Benchmarking Android Security Analysis

A Bachelors Project,

Intermediate Presentation

by Timo Spring

Supervised by Claudio Corrodi

1. Project Motivation

What Is It About?

Problem

- Millions of android apps
- Hundreds of Analyses tools
- Large scale taxonomies classifying them
- Lack of comparison in practice

Project Idea

- Run selected tools on common dataset
- Evaluate the results and compare them
- Draw conclusions why they might be different

2. Tool Selection Process

Literature: Reviewing Types Of Vulnerabilities

SCAM 2017

Security Smells in Android

Mohammad Ghafari, Pascal Gadient, Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
{ghafari, gadient, oscar}@inf.unibe.ch

***Abstract*—The ubiquity of smartphones, and their very broad capabilities and usage, make the security of these devices tremendously important. Unfortunately, despite all progress in security and privacy mechanisms, vulnerabilities continue to proliferate.**

Research has shown that many vulnerabilities are due to insecure programming practices. However, each study has often dealt with a specific issue, making the results less actionable for practitioners.

To promote secure programming practices, we have reviewed related research, and identified avoidable vulnerabilities in Android-run devices and the *security code smells* that indicate their presence. In particular, we explain the vulnerabilities, their corresponding smells, and we discuss how they could be eliminated or mitigated during development. Moreover, we develop a lightweight static analysis tool and discuss the extent to which it successfully detects several vulnerabilities in about 46 000 apps hosted by the official Android market.

I. Introduction

Given these premises, the primary goal of this work is to shed light on the root causes of programming choices that compromise users' security. In contrast to previous research that has often dealt with a specific issue, we study this phenomenon from a broad perspective. We introduce the notion of ***security code smells*** *i.e., symptoms in the code that signal the prospect of a security vulnerability*. We have identified avoidable vulnerabilities, their corresponding smells in the code; and discuss how they could be eliminated or mitigated during development. We have also developed a lightweight static analysis tool to look for several of the identified security smells in 46 000 apps. In particular, we answer the following three research questions:

- **RQ**$_1$: What are the security code smells in Android apps? We have reviewed major related work, especially those

2. Tool Selection Process

Literature: Reviewing Benchmarking Process

SCAM 2017

Security Smells in

Mohammad Ghafari, Pascal Gadient, C
Software Composition Group, Univer
Bern, Switzerland
{ghafari, gadient, oscar}@inf.ur

***Abstract*—The ubiquity of smartphones, and their very broad capabilities and usage, make the security of these devices tremendously important. Unfortunately, despite all progress in security and privacy mechanisms, vulnerabilities continue to proliferate.**

Research has shown that many vulnerabilities are due to insecure programming practices. However, each study has often dealt with a specific issue, making the results less actionable for practitioners.

To promote secure programming practices, we have reviewed related research, and identified avoidable vulnerabilities in Android-run devices and the *security code smells* that indicate their presence. In particular, we explain the vulnerabilities, their corresponding smells, and we discuss how they could be eliminated or mitigated during development. Moreover, we develop a lightweight static analysis tool and discuss the extent to which it successfully detects several vulnerabilities in about 46 000 apps hosted by the official Android market.

I. INTRODUCTION

Given th
shed light
compromise
that has oft
nomenon fr
of *security*
the prospec
avoidable v
code; and d
during deve
static analys
smells in 4
three resear

- **RQ**$_1$:
We ha

2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

MUBench: A Benchmark for API-Misuse Detectors

Sven Amann[†] Sarah Nadi[†] Hoan A. Nguyen[‡] Tien N. Nguyen[‡] Mira Mezini[†§]
Technische Universität Darmstadt[†] Iowa State University[‡] Lancaster University[§]
{amann, nadi, mezini}@cs.tu-darmstadt.de, {hoan, tien}@iastate.edu

ABSTRACT

Over the last few years, researchers proposed a multitude of automated bug-detection approaches that mine a class of bugs that we call *API misuses.* Evaluations on a variety of software products show both the omnipresence of such misuses and the ability of the approaches to detect them.

This work presents MUBENCH, a dataset of 89 API misuses that we collected from 33 real-world projects and a survey. With the dataset we empirically analyze the prevalence of API misuses compared to other types of bugs, finding that they are rare, but almost always cause crashes. Furthermore, we discuss how to use it to benchmark and compare API-misuse detectors.

CCS Concepts

•Software and its engineering → Software defect analysis; *Software post-development issues;*

Source	Total Size	Reviewed	Misuse	Crash
BUGCLASSIFY	2,914	294	26	16
DEFECTS4J	357	357	14	12
IBUGS	390	390	56	?
QACRASHFIX	24	24	15	15
SOURCEFORGE	130	130	13	6
GITHUB	2,660	78	3	2
SURVEY	17	17	12	5
Total	6,491	1,189	89	61

Table 1: API Misuses by Source

towards these goals, we present MUBENCH, a dataset of API misuses that can be used to benchmark and compare API-misuse detectors. We explored existing bug datasets, mined projects from SOURCEFORGE and GITHUB, and conducted a survey to collect 89 instances of API misuses. From this sample, we created a taxonomy of API misuses and a dataset

2. Tool Selection Process

Literature: Reviewing Ground Concepts

SCAM 2017

Se

Mo

Abstract—The ubiquity of smartph
capabilities and usage, make the secur
dously important. Unfortunately, desp
and privacy mechanisms, vulnerabilit
Research has shown that many
insecure programming practices. How
dealt with a specific issue, making the
practitioners.
To promote secure programming p
related research, and identified av
Android-run devices and the *security*
their presence. In particular, we e
their corresponding smells, and we
be eliminated or mitigated during d
develop a lightweight static analysis t
to which it successfully detects sever
46 000 apps hosted by the official An

I. INTRODUCT

A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks

Siegfried Rasthofer & Steven Arzt
Secure Software Engineering Group
EC SPRIDE, Technische Universität Darmstadt
{firstname.lastname}@ec-spride.de

Eric Bodden
Secure Software Engineering Group
Fraunhofer SIT & Technische Universität Darmstadt
eric.bodden@sit.fraunhofer.de

***Abstract*—Today's smartphone users face a security dilemma: many apps they install operate on privacy-sensitive data, although they might originate from developers whose trustworthiness is hard to judge. Researchers have addressed the problem with more and more sophisticated static and dynamic analysis tools as an aid to assess how apps use private user data. Those tools, however, rely on the manual configuration of lists of *sources* of sensitive data as well as *sinks* which might leak data to untrusted observers. Such lists are hard to come by.**

We thus propose SuSi, a novel machine-learning guided approach for identifying sources and sinks directly from the code of any Android API. Given a training set of hand-annotated sources and sinks, SuSi identifies other sources and sinks in the entire API. To provide more fine-grained information, SuSi further categorizes the sources (e.g., unique identifier, location information, etc.) and sinks (e.g., network, file, etc.).

For Android 4.2, SuSi identifies hundreds of sources and sinks with over 92% accuracy, many of which are missed by current information-flow tracking tools. An evaluation of about 11,000 malware samples confirms that many of these sources and sinks are indeed used. We furthermore show that SuSi can reliably classify sources and sinks even in new, previously unseen Android versions and components like Google Glass or the Chromecast API.

experience, they also create additional privacy concerns if used for tracking or monitoring.

To address this problem, researchers have proposed various analysis tools to detect and react to data leaks, both statically [1]–[13] and dynamically [14]–[17]. Virtually all of these tools are configured with a privacy policy, usually defined in terms of lists of *sources* of sensitive data (e.g., the user's current location) and *sinks* of potential channels through which such data could leak to an adversary (e.g., a network connection). As an important consequence, no matter how good the tool, it can only provide security guarantees if its list of sources and sinks is complete. If a source is missing, a malicious app can retrieve its information without the analysis tool noticing. A similar problem exists for information written into unrecognized sinks.

This work focuses on Android. As we show, existing analysis tools, both static and dynamic, focus on a handful of hand-picked sources and sinks, and can thus be circumvented by malicious applications with ease. It would be too simple, though, to blame the developers of those tools. Android's version 4.2, for instance, comprises about 110,000 public methods, which makes a manual classification of sources and sinks clearly infeasible. Furthermore, each new Android version includes new functionality (e.g., NFC in Android 2.3 or Restricted Profiles

positories

e Detectors

lguyen Mira Mezini

ster University
@iastate.edu

ze	Reviewed	Misuse	Crash
14	294	26	16
57	357	14	12
90	390	56	?
24	24	15	15
30	130	13	6
50	78	3	2
17	17	12	5
91	1,189	89	61

Misuses by Source

sent MuBench, a dataset of API
to benchmark and compare API-
ored existing bug datasets, mined
GE and GitHub, and conducted
ances of API misuses. From this
omy of API misuses and a dataset

2. Tool Selection Process

Literature: Reviewing Different Tools

866 IEEE TRANSACTIONS ON SOFTWARE ENGIN

COVERT: Compositional Analys
Inter-App Permission Le

2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications

AppCaulk: Data Leak Prevention by Injecting Targeted Taint Tracking Into Android Apps

Julian Schütte, Dennis Titze, and J. M. de Fuentes
{schuette,titze}@aisec.fraunhofer.de, jfuentes@inf.uc3m.es

Abstract

As Android is entering the business domain, leaks of business-critical and personal information through apps become major threats. Due to the context-insensitive nature of the Android permission model, information flow policies cannot be enforced by on-board mechanisms. We therefore propose AppCaulk, an approach to harden any existing Android app by injecting a targeted dynamic taint analysis, which tracks and blocks unwanted information flows at runtime. Critical data flows are first discovered using a static taint analysis and the relevant data propagation paths

To cope with information leaks, several approaches have been proposed and some practically applicable solutions exist. Most of them refer to container-based approaches where either applications are wrapped in a “security container” or domains are isolated at kernel level (see [13], [4]). These approaches are however *context-free*, as they do not keep track of individual data flows but rather apply a perimeter security, either at API or OS level.
Dynamic taint analysis, in contrast, monitors how data is handled by an application and detects when an unwanted flow from a specific data source (e.g., the address book) to a specific sink (e.g., a socket) is

2016 IEEE European Symposium on Security and Privacy

HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving

Stefano Calzavara, *Università Ca’ Foscari Venezia*
Ilya Grishchenko, *CISPA, Saarland University*
Matteo Maffei, *CISPA, Saarland University*

2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Composite Constant Propagation: Application to Android Inter-Component Communication Analysis

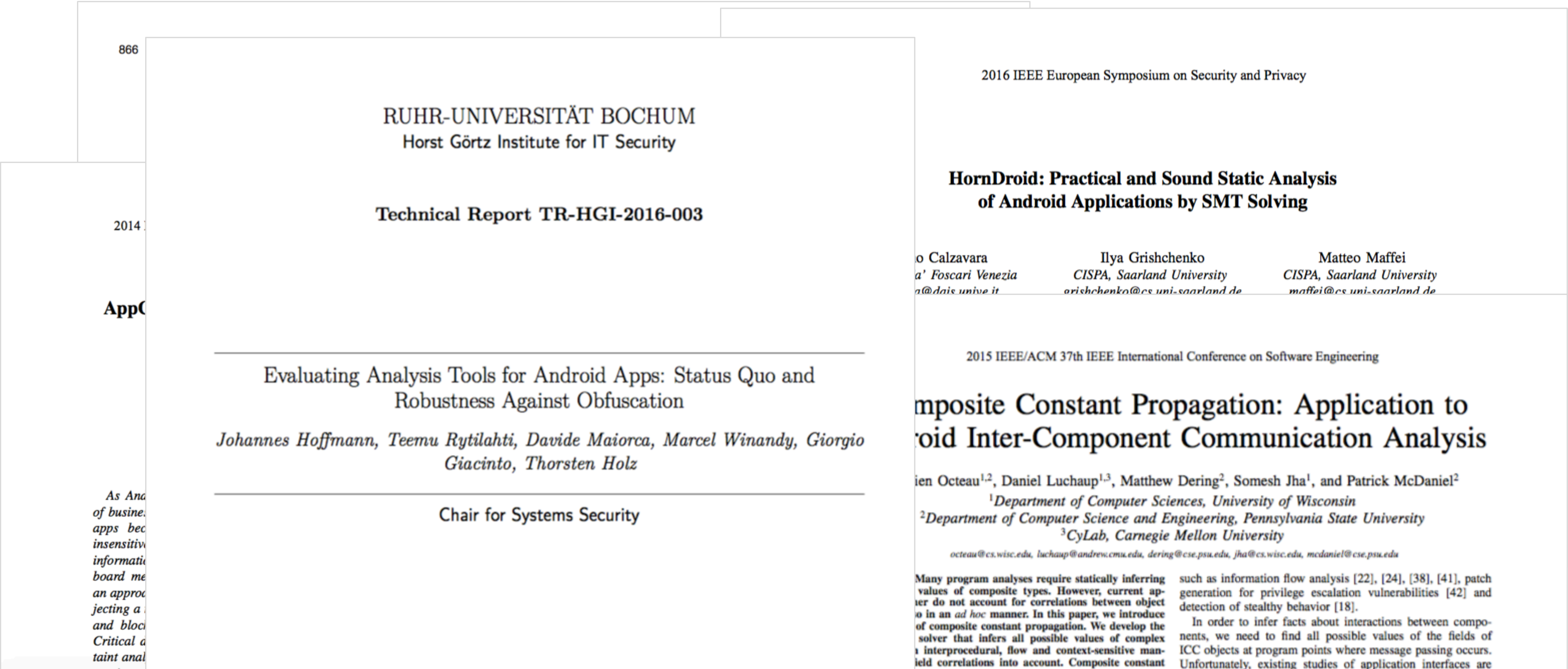
Damien Octeau[1,2], Daniel Luchaup[1,3], Matthew Dering[2], Somesh Jha[1], and Patrick McDaniel[2]
[1]*Department of Computer Sciences, University of Wisconsin*
[2]*Department of Computer Science and Engineering, Pennsylvania State University*
[3]*CyLab, Carnegie Mellon University*
octeau@cs.wisc.edu, luchaup@andrew.cmu.edu, dering@cse.psu.edu, jha@cs.wisc.edu, mcdaniel@cse.psu.edu

***Abstract*—Many program analyses require statically inferring the possible values of composite types. However, current approaches either do not account for correlations between object fields or do so in an *ad hoc* manner. In this paper, we introduce the problem of composite constant propagation. We develop the first generic solver that infers all possible values of complex objects in an interprocedural, flow and context-sensitive manner, taking field correlations into account. Composite constant**

such as information flow analysis [22], [24], [38], [41], patch generation for privilege escalation vulnerabilities [42] and detection of stealthy behavior [18].
In order to infer facts about interactions between components, we need to find all possible values of the fields of ICC objects at program points where message passing occurs. Unfortunately, existing studies of application interfaces are

2. Tool Selection Process

Literature: Reviewing Robustness Of Tools



866

2014

App(

As Ana
of busine
apps bec
insensitiv
informati
board me
an approa
jecting a
and bloci
Critical a
taint anal

RUHR-UNIVERSITÄT BOCHUM
Horst Görtz Institute for IT Security

Technical Report TR-HGI-2016-003

Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation

Johannes Hoffmann, Teemu Rytilahti, Davide Maiorca, Marcel Winandy, Giorgio Giacinto, Thorsten Holz

Chair for Systems Security

2016 IEEE European Symposium on Security and Privacy

HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving

o Calzavara — *a' Foscari Venezia*
Ilya Grishchenko — *CISPA, Saarland University*
Matteo Maffei — *CISPA, Saarland University*

2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

mposite Constant Propagation: Application to
roid Inter-Component Communication Analysis

ien Octeau[1,2], Daniel Luchaup[1,3], Matthew Dering[2], Somesh Jha[1], and Patrick McDaniel[2]
[1]*Department of Computer Sciences, University of Wisconsin*
[2]*Department of Computer Science and Engineering, Pennsylvania State University*
[3]*CyLab, Carnegie Mellon University*
octeau@cs.wisc.edu, luchaup@andrew.cmu.edu, dering@cse.psu.edu, jha@cs.wisc.edu, mcdaniel@cse.psu.edu

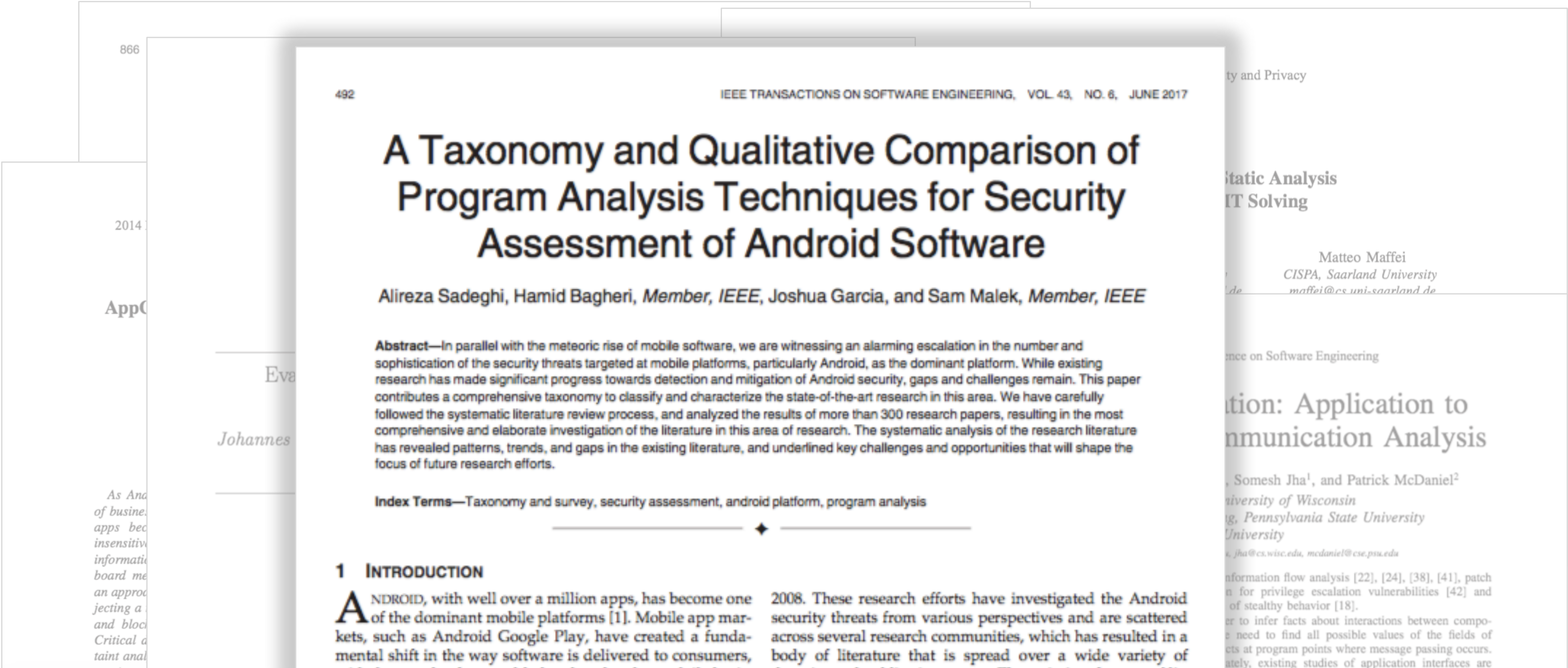
Many program analyses require statically inferring values of composite types. However, current ap- er do not account for correlations between object o in an *ad hoc* manner. In this paper, we introduce of composite constant propagation. We develop the solver that infers all possible values of complex interprocedural, flow and context-sensitive man- ield correlations into account. Composite constant

such as information flow analysis [22], [24], [38], [41], patch generation for privilege escalation vulnerabilities [42] and detection of stealthy behavior [18].

In order to infer facts about interactions between components, we need to find all possible values of the fields of ICC objects at program points where message passing occurs. Unfortunately, existing studies of application interfaces are

2. Tool Selection Process

Literature: Major Contribution



492 IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 43, NO. 6, JUNE 2017

A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software

Alireza Sadeghi, Hamid Bagheri, *Member, IEEE*, Joshua Garcia, and Sam Malek, *Member, IEEE*

Abstract—In parallel with the meteoric rise of mobile software, we are witnessing an alarming escalation in the number and sophistication of the security threats targeted at mobile platforms, particularly Android, as the dominant platform. While existing research has made significant progress towards detection and mitigation of Android security, gaps and challenges remain. This paper contributes a comprehensive taxonomy to classify and characterize the state-of-the-art research in this area. We have carefully followed the systematic literature review process, and analyzed the results of more than 300 research papers, resulting in the most comprehensive and elaborate investigation of the literature in this area of research. The systematic analysis of the research literature has revealed patterns, trends, and gaps in the existing literature, and underlined key challenges and opportunities that will shape the focus of future research efforts.

Index Terms—Taxonomy and survey, security assessment, android platform, program analysis

1 Introduction

Android, with well over a million apps, has become one of the dominant mobile platforms [1]. Mobile app markets, such as Android Google Play, have created a fundamental shift in the way software is delivered to consumers, ...

2008. These research efforts have investigated the Android security threats from various perspectives and are scattered across several research communities, which has resulted in a body of literature that is spread over a wide variety of ...

2. Tool Selection Process

Focus On Vulnerability Detection

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

- Not Found

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

- No Tools

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

- Not Reachable Researcher

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab’nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

- Access Refused

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

- Unresponsive Researcher

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab’nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

+ Responsive Researcher

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

Focus On Information Disclosure

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

And The Winners Are…

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

… But, Remove Tools That Cannot Be Setup

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab'nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

2. Tool Selection Process

… And Those That Cannot Be Analysed

ADDICTED, Amandroid, ApkCombiner, App-Ray, AppAudit, AppCaulk, AppCracker, AppFence, AppGuard, AppProfiler, AppSealer, Aquifer, ASM, AuthDroid, Bagheri, Bartel, Bartsch, Bifocals, Buhov, Buzzer, CMA, CoChecker, ComDroid, ConDroid, ContentScope, Cooley, COPES, COVERT, CredMiner, CRePE, CryptoLint, Desnos, DexDiff, DroidAlarm, DroidChecker, DroidCIA, DroidGuard, DroidRay, Droidsearch, Enck, Epicc, FineDroid, Flowdroid, Gallo, Geneiatakis, Grab’nRun, Harehunter, HornDroid, IccTA, IPCInspection, IVDroid, Juxtapp, Kantola, KLD, Lintent, Lu, MalloDroid, Matsumoto, Mutchler, NoFrak, NoInjection, Onwuzurike, PaddyFrog, PatchDroid, PCLeaks, PermCheckTool, PermissionFlow, Poeplau, Pscout, QUIRE, Ren, SADroid, SCanDroid, Scoria, SecUP, SEFA, Smith, SMV-HUNTER, STAMBA, Stowaway, SUPOR, TongxinLi, Vecchiato, VetDroid, WeChecker, Woodpecker, Zuo

3. Selected Tools In A Nutshell

In A Nutshell – Pretty Much The Same

	COVERT	Flowdroid	IccTA	IC3 (Epicc)	Horndroid
Type:	Static & Formal	Static	Static	Static	Static & Formal
Artefact:	Manifest	Manifest Code (native)	Manifest Layout	Manifest	Code (reflective)
Data Structure:	Call Graph CFG ICFG	Call Graph CFG ICFG	Call Graph CFG ICFG	Call Graph CFG ICFG	N/A
Code Representation	Jimple	Jimple	Jimple	Jimple	N/A
Sensitivity	Flow Context	Flow Context	Flow Context	Flow Context	N/A

[1] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek. "A Taxonomy and qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Software"

3. Selected Tools In A Nutshell

Results Hard To Find, To Read And To Understand…


```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<analysisReport>
    <name>apksToTest</name>
    <apps/>
    <vulnerabilities>
        <vulnerability>
            <type>Intent Spoofing</type>
            <description>App org.cert.sendsms puts data (retrieved from an Explicit Intent (Component
                = MainActivity)) on an unsafe sink (SMS_MMS) in one of its components
                (org.cert.sendsms.MainActivity). A malicious app can send a sensitive data from this
                channel.</description>
            <vulnerabilityElements>
                <type>APP</type>
                <description>org.cert.sendsms</description>
                <element>
                    <type>COMPONENT</type>
                    <description>org.cert.sendsms.MainActivity</description>
                    <element>
                        <type>INTENT</type>
                        <description>Explicit Intent (Component = MainActivity)</description>
                        <alloyLabel>i2</alloyLabel>
                    </element>
                    <element>
                        <type>METHOD</type>
                        <description>org.cert.sendsms.MainActivity: void
                            onActivityResult(int,int,android.content.Intent)</description>
                        <element>
                            <type>SINK_TYPE</type>
                            <description>SMS_MMS</description>
```


COVERT
.xml file

3. Selected Tools In A Nutshell

Results Hard To Find, To Read And To Understand…

Flowdroid Console output

```
Running data flow analysis...
Found dex file 'classes.dex' with 456 classes in '/Users/timospring/Desktop/droid-Security-Thesis/apk_sample/
validation_apk/SendSMS.apk'
[Call Graph] For information on where the call graph may be incomplete, use the verbose option to the cg phase.
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.1 seconds.
[Spark] Solution found in 0.1 seconds.
Callback analysis done.
Found 1 layout controls in file res/layout/activity_main.xml
[Call Graph] For information on where the call graph may be incomplete, use the verbose option to the cg phase.
[Spark] Pointer Assignment Graph in 0.0 seconds.
[Spark] Type masks in 0.0 seconds.
[Spark] Pointer Graph simplified in 0.0 seconds.
[Spark] Propagation in 0.0 seconds.
[Spark] Solution found in 0.0 seconds.
Running incremental callback analysis for 1 components...
Incremental callback analysis done.
Found a flow to sink virtualinvoke $r3.<org.cert.sendsms.MainActivity: void
startActivityForResult(android.content.Intent,int)>($r2, 0), from the following sources:
        - $r6 = virtualinvoke $r5.<android.telephony.TelephonyManager: java.lang.String getDeviceId()>() (in
<org.cert.sendsms.Button1Listener: void onClick(android.view.View)>)
Found a flow to sink virtualinvoke $r3.<android.telephony.SmsManager: void
sendTextMessage(java.lang.String,java.lang.String,java.lang.String,android.app.PendingIntent,android.app.PendingIn
tent)>("1234567890", null, $r1, null, null), from the following sources:
        - $r1 := @parameter2: android.content.Intent (in <org.cert.sendsms.MainActivity: void
onActivityResult(int,int,android.content.Intent)>)
Found a flow to sink staticinvoke <android.util.Log: int i(java.lang.String,java.lang.String)>("SendSMS: ", $r6),
from the following sources:
        - $r6 = virtualinvoke $r5.<android.telephony.TelephonyManager: java.lang.String getDeviceId()>() (in
<org.cert.sendsms.Button1Listener: void onClick(android.view.View)>)
Analysis has run for 6.296909903 seconds
```

3. Selected Tools In A Nutshell

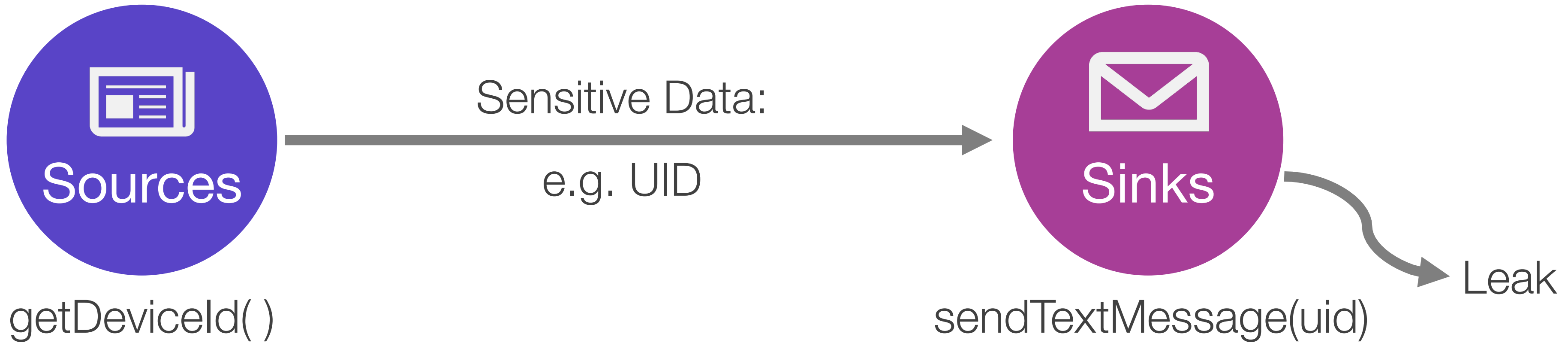
Results Hard To Find, To Read And To Understand…

```
PendingIntent;Landroid/app/PendingIntent;)V:NO LEAK
2018-23-20 17:23:29.525 [main] INFO  com.horndroid.z3.FSEngine - 11 [REF] Test if register 0 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:POTENTIAL LEAK
2018-23-20 17:23:31.333 [main] INFO  com.horndroid.z3.FSEngine - 12 [REF] Test if register 1 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:POTENTIAL LEAK
2018-23-20 17:23:31.500 [main] INFO  com.horndroid.z3.FSEngine - 13 [REF] Test if register 2 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:NO LEAK
2018-23-20 17:23:33.623 [main] INFO  com.horndroid.z3.FSEngine - 14 [REF] Test if register 3 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:POTENTIAL LEAK
2018-23-20 17:23:33.806 [main] INFO  com.horndroid.z3.FSEngine - 15 [REF] Test if register 4 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:NO LEAK
2018-23-20 17:23:33.970 [main] INFO  com.horndroid.z3.FSEngine - 16 [REF] Test if register 5 leaks at
line 11 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink sendTextMessage(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
PendingIntent;Landroid/app/PendingIntent;)V:NO LEAK
2018-23-20 17:23:36.660 [main] INFO  com.horndroid.z3.FSEngine - 17 Test if register 6 leaks at line 43
in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the sink
printStackTrace()V:NO LEAK
2018-23-20 17:23:40.795 [main] INFO  com.horndroid.z3.FSEngine - 18 [REF] Test if register 6 leaks at
line 43 in method sendSMSMessage(Ljava/lang/String;)V of the class Lorg/cert/sendsms/MainActivity; to the
sink printStackTrace()V:NO LEAK
2018-23-20 17:23:40.998 [main] INFO  com.horndroid.z3.FSEngine - 19 Test if register 1 leaks at line 30
in method onActivityResult(IILandroid/content/Intent;)V of the class Lorg/cert/sendsms/MainActivity; to
the sink v(Ljava/lang/String;Ljava/lang/String;)I:NO LEAK
```

Horndroid .log file

3. Selected Tools In A Nutshell

It's All About Sources And Sinks



Is it only a question of who has the best sources and sinks list?

3. Selected Tools In A Nutshell

Own Implementation Runs Tools And Parses Output

Component:	`android.util.Log`
Class:	`org.cert.sendsms.Button1Listener`
Method:	`void onClick(android.view.View)`
Line:	`25`
Detected by:	`flowdroid, iccta`

Problem: Only class and method are reported by all tools

4. Case Study: SendSMS App

Run tools on app with known vulnerabilities

- SendsSMS.apk with known inter-app communication vulnerabilities
- Gets the UID and sends it over SMS and writes it to log file

4. Case Study: SendSMS App

App Leaks The UID Over SMS And To The Log File

Component: Button1Listener

```
onClick(View arg0)
    String uid = tManager.getDeviceId(); // SOURCE
    Log.i("SendSMS: ", "DeviceId "+uid); // SINK
    this.act.startActivityForResult(i, 0); // SINK
```

Component: MainActivity

```
sendSMSMessage(String message)
    smsManager.sendTextMessage("1234567890", message); //SINK
```

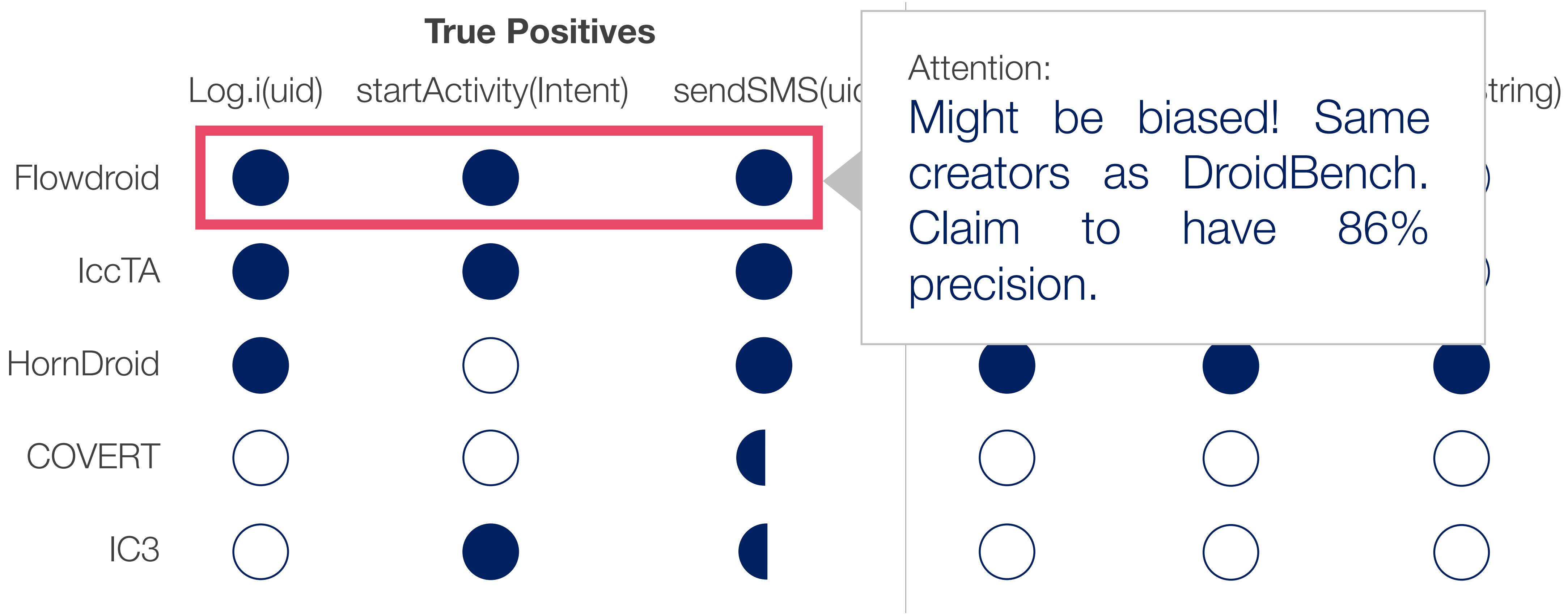
4. Case Study: SendSMS App

Especially HornDroid Seems To “Over-Report”

	True Positives			False Positives		
	Log.i(uid)	startActivity(Intent)	sendSMS(uid)	Log.v(String)	Log.i(String)	Log.i(String)
Flowdroid	●	●	●	○	○	○
IccTA	●	●	●	○	○	○
HornDroid	●	○	●	●	●	●
COVERT	○	○	◐	○	○	○
IC3	○	●	◐	○	○	○

4. Case Study: SendSMS App

Flowdroid Might Be Biased



Attention:

Might be biased! Same creators as DroidBench. Claim to have 86% precision.

4. Case Study: SendSMS App

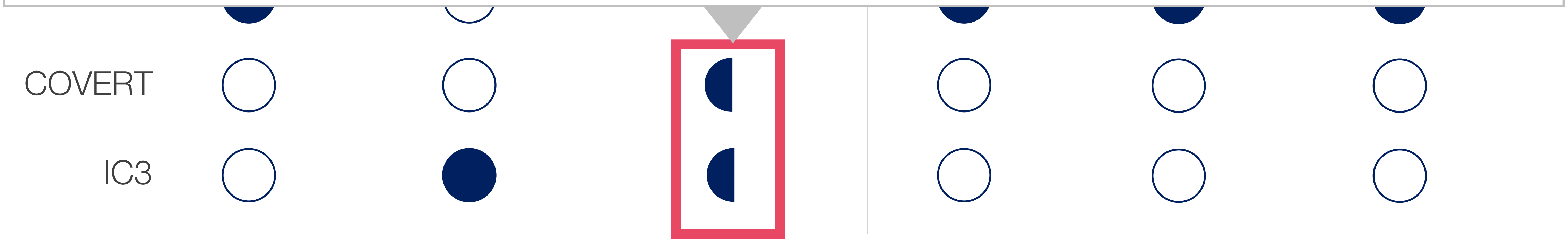
Vulnerability Detected, But Not As Precise As Others

Report:

```
onActivityResult(int, int, Intent)
37      sendSMSMessage(data.getExtras().getString("secret"));
```

Instead of:

```
sendSMSMessage(String message)
52      smsManager.sendTextMessage("1234567890", null, message, null, null); //SINK
```

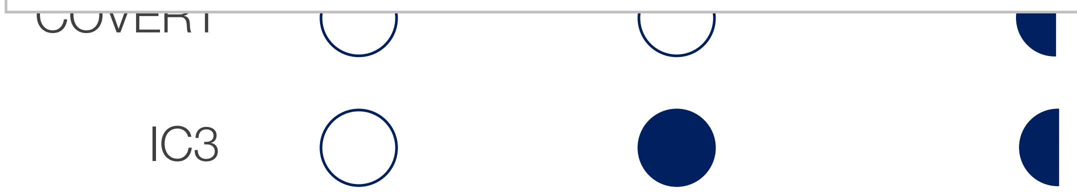


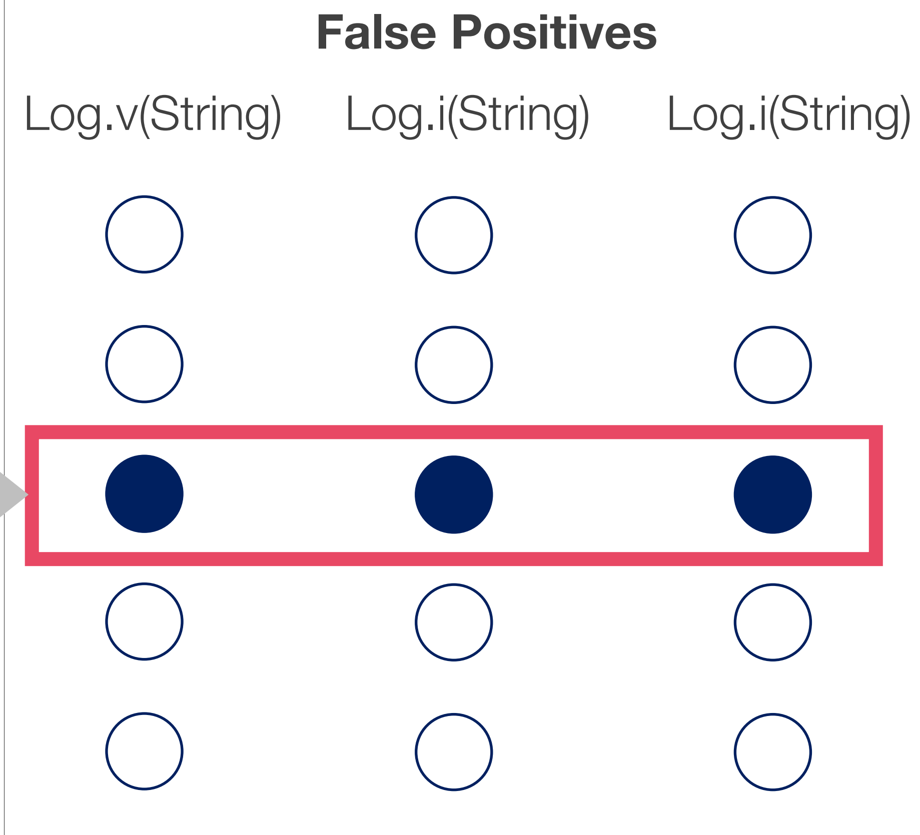
4. Case Study: SendSMS App

HornDroid Is The Only Tool Reporting False Positives

Report:

```
onActivityResult(int, int, Intent)
36      Log.v("In SendSMS: ", "Data
            received");
…
40      Log.i("In SendSMS: ", "Data
            received");
…
44      Log.i("In SendSMS: ", "No data
            received”);
```





4. Case Study: SendSMS App

There Are Differences In What The Tools Report

- FlowDroid and IccTA found all leaks without false alarm
- HornDroid found most leaks, but reports many false positives
- COVERT and IC3 only found part of the leaks

! **Problem:** Analysis for false positives not scalable!

5. Benchmarking Concept

How To Include Both Worlds

Small scale qualitative

- DroidBench dataset (~30 apps)
- Manually check for false positives and false negatives

Large scale quantitative

- F-Droid dataset (~2.6k apps)
- Automatically analyse number of detections and matchings

6. Lessons Learned So Far

It’s Tricky!

- Hundreds of tools, but only few are actually available and can be setup
- Tools are not user-friendly and results poorly documented
- All of them claim to be the best – we’ll see about that …

7. Outlook

What's Next?

- Fine tune automatic analysis (refactoring)
- Check DroidBench dataset
- Run automatic analysis on F-droid dataset
- Evaluation of results (quantitative)
- Draw conclusions