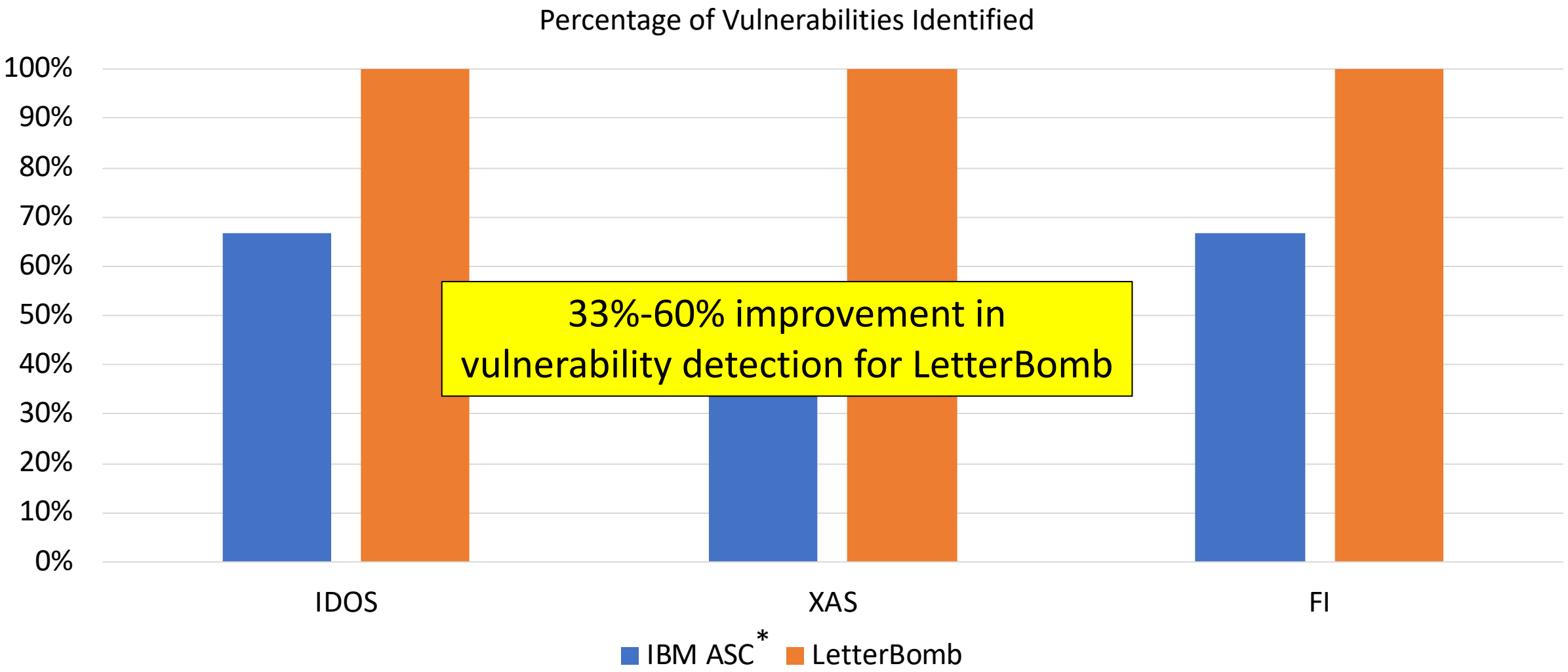


A wide variety of apps exploited with as many as 10,000,000 downloads

Non-Exploitable Vulnerability Reduction - Results

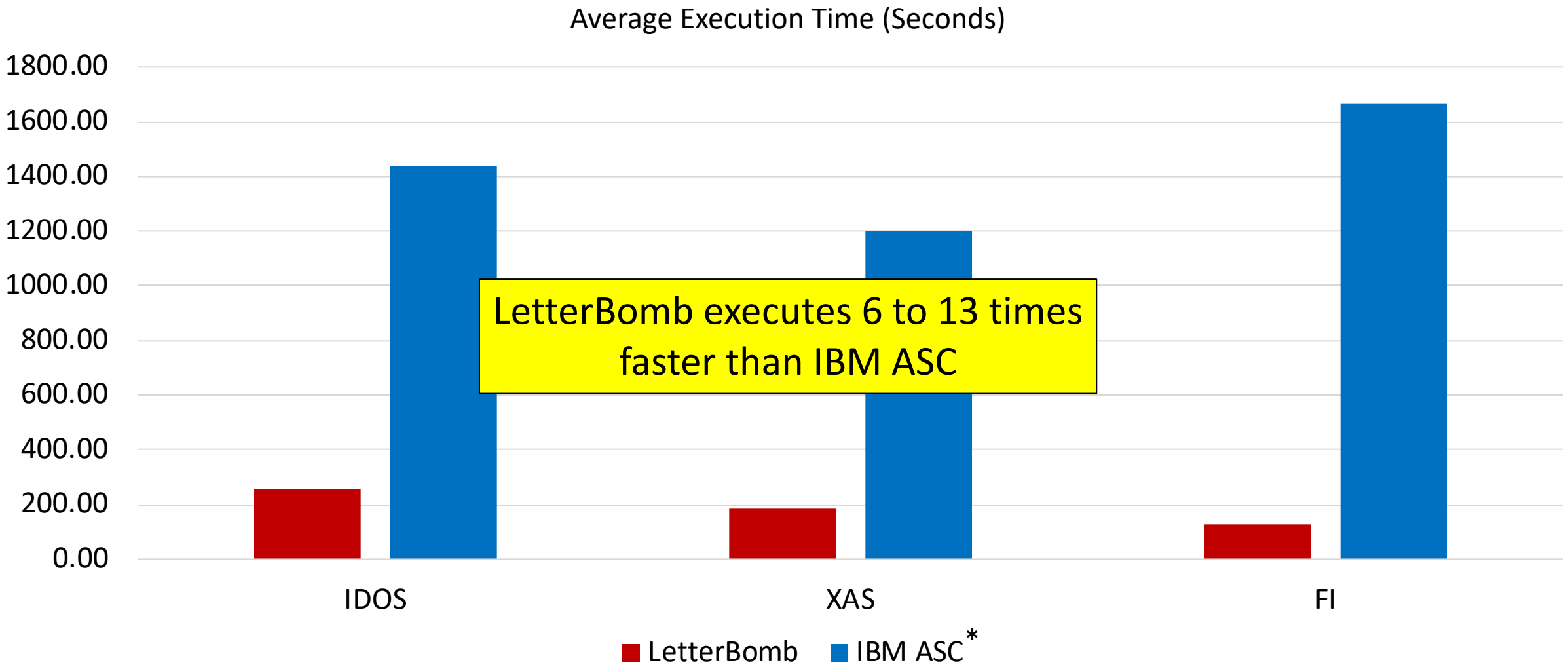


Vulnerability Detection Comparison - Results



*Hay, Roee, Omer Tripp, and Marco Pistoia. "Dynamic detection of inter-application communication vulnerabilities in Android." *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015.

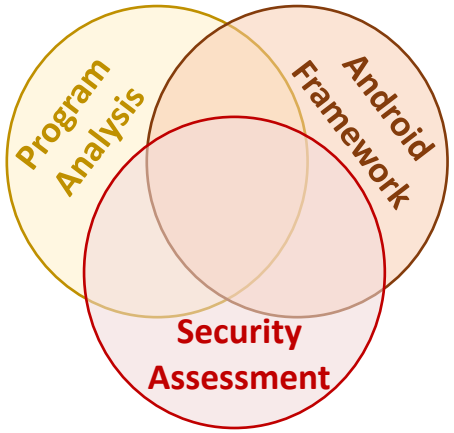
Efficiency Comparison - Results



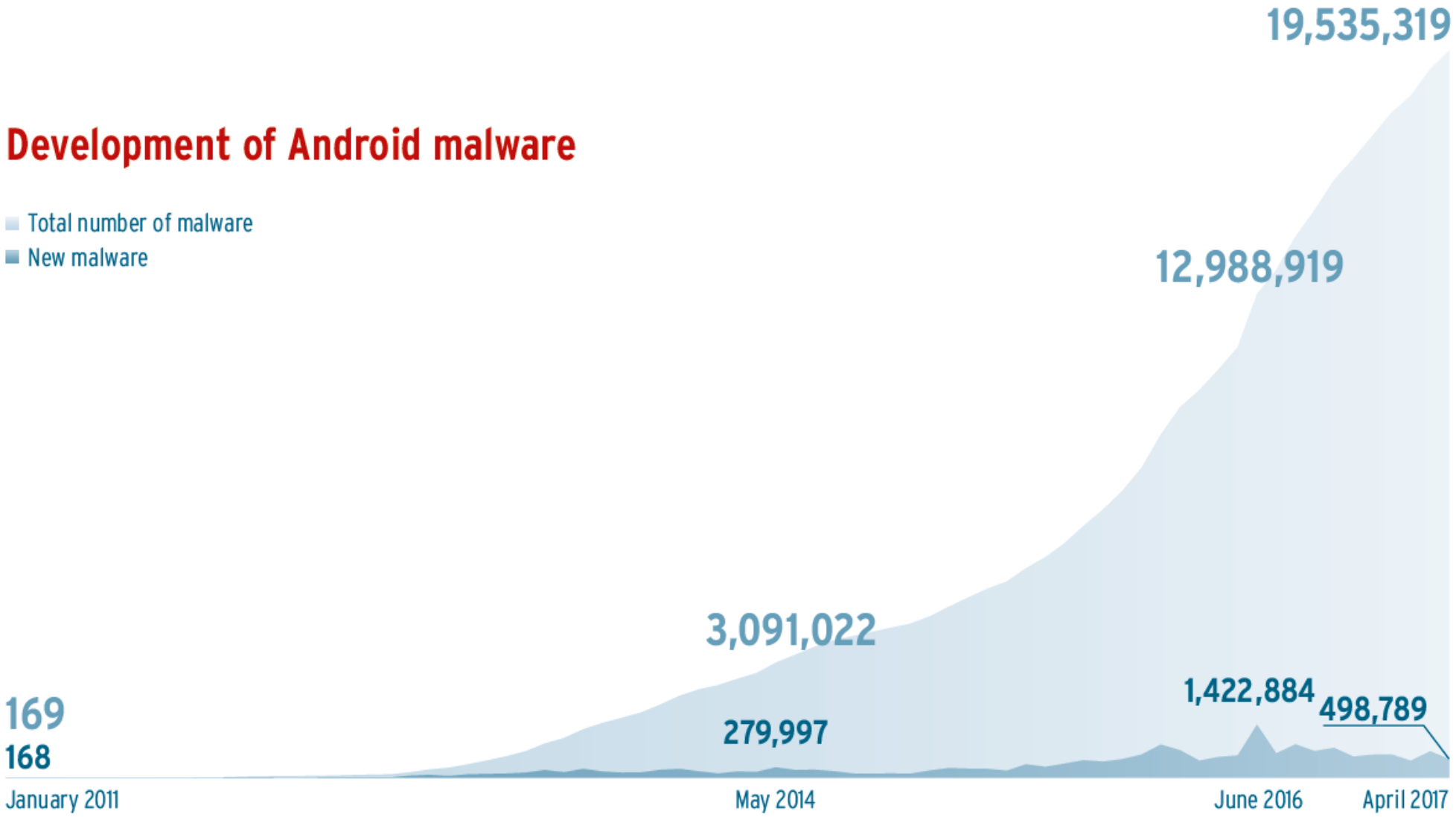
*Hay, Roee, Omer Tripp, and Marco Pistoia. "Dynamic detection of inter-application communication vulnerabilities in Android." *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015.

Structure of Today's Lecture

- A Taxonomy of Program Analysis Techniques for Security Assessment of Android Software
- Automatic Exploit Generation of Android Apps
- Lightweight, Obfuscation-Resilient Android Malware Classification
- Self-Protection of Android Systems from Inter-Component Communication Attacks



Explosive Growth of Android Malware



Source:

AV-TEST
The Independent IT-Security Institute
Magdeburg Germany

Mobile Malware Using Unconventional Code

AndroidOS/FakeInst Sends SMS uses reflection

- Bootkit with malicious native code
- Over 350,000 devices

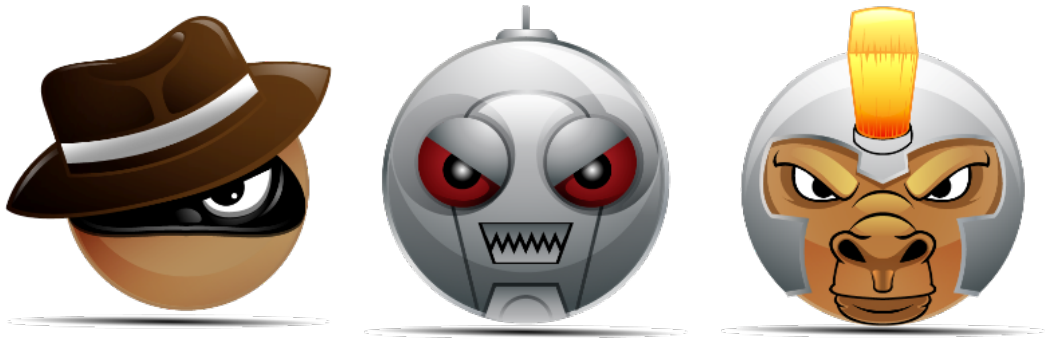
- Botnet operating from native code
- 50,000-100,000 downloads

2012 — 2014 — 2016 — 2017

Sources: total defense, 360 www.360.cn, Check Point SOFTWARE TECHNOLOGIES LTD.

Countermeasures

- Detection and removal



Countermeasures

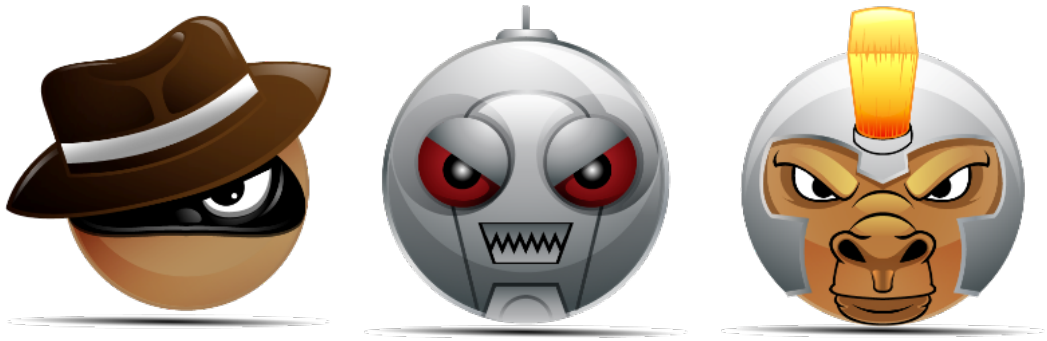
- Detection and removal is not enough

Countermeasures

- Detection and removal is not enough—identify **families**

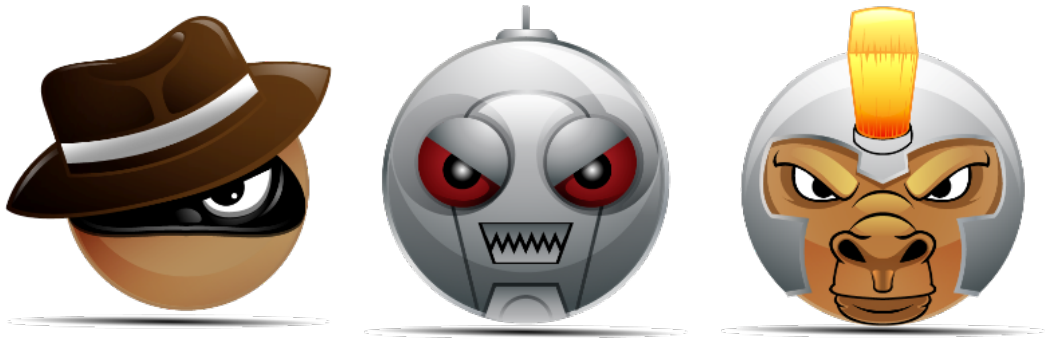
Countermeasures

- Detection and removal is not enough—identify **families**
- Malware likes to **hide,** causing a 20% average decrease and up to a 91% decrease for 60 anti-malware products **[ICSE 2018]**



Countermeasures

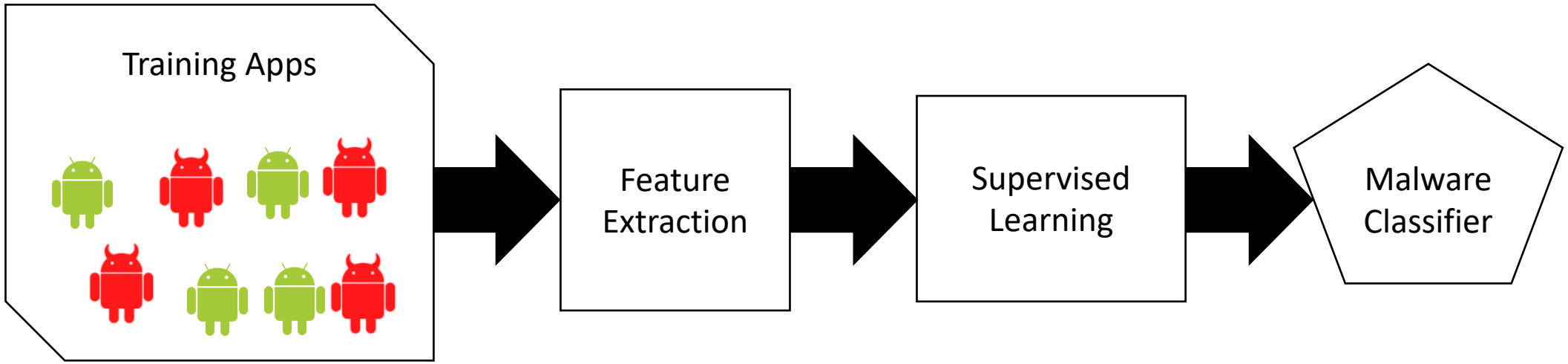
- Detection and removal is not enough—identify **families**
- Malware likes to **hide,** causing a 20% average decrease and up to a 91% decrease for 60 anti-malware products **[ICSE 2018]**
- Catch them **fast**



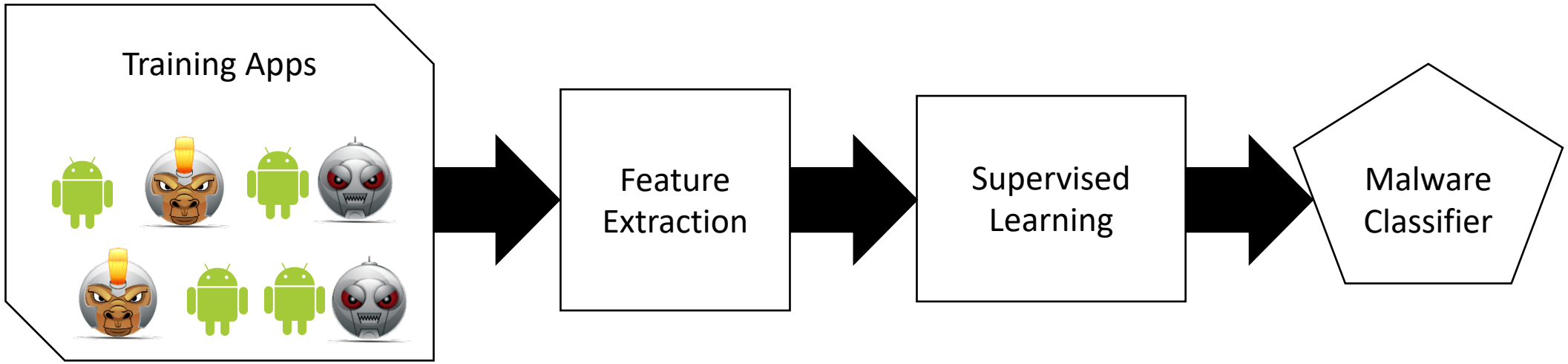
RevealDroid

- A machine learning-based approach for malware detection and family identification
  - Accurate
  - Highly efficient
  - Obfuscation-resilient
- Analysis of conventional and unconventional code
  - Android platform
  - Reflection
  - Native code

Classifier Construction for Malware Detection and Family Identification



Classifier Construction for Malware Detection and Family Identification



Feature Selection

	Perm	Comp	IFilters	Flows	MAPI	PAPI	IActions	Reflection	Native
Accuracy	X	X	X						
Efficiency				X					
Obfuscation		X	X	X			X		

Automated feature selection reducing over a million features to 1,054 features

Feature Examples: Method and Package API

- Numbers of Android API methods invoked by app
  - android.telephony
    - TelephonyManager.getCellLocation()
    - CellIdentityLte.getCi()

	telephony	location	sqlite	Fam
mal1	8	0	2	jSMSHider
mal2	0	12	0	Geinimi
mal3	2	0	7	BaseBridge

Feature Examples: Reflective Calls

- Apps may dynamically load or invoke libraries/classes through reflection
  - Used frequently to obfuscate malicious behavior
  - Resolution levels

```
1   ClassLoader cl = MyClass.getClassLoader();
2   try { Class c = cl.loadClass("MyActivity");
3         ...
4         Method m = c.getMethod("onPause",...);
5         ...
6         m.invoke(...); }
7   catch { ... }
```

Reflective Call Features: Resolution Levels

- Resolution levels
  - Fully resolved (e.g., MyActivity.onPause(…))
  - Partially resolved (e.g., onPause(…))
  - Unresolved
- Counts of resolution levels

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("MyActivity");
3        ...
4        Method m = c.getMethod("onPause",...);
5        ...
6        m.invoke(...); }
7  catch { ... }
```

Feature Examples: Native Calls

- Apps can make system calls and calls to native binaries
  - Analysis of native binaries requires disassembly of ELF files
  - Procedure Linkage Table (PLT) used to determine the address of external functions not known at runtime

```
1    99ec: e59d0010 ldr r0, [sp, #16]
2    99f0: e59f13c0 ldr r1, [pc, #960]
3    99f4: ebfffc3e bl  8af4 <chmod@plt>
```

Code segment where *chmod* is invoked

Statistical Feature Selection: Design

- Benefits of feature selection
  - Faster classifier creation
  - Reduced training and testing time
  - Reduced possibility of overfitting
- Stochastic gradient descent classifier for feature selection
  - Supports incremental learning and memory constraints
- Reduced over a million features to 1,054

Statistical Feature Selection: Details

- 1,054 features in total
  - 595 method-level and package-level features
  - 454 native call features
  - 5 reflection features
- Method-level and package-level features
  - Security-sensitive
  - UI-oriented features
- Native features
  - For exploits and security-sensitive functionality
  - For benign apps, graphics rendering libraries and image manipulation

Labeling and Classifier Selection

- Classifier for detection
  - 2-way classifier with labels “benign” or “malicious”
  - Support Vector Machine (SVM)

- Classifier for family identification
  - *n*-way classifier where *n = the number of families*
  - Classification and Regression Trees (CART)

Experimental Setup

- Prototype built using open-source software
  - Java and Python

- Over 24,679 benign and 30,203 malicious apps
  - Collected from Malware Genome, Drebin, VirusShare, and VirusTotal
  - Purely benign apps from GooglePlay

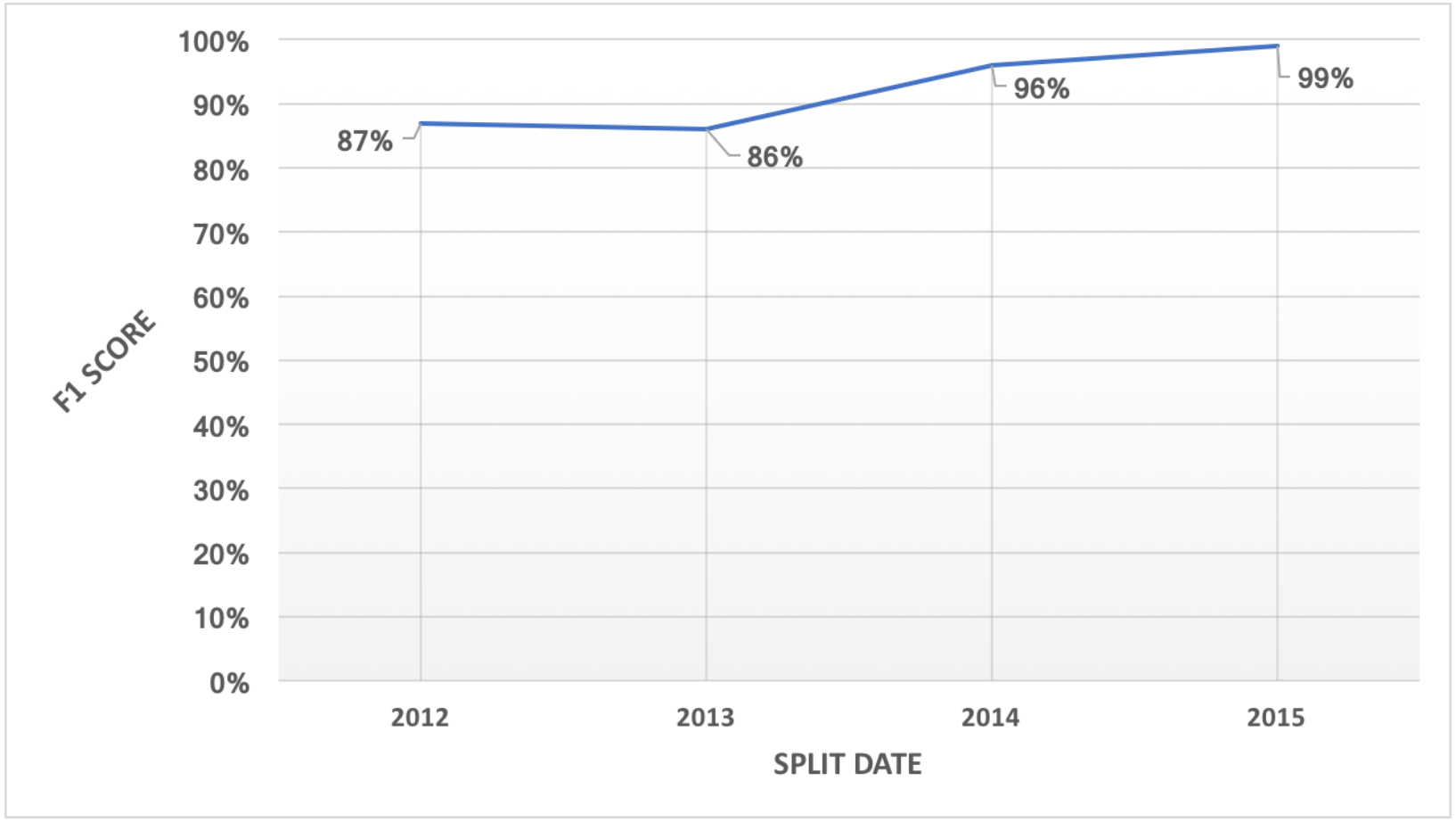
- 447 different malware families

Detection Accuracy: 10-Fold Cross-Validation

	Precision	Recall	F1
Benign	98%	97%	98%
Malicious	98%	98%	98%
Average	98%	98%	98%

RevealDroid has a very high detection accuracy

Time-Aware Malware Detection Results



Family Identification Accuracy: 10-Fold Cross Validation

	No. Apps	No. Families	RevealDroid Correct Classification Rate	Naïve Classifier Correct Classification Rate
Malware Genome	1,250	49	95%	25%
All Families	27,979	447	84%	26%

RevealDroid has a highly accurate family identification rate.

Applying Transformations to Obfuscate Apps

- Testing apps were obfuscated using DroidChameleon
  - Shown to evade all commercial antivirus products [ICSE 2018]

- Applied transformations
  - String encryption
  - Array encryption
  - Class renaming
  - Call indirection

Family identification accuracy on obfuscated apps

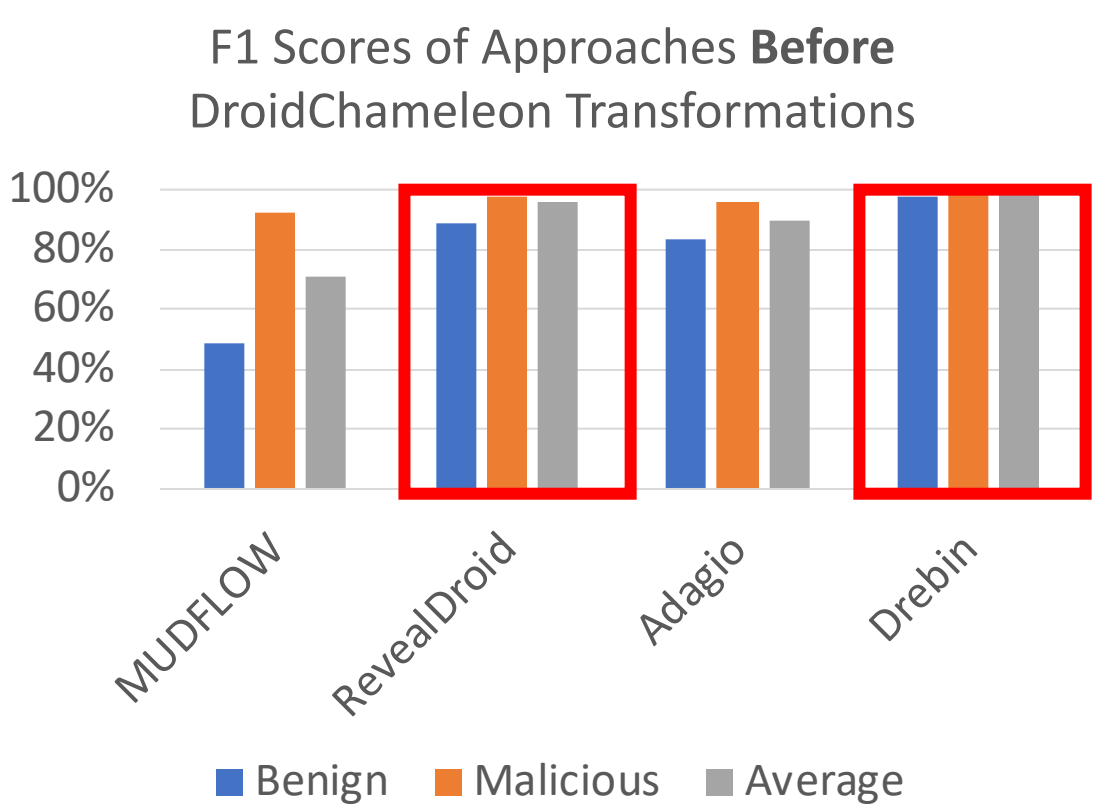
	No. Apps	No. Families	Correct Classification Rate
Malware Genome	1,233	33	97%

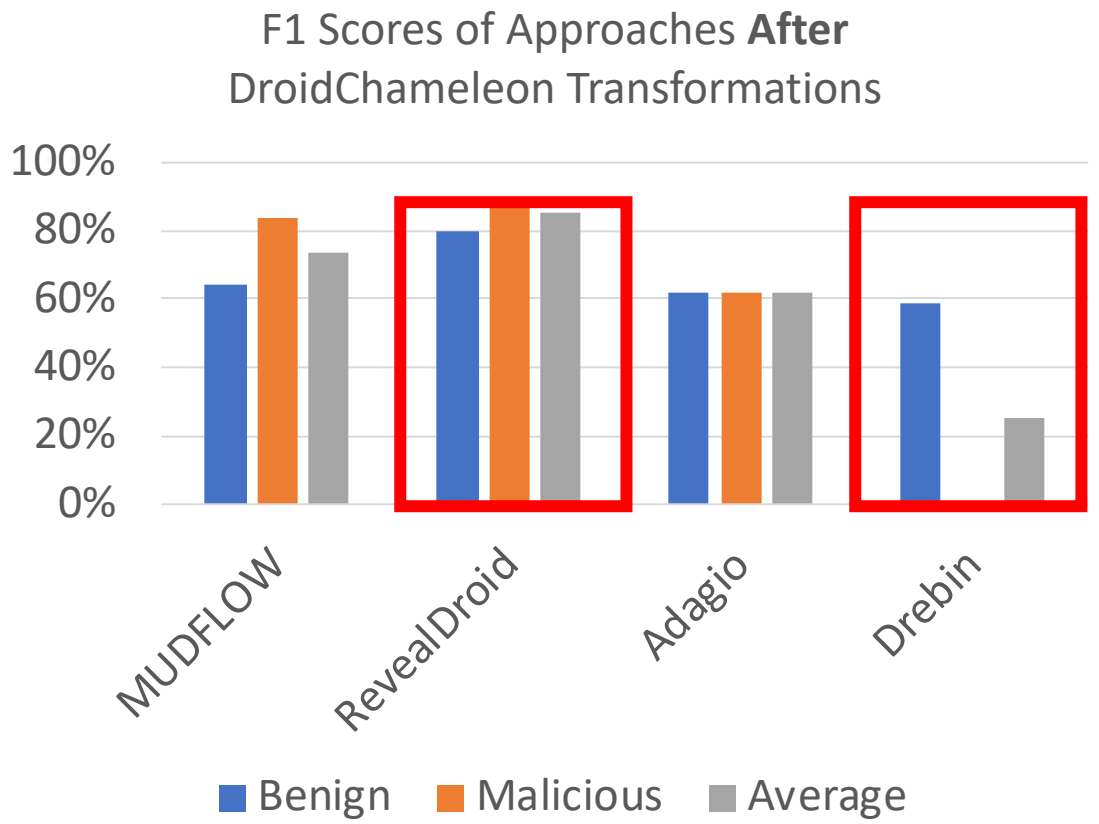
RevealDroid has a very high family identification rate even on obfuscated apps.

Setup: Comparison with State-Of-The-Art

- Three state-of-the-art approaches
  - MUDFLOW
  - Adagio
  - Drebin
- Compared with and without DroidChameleon transformations

Results: Comparison with State-of-the-Art Approaches

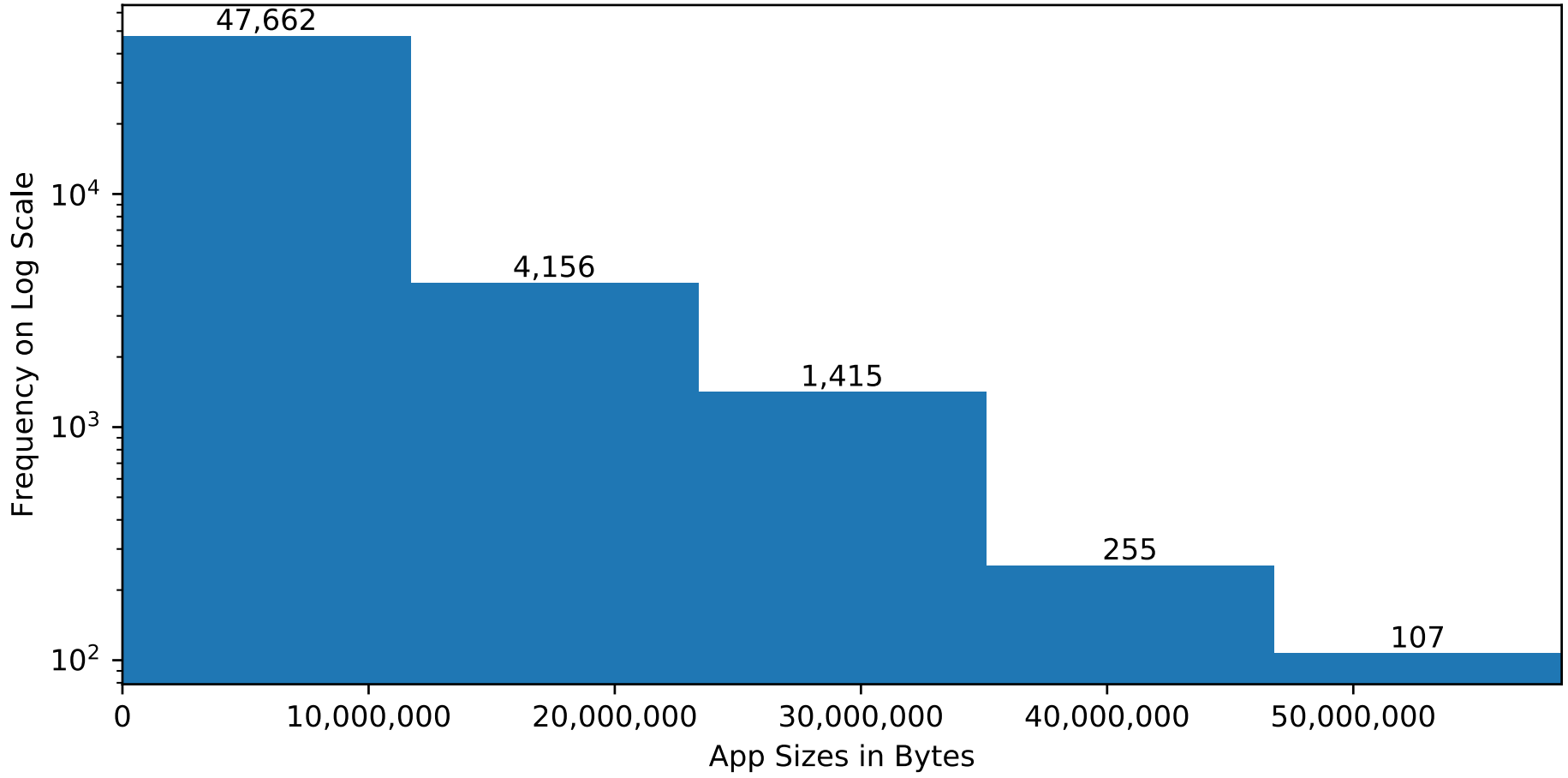




RevealDroid alone maintains high accuracy before and after obfuscations are applied.

Runtime Efficiency

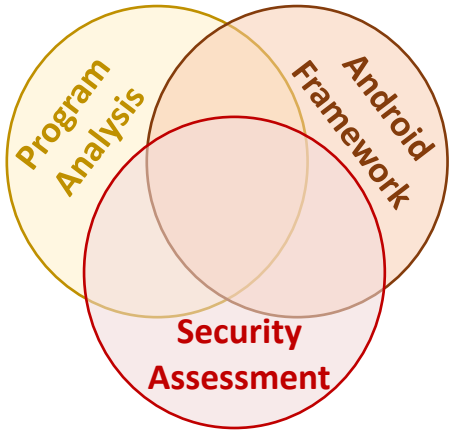
- Histogram of app sizes in bytes with 5 bins, with 20 apps from each bin



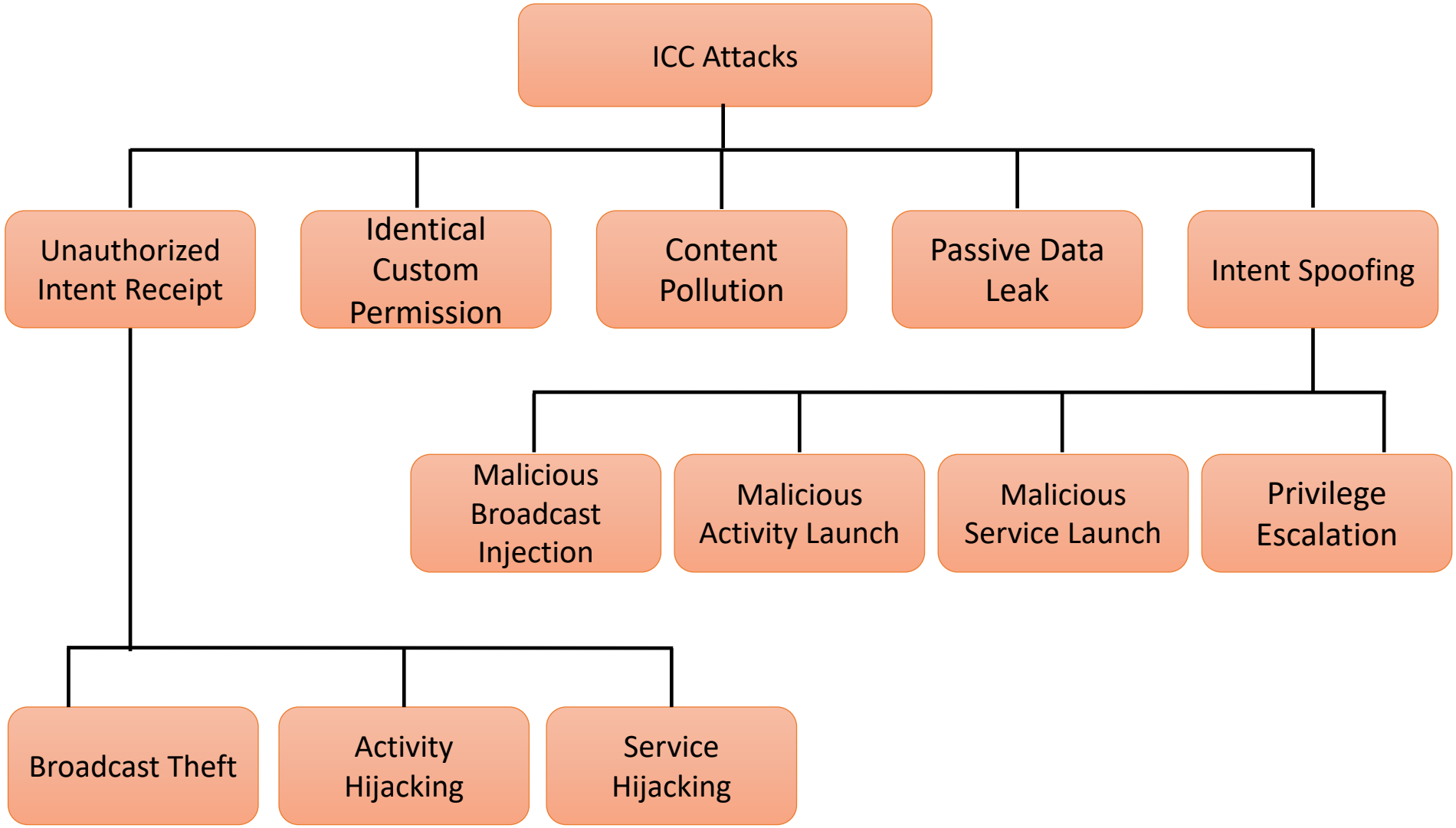
On average, RevealDroid extracts features in under 90 seconds.

Structure of Today's Lecture

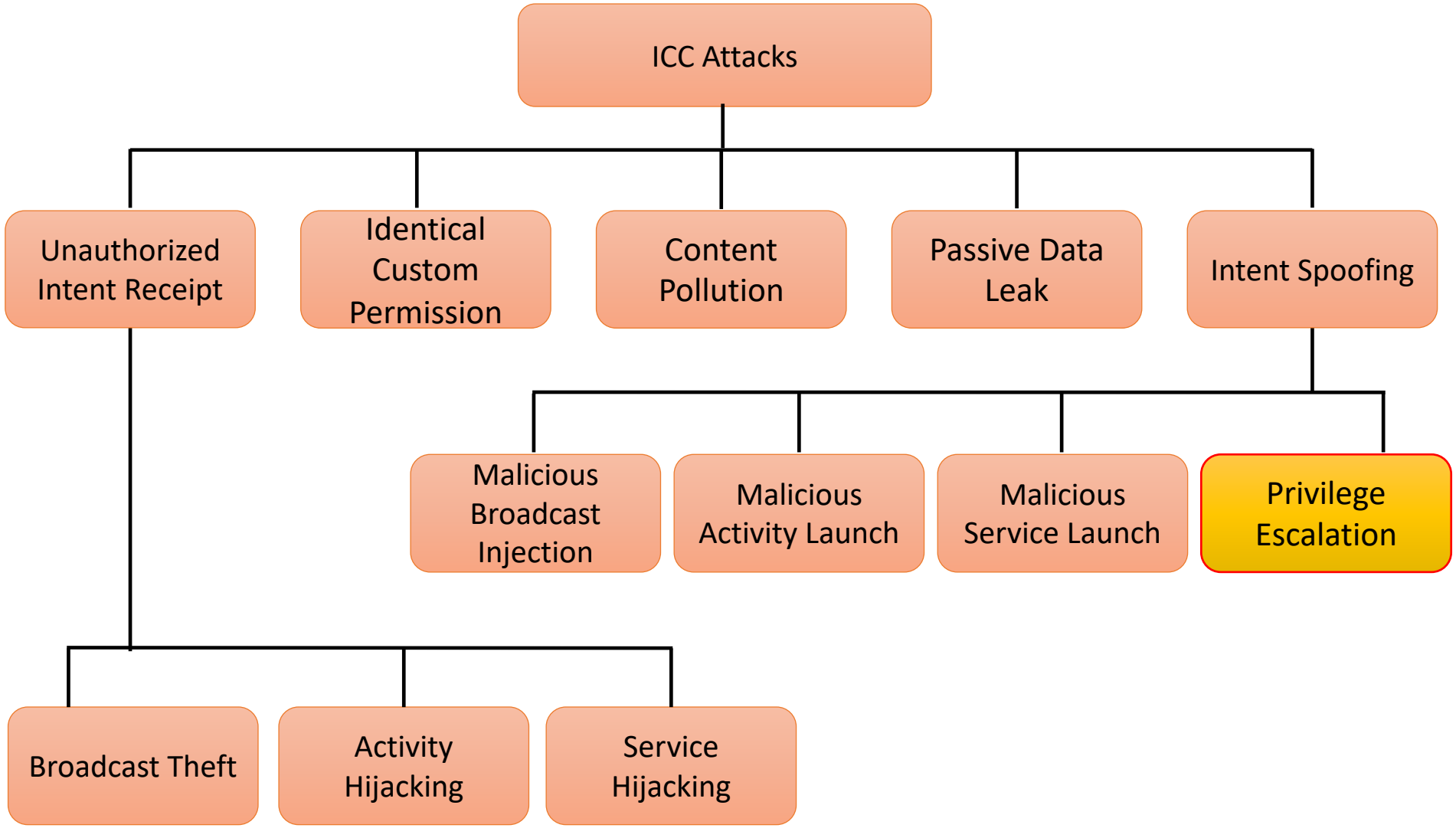
- A Taxonomy of Program Analysis Techniques for Security Assessment of Android Software
- Automatic Exploit Generation of Android Apps
- Lightweight, Obfuscation-Resilient Android Malware Classification
- Self-Protection of Android Systems from Inter-Component Communication Attacks



Inter-component communication attacks

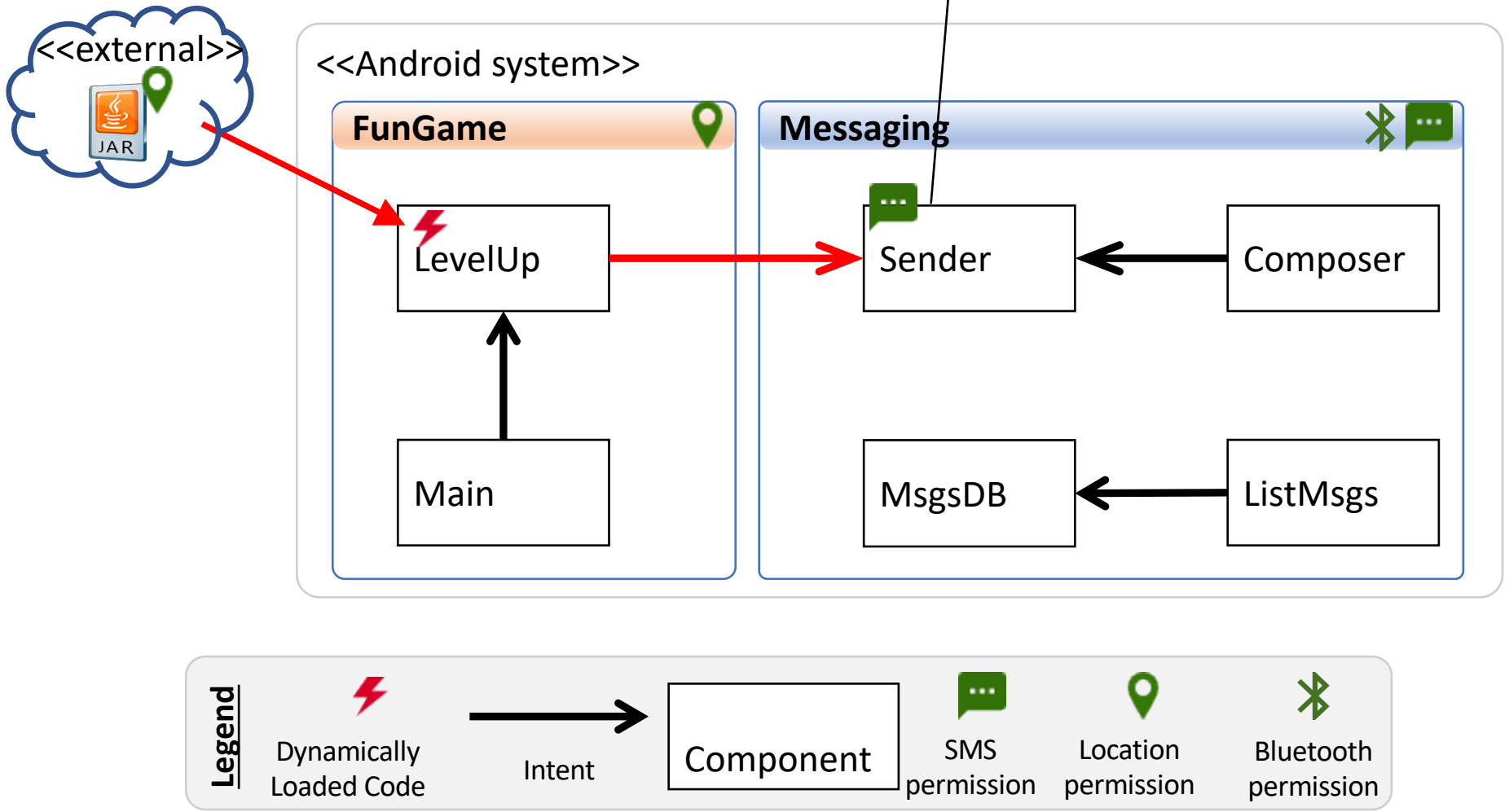


Inter-component communication attacks

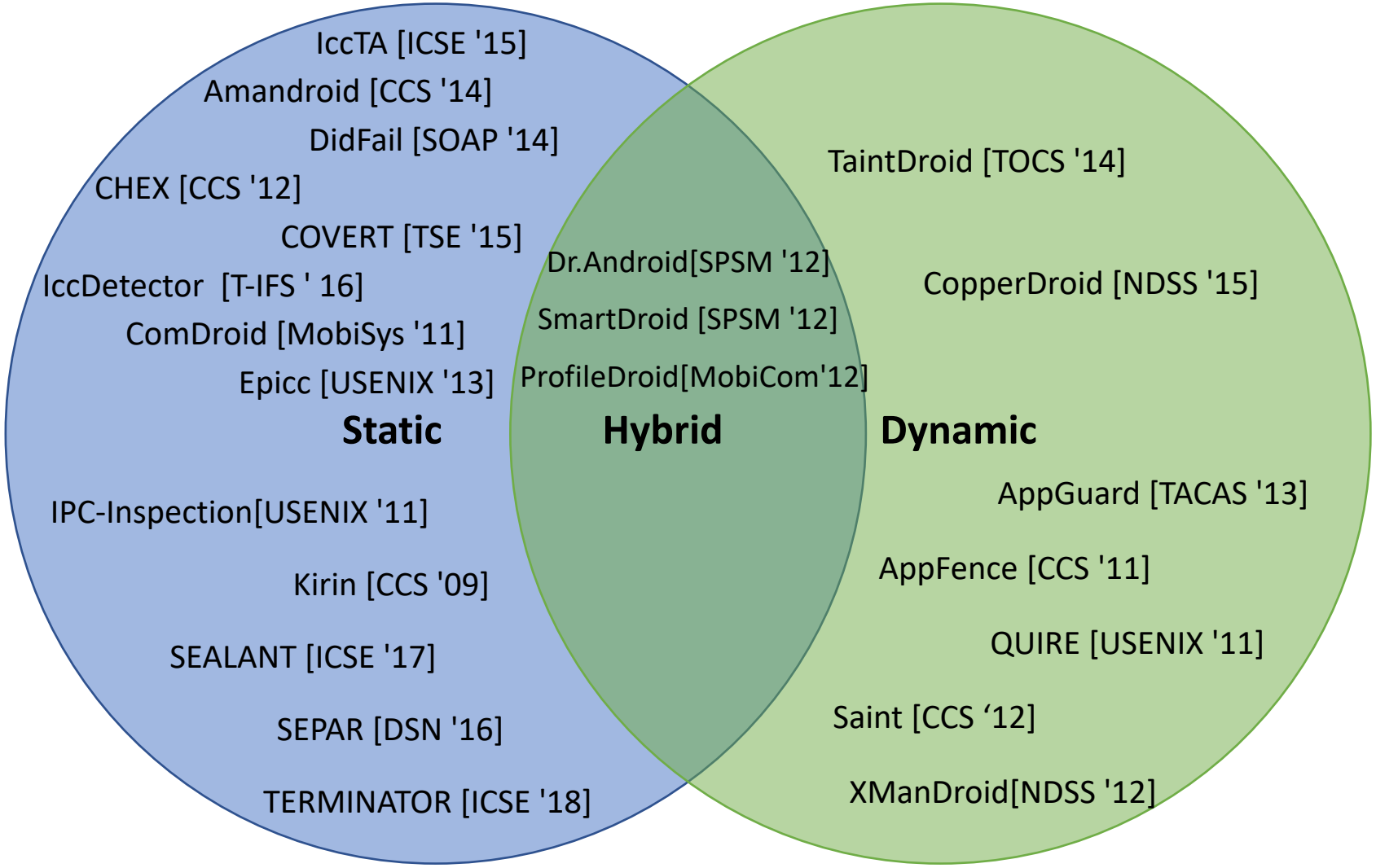


Privilege escalation I(

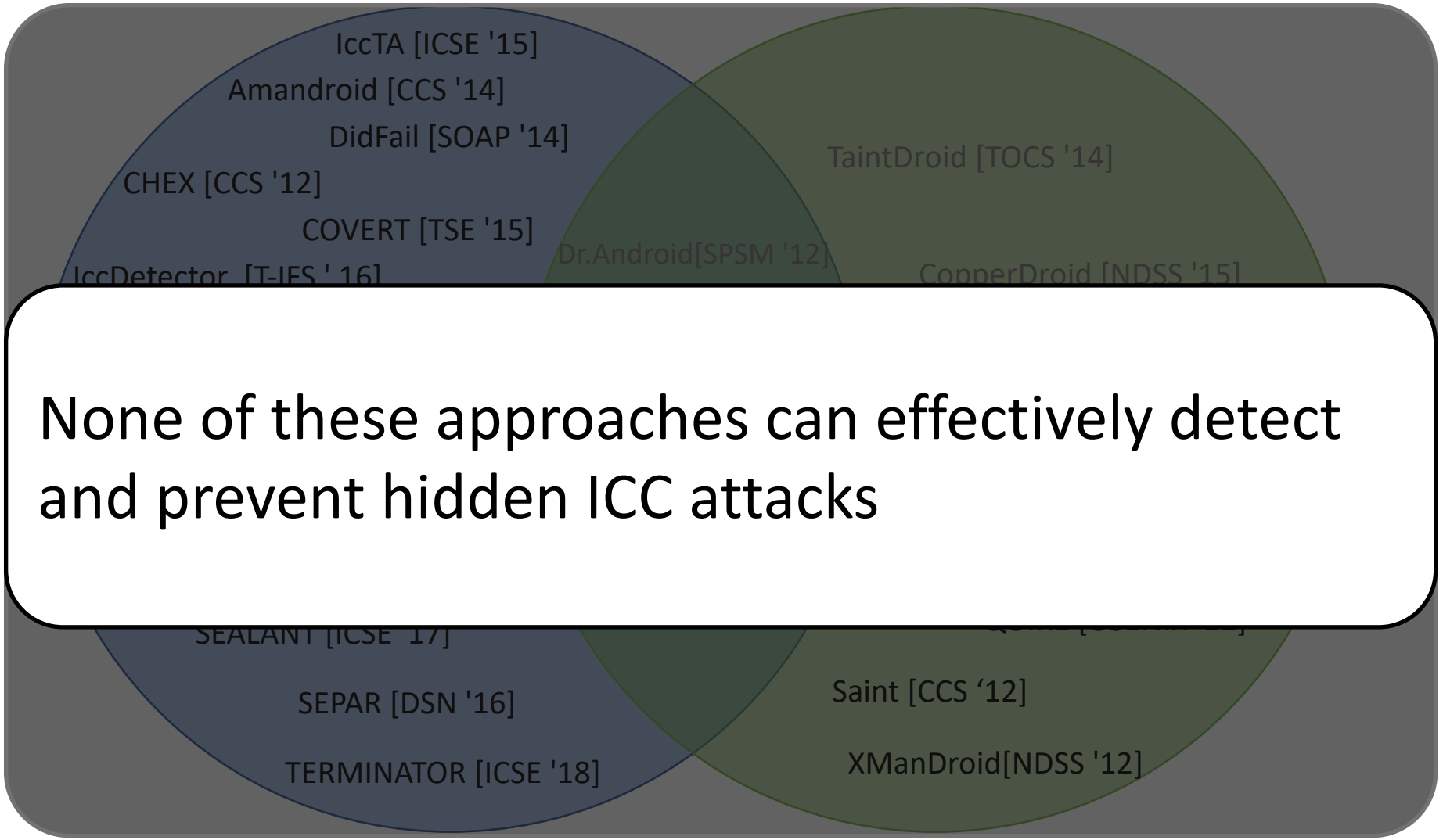
```
// if (checkCallingPermission
("SEND_SMS") ==
PackageManager.PERMISSION_GRANTED)
SendSMS ….
```



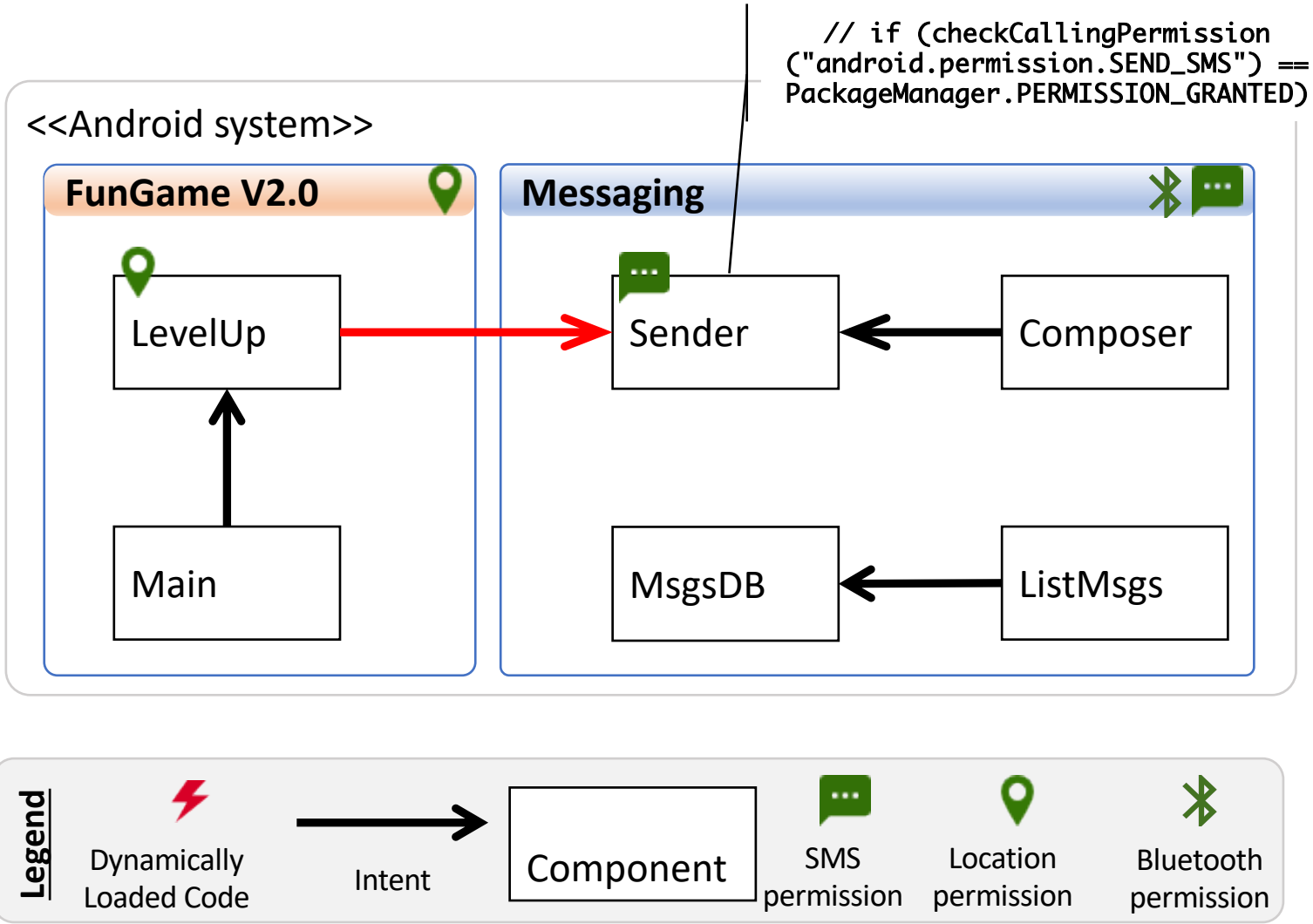
Limitations of the current security mechanisms



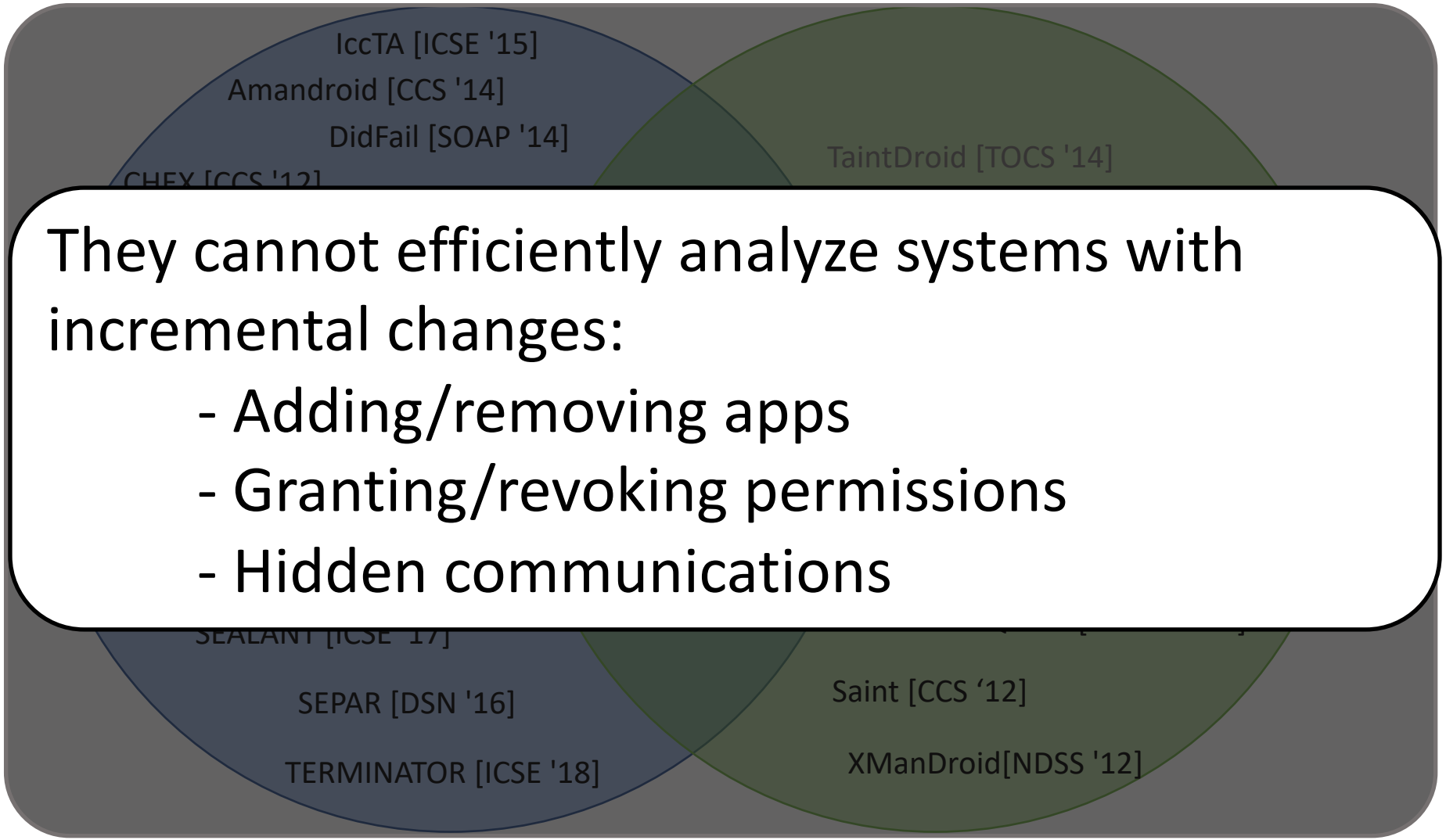
Limitations of the current security mechanisms



Android systems are highly dynamic



Limitations of the current security mechanisms

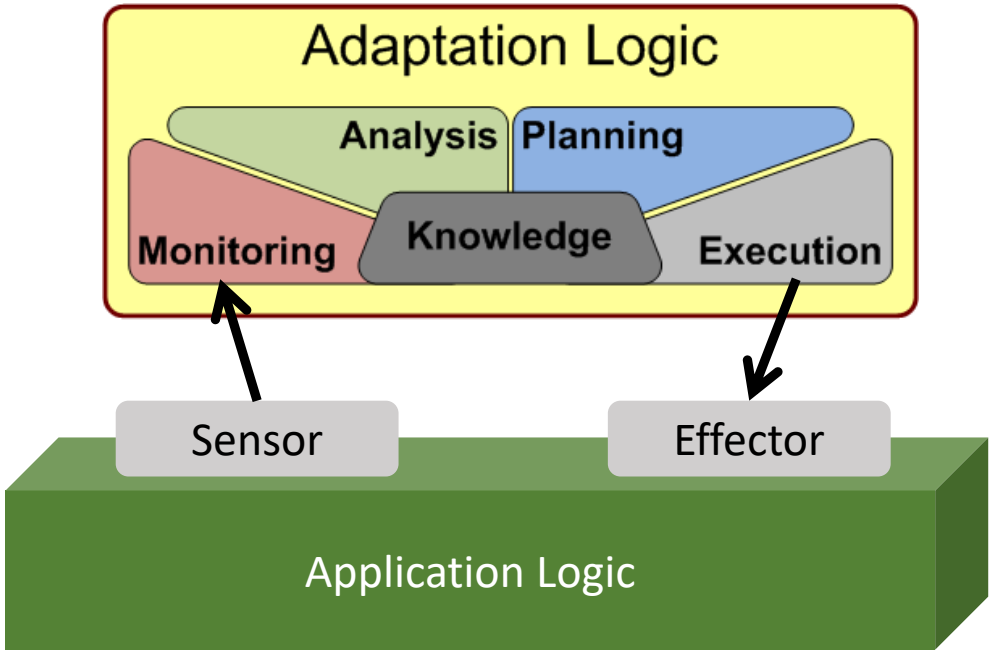


Outline

- ✓ Motivation and research problem
- ❑ Approach: self-protecting Android software system
  - ❑ Determination and enforcement of the architecture
  - ❑ Continuous monitoring and efficient analysis
- ❑ Evaluation
- ❑ Conclusion

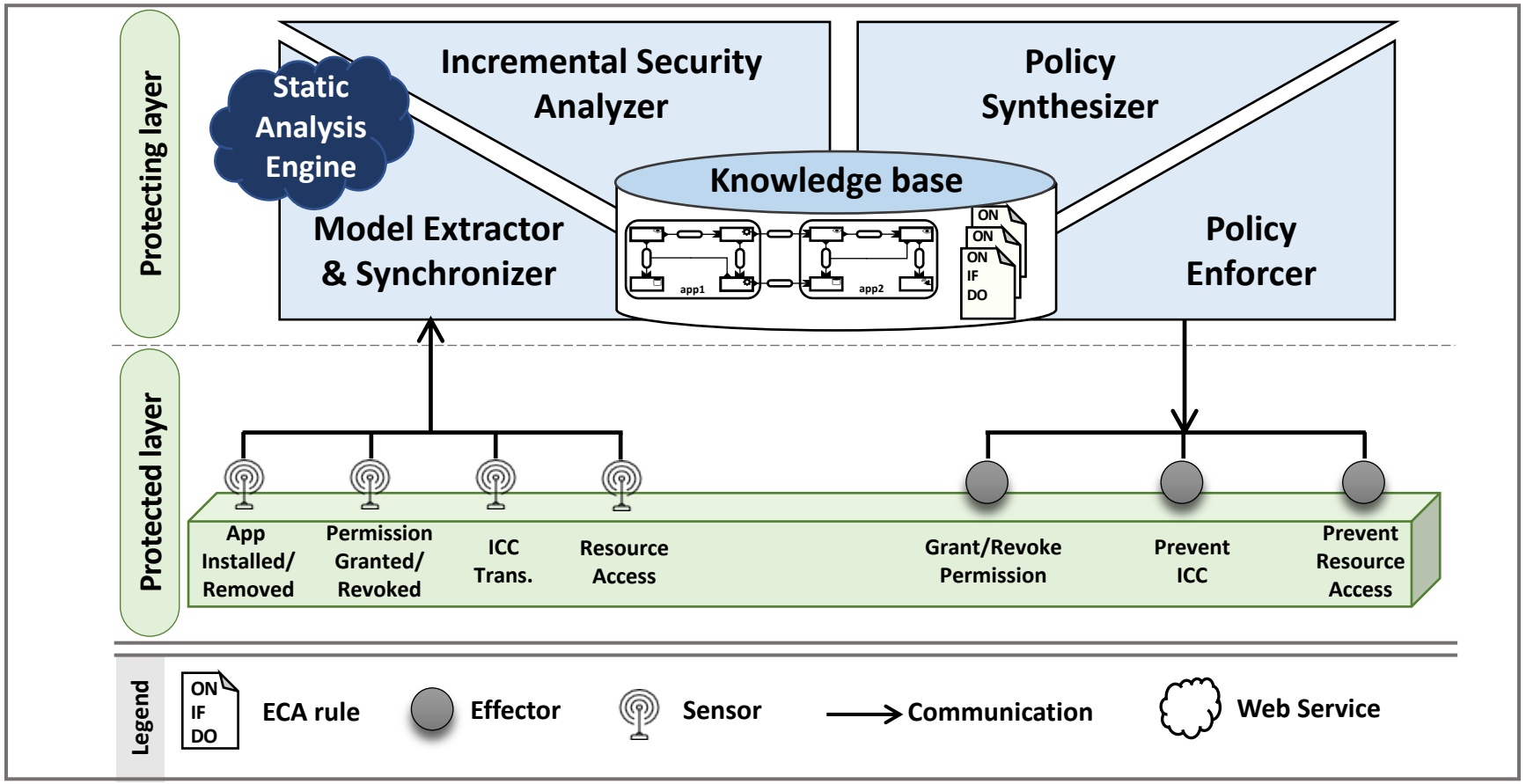
Self-protecting Software System

- Software system that monitors itself and adapts its behavior at runtime
- Adaptation logic relies on the **IBM MAPE-K** [1] reference architecture

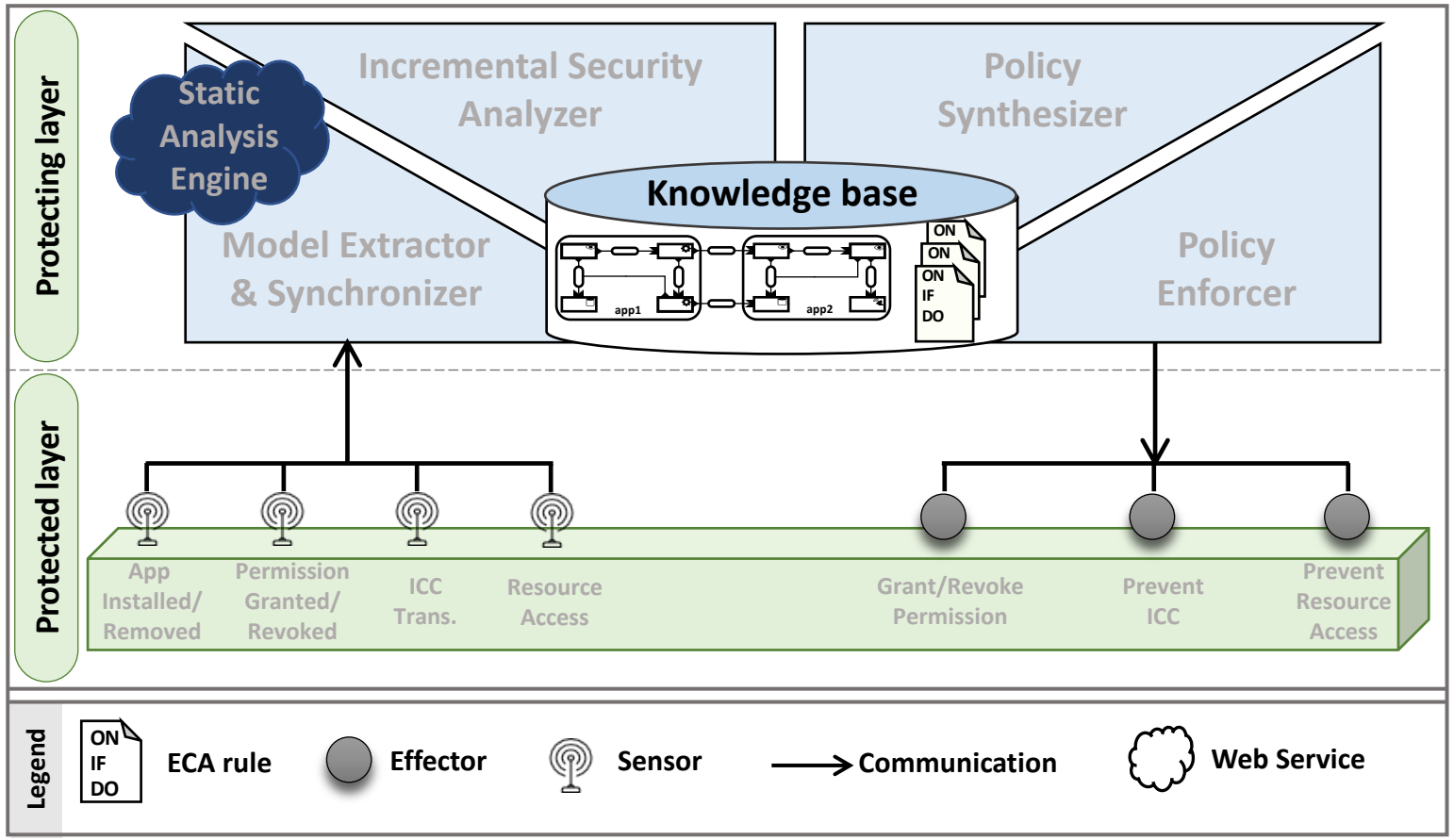


[1] J. O. Kephart and D. M. Chess. The vision of autonomic computing. IEEE Computer, 2003.

SALMA: Self-protecting Android System

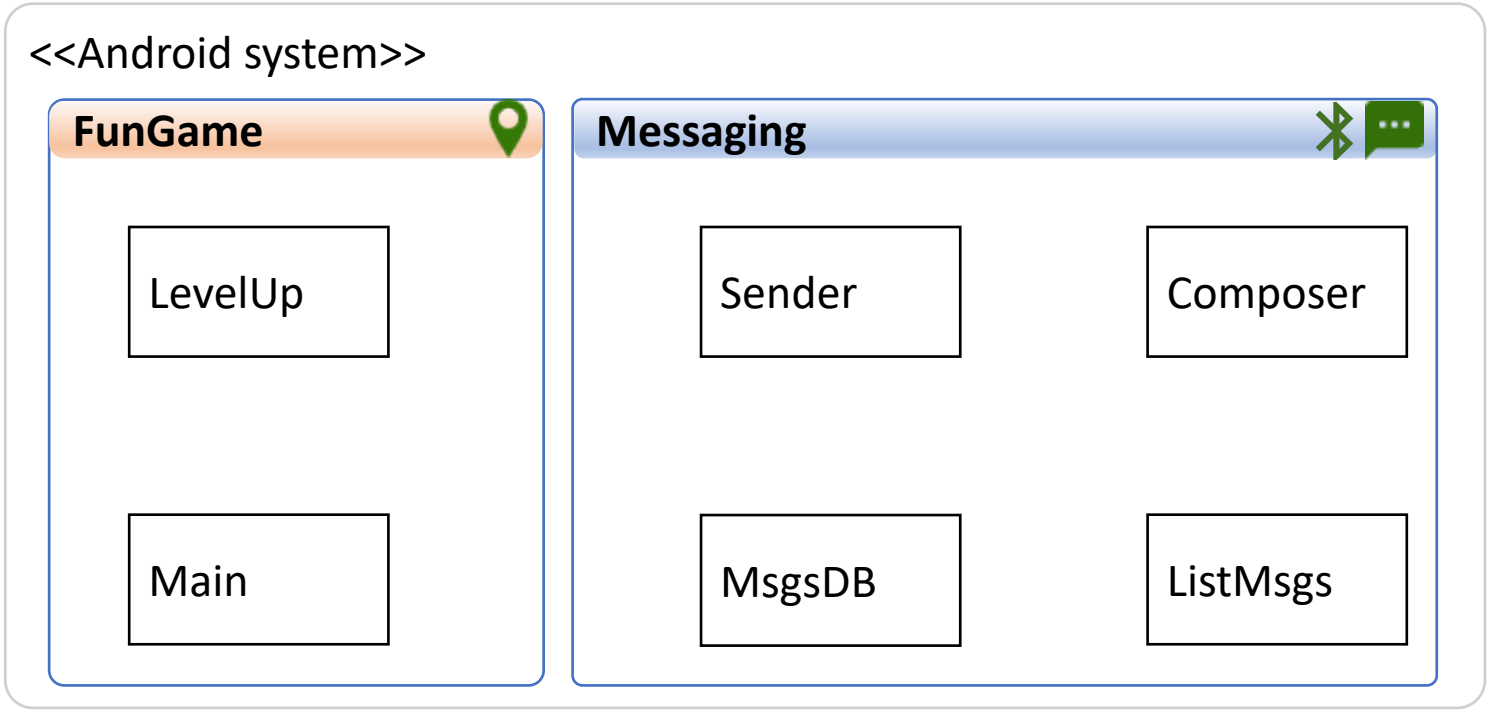


SALMA: Self-protecting Android System



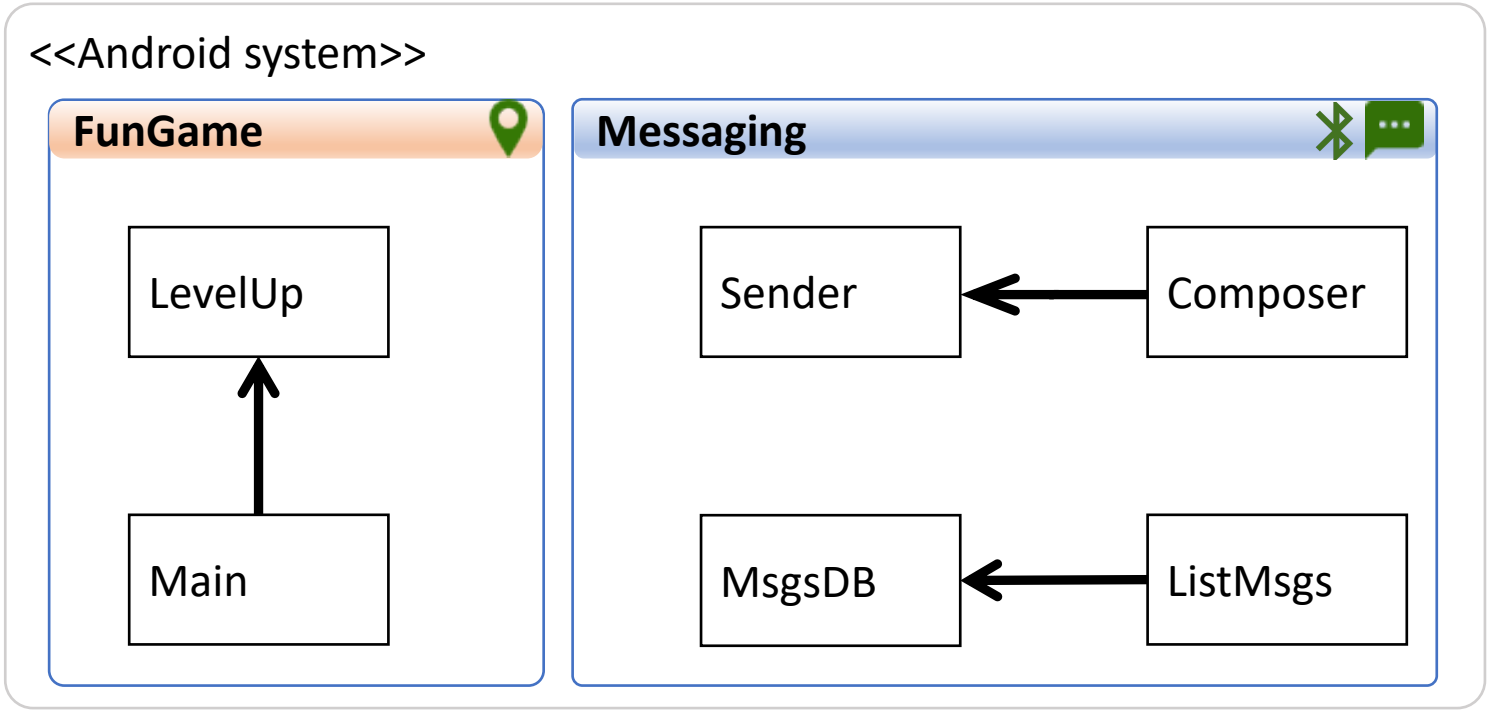
Static extraction of architecture

1. **Architectural elements and properties defined in the manifest file**



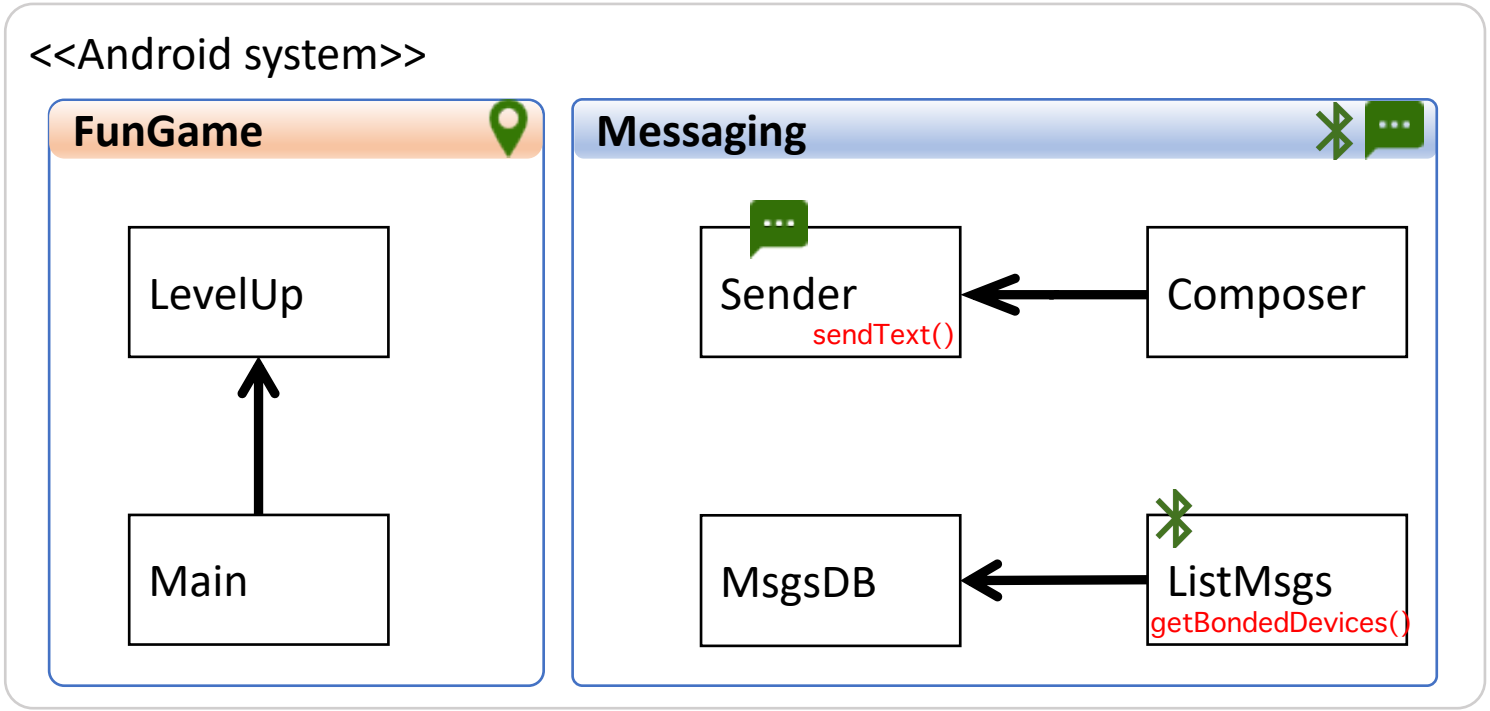
Static extraction of architecture

1. Architectural elements and properties defined in the manifest file
2. **Event-driven behavior of each app (e.g., Intent and Filters) that are latent in code**



Static extraction of architecture

1. Architectural elements and properties defined in the manifest file
2. Architectural elements (e.g., Intent and Filters) that are latent in code
3. **Needed permission for each component from its implementation logic**

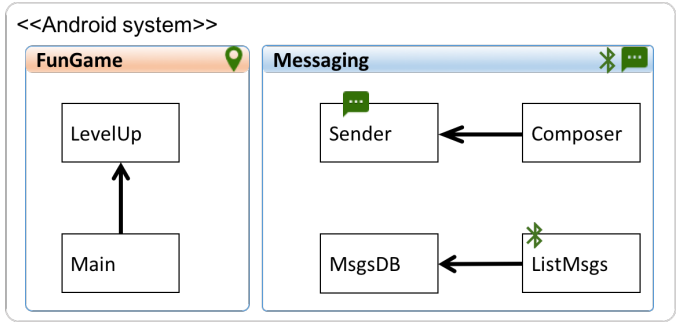


Pscout [CCS `12]: maps Android API calls and the permissions required to perform those calls

Multiple Domain Matrix (MDM)

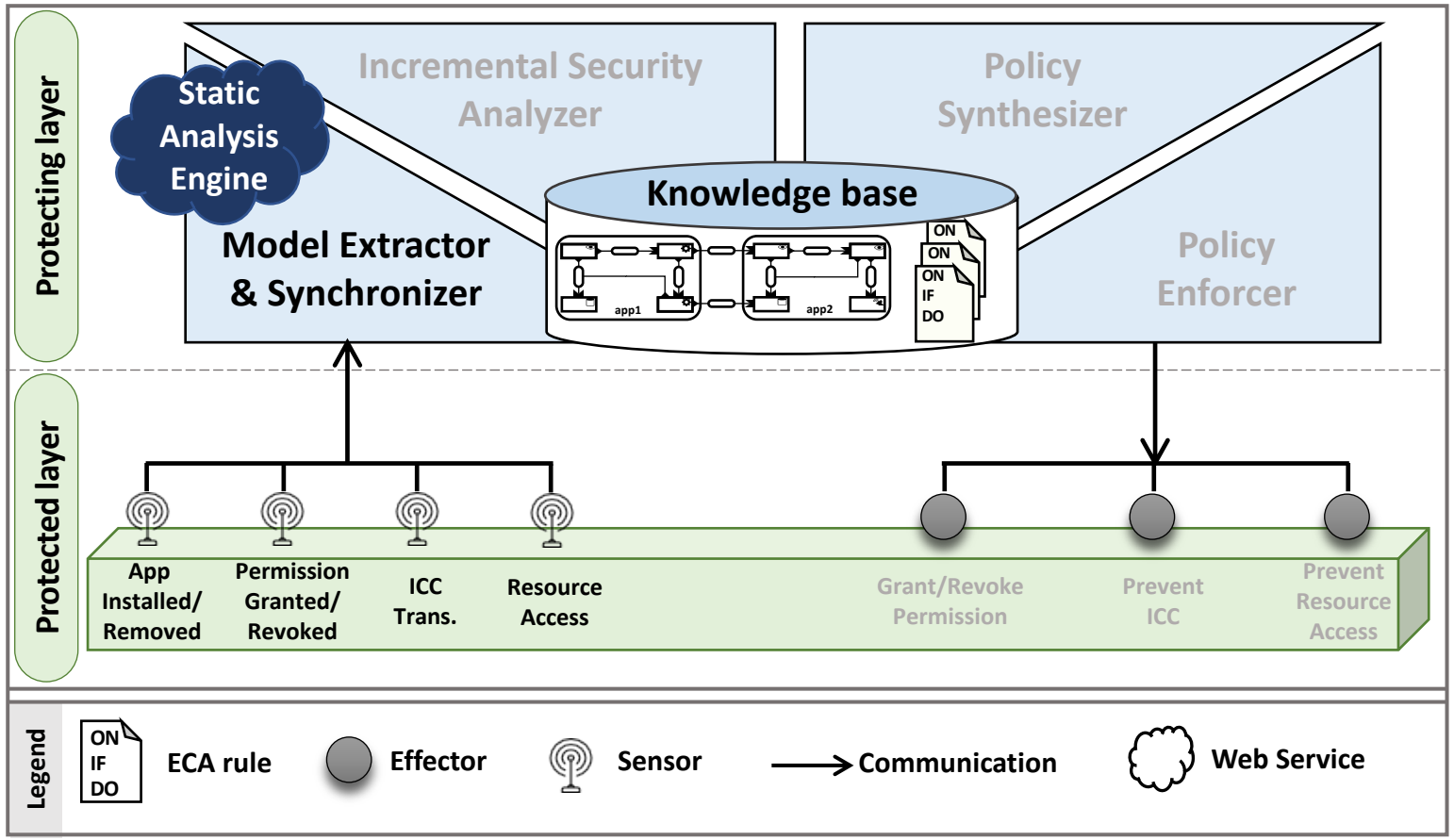
- MDM models a complex system with multiple domains
- Each domain is modeled as a Design Structure Matrix (DSM)
- DSM and MDM are very effective in capturing and analyzing the architecture of a complex system

MDM representation of the architecture

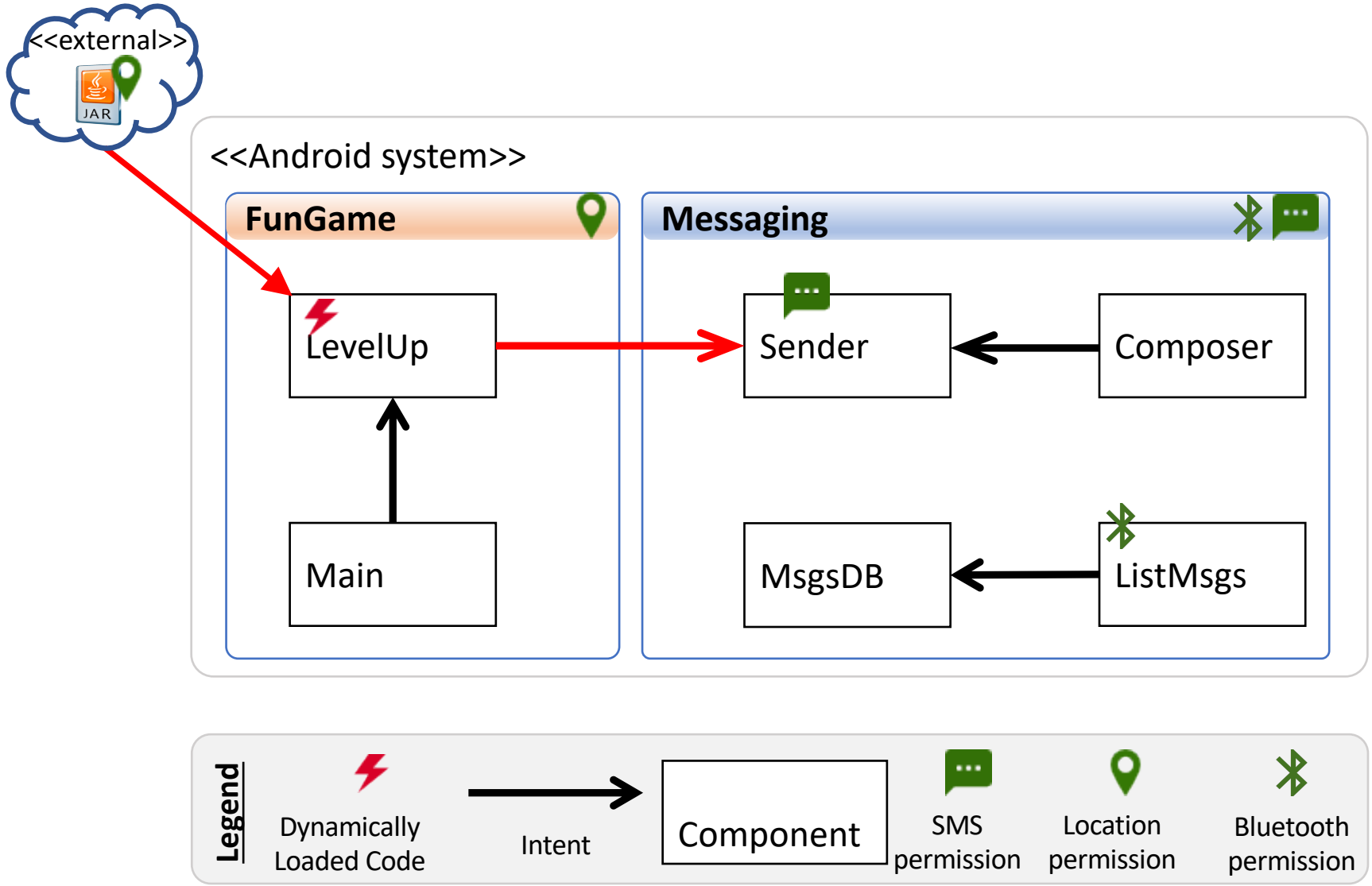


			Communication Domain						Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	6									
Messaging	ListMsgs	1		1							1			1			
	Composer	2			1					1							
	Sender	3								1			1				
	MsgsDB	4															
FunGame	LevelUp	5															
	Main	6					1										

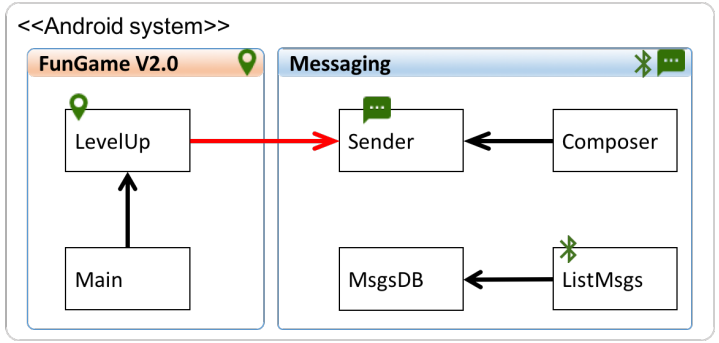
Self-protecting Android System



Evolving Android system: Architecture

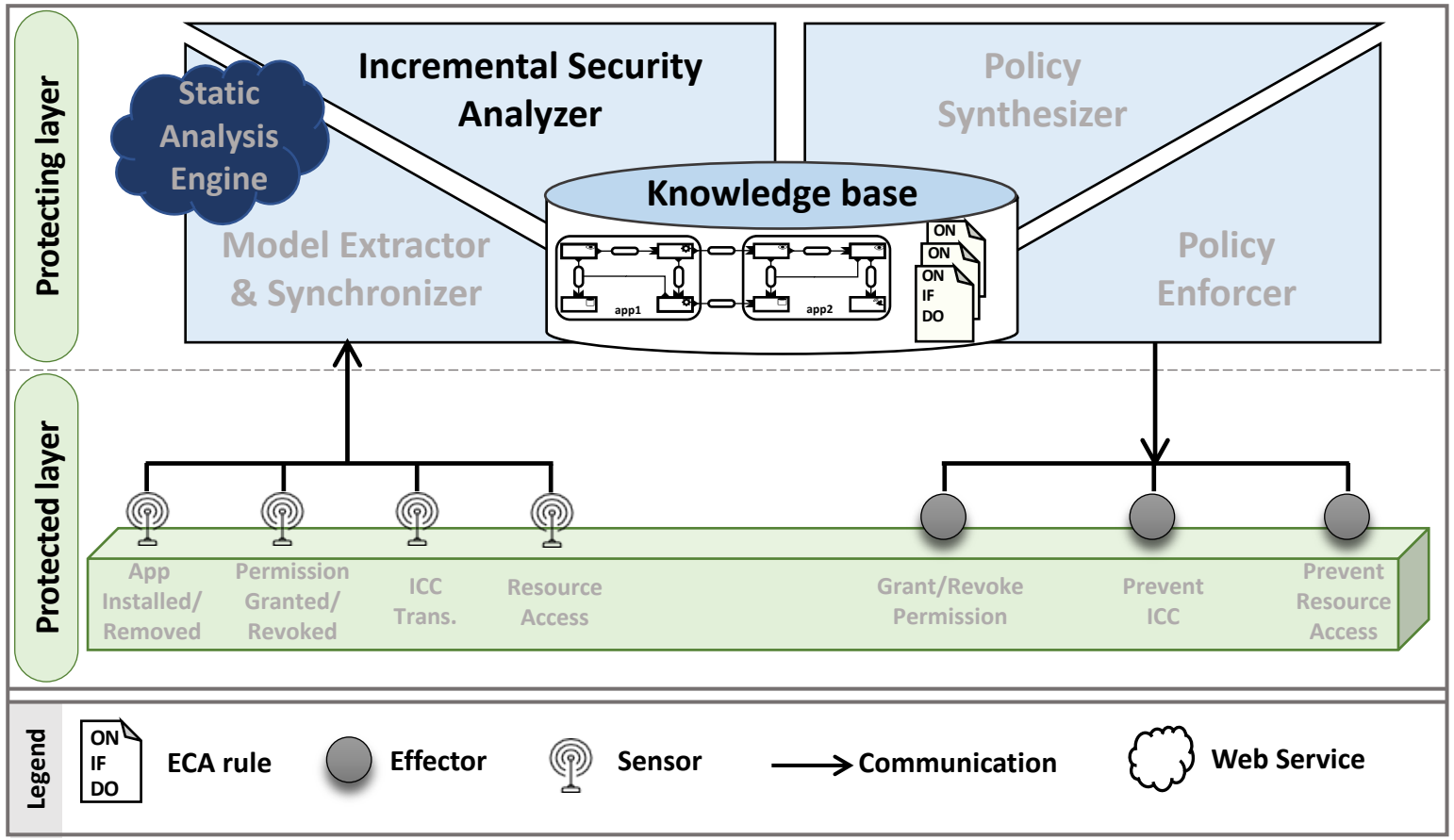


Evolving Android system: MDM

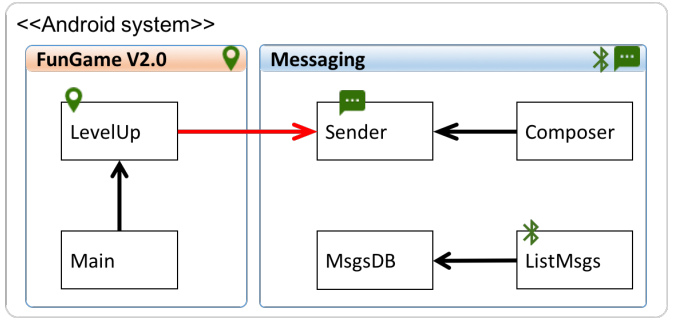


			Communication Domain						Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	6									
Messaging	ListMsgs	1		1							1			1			
	Composer	2			1					1							
	Sender	3								1			1				
	MsgsDB	4															
FunGame	LevelUp	5			1				1			1					
	Main	6					1		1								

Self-protecting Android System



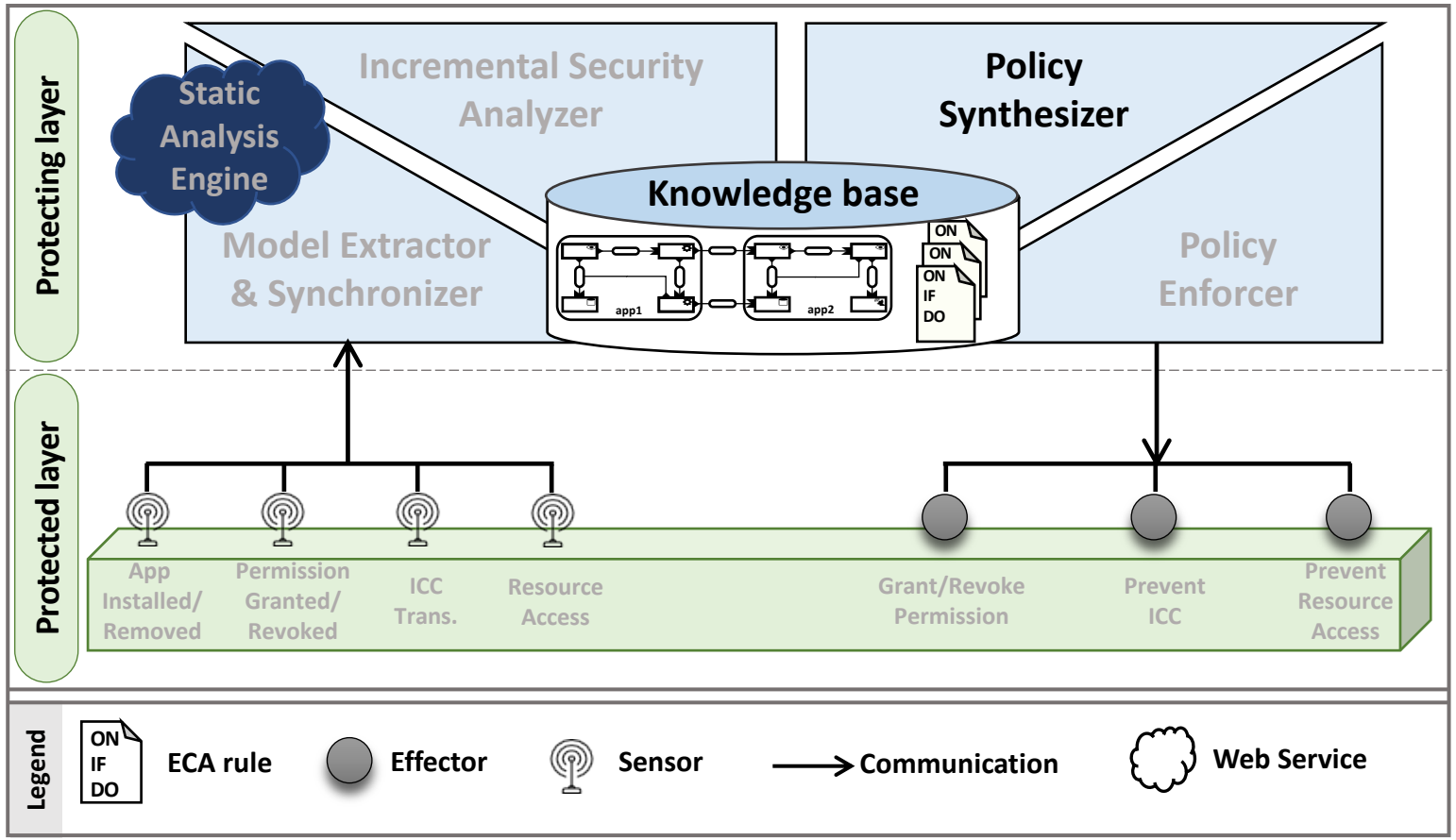
Privilege escalation analysis



			Communication Domain						Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	6									
Messaging	ListMsgs	1		1							1			1			
	Composer	2			1					1							
	Sender	3								1			1				
	MsgsDB	4															
FunGame	LevelUp	5			1				1			1					
	Main	6					1		1								

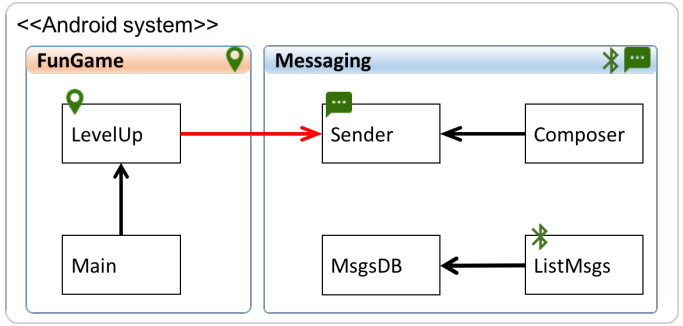
communicate **(*LevelUp*, *Sender*) will be marked as a privilege escalation attack**

Self-protecting Android System



Policy Synthesizer create security policies in the form of Event-Condition-Action (ECA) rules.

Privilege escalation analysis

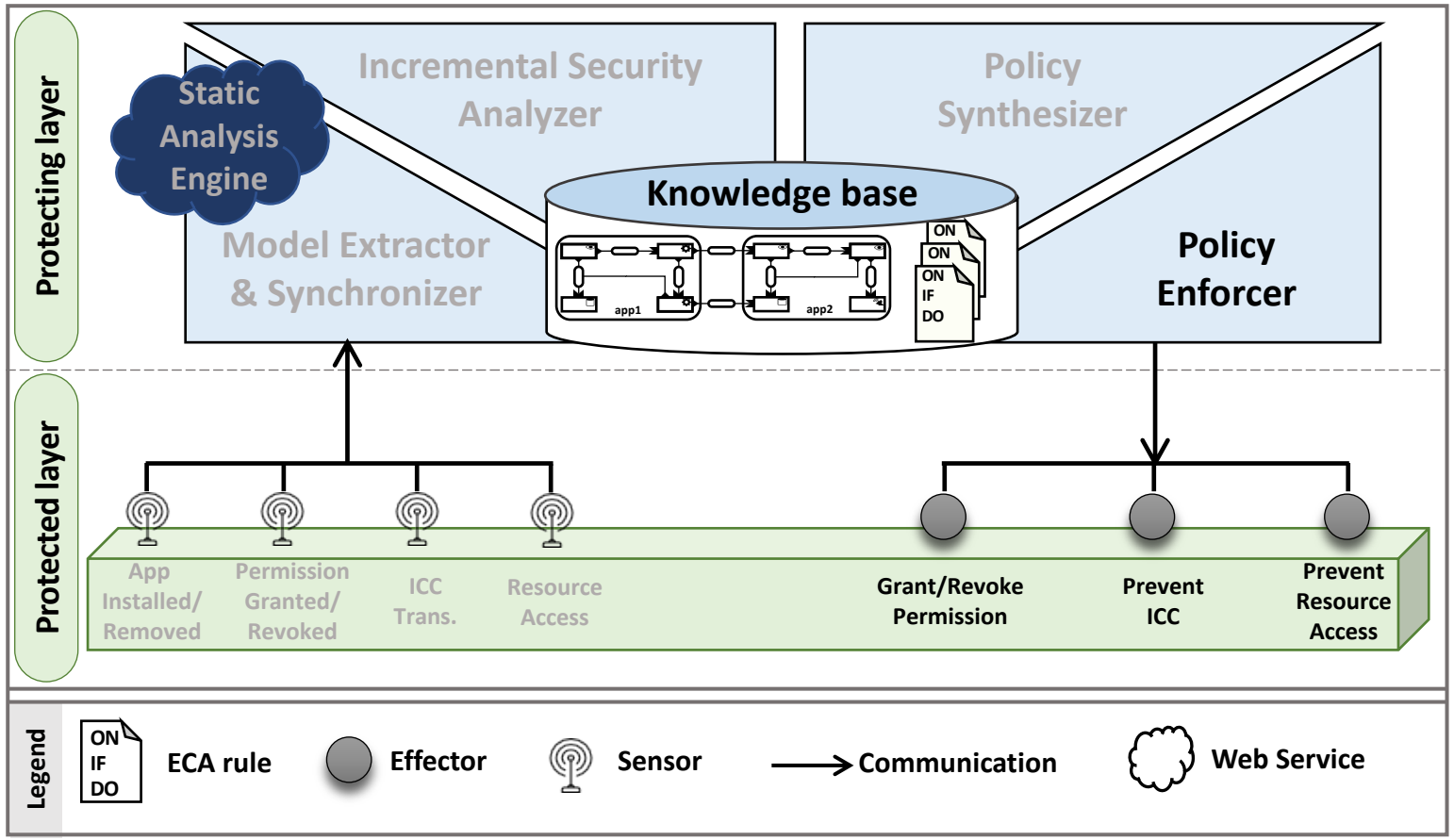


			Communication Domain						Permission Granted Domain			Permission Usage Domain			Permission Enforcement Domain		
		ID	1	2	3	4	5	6									
Messaging	ListMsgs	1		1							1			1			
	Composer	2			1					1							
	Sender	3								1			1				
	MsgsDB	4															
FunGame	LevelUp	5			1				1			1					
	Main	6					1		1								

communicate **(*LevelUp*, *Sender*) will be marked as a privilege escalation attack**

Event: $i \in ICC\ occurs$
Condition: $i.senderComp = FunGame.LevelUp \wedge i.receiverComp = Messaging.Sender$
Action: $prevent$

Self-protecting Android System



Outline

- ✓ Motivation and research problem
- ✓ Approach: self-protecting Android software system
  - ✓ Determination and enforcement of LP architecture
  - ✓ Continuous monitoring and efficient analysis
- ❑ Evaluation
- ❑ Conclusion

Evaluation

1. Attacks detection & prevention
2. Incremental analysis efficiency
3. Disruption
4. Performance

(1) Attack detection and prevention

- 188 malicious and vulnerable apps
  - The steps and inputs required to create the attacks are known
- The dataset contains 94 ICC attacks
  - 45 hidden ICC attacks through dynamic class loading
- Compared our approach with other approaches:
  - DELDroid [ICSA'17]
  - SEPAR [DSN'16]
  - SEALANT [ICSE'17]

(1) Attack detection

Attack Type (Count)	Security Attack	# Attacks	Attack Detection			
			DELDroid	SEPAR	SEALANT	SALMA
Not hidden (49)	Intent Spoofing	3	3	3	2	3
	Unauthorized Intent Receipt	5	5	0	1	5
	Privilege Escalation	20	20	12	14	20
	Identical Custom Permission	7	0	0	1	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Hidden (45)	Intent Spoofing	13	0	0	0	13
	Unauthorized Intent Receipt	2	0	0	0	2
	Privilege Escalation	9	0	0	0	9
	Identical Custom Permission	7	0	0	0	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
	Total attacks	94	28	15	18	94
	Detection Rate		30%	16%	19%	100%

DELDroid: M. Hammad, H. Bagheri, and S. Malek. Determination and Enforcement of Least-Privilege Architecture in Android. In International Conference of Software Architecture (ICSA), Gothenburg, Sweden, April 2017.

SEPAR: H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In Int'l Conf. Dependable Systems and Networks (DSN), Toulouse, France, June 2016.

SEALANT: Y. K. Lee, J. Y. Bang, G. Saff, A. Shahbazian, Y. Zhao, and N. Medvidovic. A sealant for inter-app security holes in android. In International Conference of Software Engineering (ICSE), Buenos Aires, Argentina, May 2017

(1) Attack prevention

Attack Type (Count)	Security Attack	# Attacks	Attack Prevention			
			DELDroid	SEPAR	SEALANT	SALMA
Not hidden (49)	Intent Spoofing	3	3	3	2	3
	Unauthorized Intent Receipt	5	5	0	1	5
	Privilege Escalation	20	20	12	14	20
	Identical Custom Permission	7	0	0	1	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Hidden (45)	Intent Spoofing	13	13	0	0	13
	Unauthorized Intent Receipt	2	2	0	0	2
	Privilege Escalation	9	9	0	0	9
	Identical Custom Permission	7	0	0	0	7
	Content Pollution	7	0	0	0	7
	Passive Data Leak	7	0	0	0	7
Total attacks		94	52	15	18	94
Prevention Rate			55%	16%	19%	100%

DELDroid: M. Hammad, H. Bagheri, and S. Malek. Determination and Enforcement of Least-Privilege Architecture in Android. In International Conference of Software Architecture (ICSA), Gothenburg, Sweden, April 2017.

SEPAR: H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In Int'l Conf. Dependable Systems and Networks (DSN), Toulouse, France, June 2016.

SEALANT: Y. K. Lee, J. Y. Bang, G. Saff, A. Shahbazian, Y. Zhao, and N. Medvidovic. A sealant for inter-app security holes in android. In International Conference of Software Engineering (ICSE), Buenos Aires, Argentina, May 2017

Evaluation

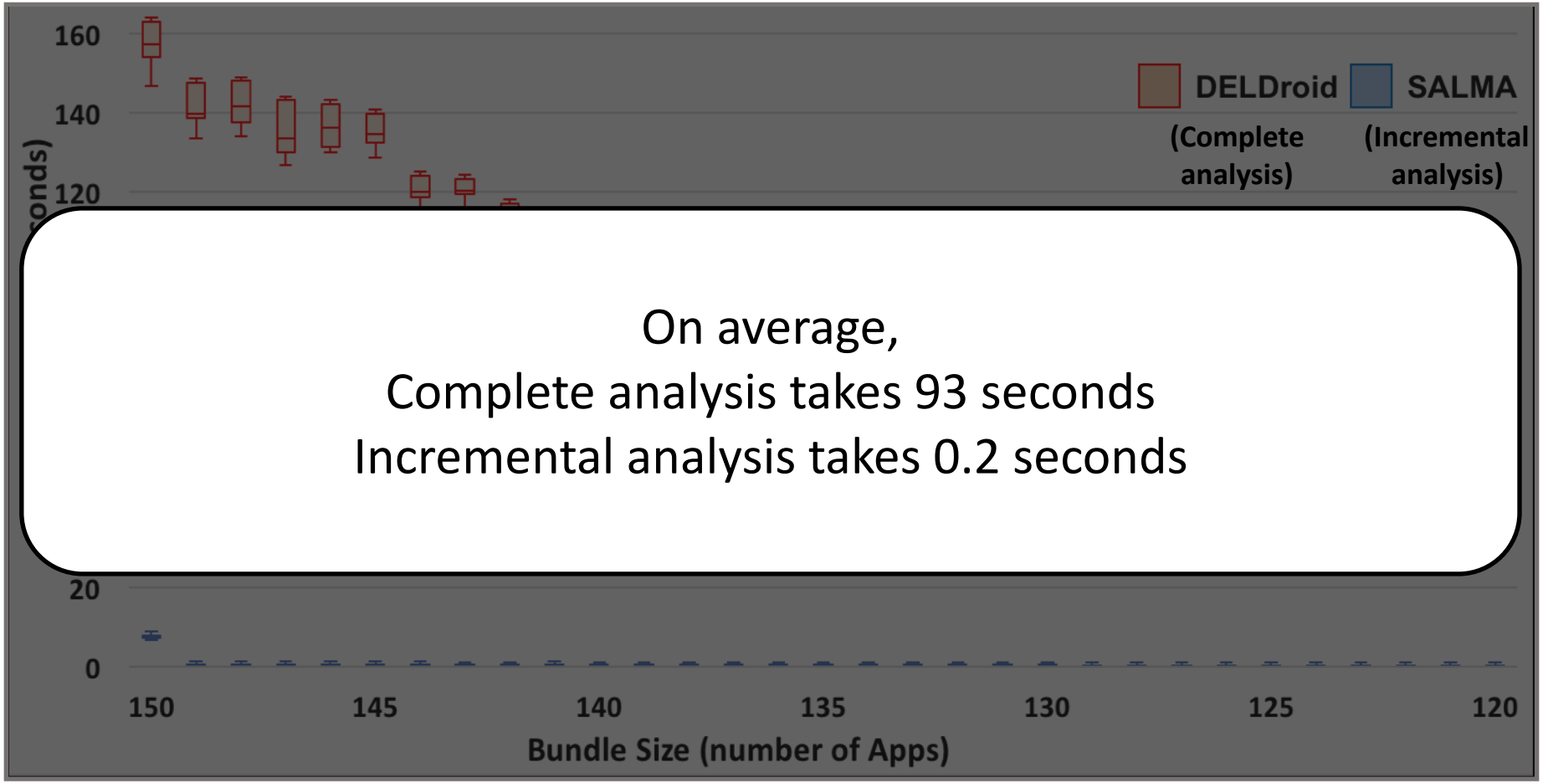
1. Attacks detection & prevention

2. **Incremental analysis efficiency**

3. Disruption

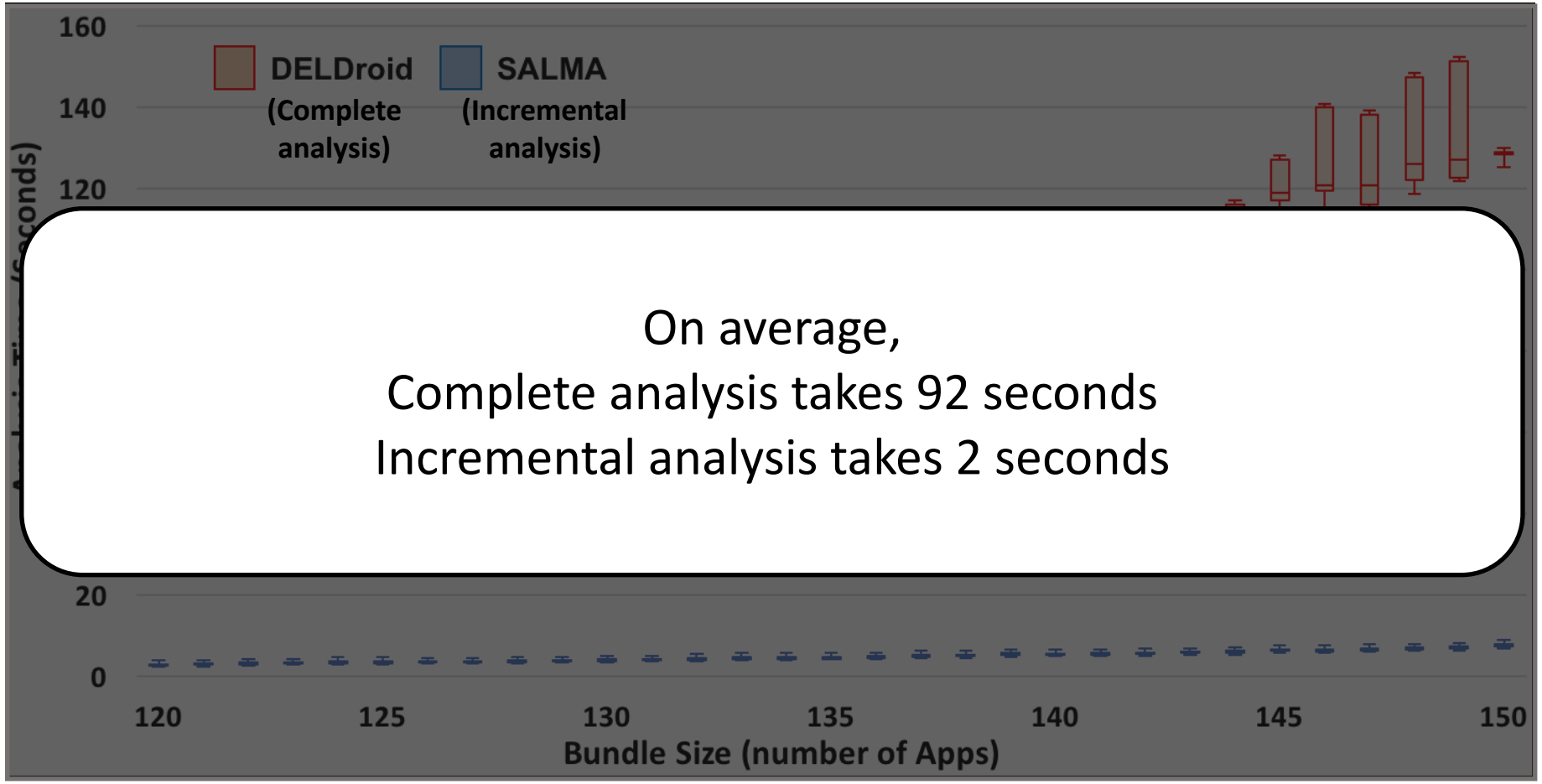
4. Performance

(2) Incremental analysis performance (removing existing apps)



The analysis time of the incremental analysis against the complete analysis approach over a decreasing size of an Android system.

(2) Incremental analysis performance (installing new apps)



The analysis time of the incremental analysis against the complete analysis approach over an increasing size of an Android system.

Structure of Today's Lecture

- A Taxonomy of Program Analysis Techniques for Security Assessment of Android Software
- Automatic Exploit Generation of Android Apps
- Lightweight, Obfuscation-Resilient Android Malware Classification
- Self-Protection of Android Systems from Inter-Component Communication Attacks