Informatics

Proof-of-Concept of a Static Analysis Tool for Android Applications with the Goal of Detecting Potential Leaks of Private Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ismar Music
Matrikelnummer 01328053

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 1. Februar 2021

Unterschrift Verfasser

Unterschrift Betreuung

Informatics

Proof-of-Concept of a Static Analysis Tool for Android Applications with the Goal of Detecting Potential Leaks of Private Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Ismar Music
Registration Number 01328053

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig

Vienna, 1st February, 2021

Signature Author

Signature Advisor

Proof-of-Concept of a Static Analysis Tool for Android Applications with the Goal of Detecting Potential Leaks of Private Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ismar Music
Matrikelnummer 01328053

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 1. Februar 2021

Erklärung zur Verfassung der Arbeit

Ismar Music

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Februar 2021

Ismar Music

Kurzfassung

Android ist ein Betriebssystem, welches auf Millionen von Geräten läuft. Die Geräte speichern und verarbeiten private und sensible Daten wie GPS-Daten, Kontakte, Gesundheitsdaten oder Bankdaten. Diese Daten können ohne Zustimmung der Benutzer absichtlich oder unbeabsichtigt an Dritte weitergegeben werden. Dies geschieht durch verschiedene Methoden wie: Kommunikation zwischen Apps (Inter-App Communication), Kommunikation zwischen Komponenten (Inter-Component Communication) und durch Bibliotheken von Drittanbietern. Diese Arbeit untersucht die neuesten Lösungen zur Erkennung von Datenlecks und stellt eine Verbesserung gegenüber einer der vorhandenen Lösungen, FlowDroid, dar.

In dieser Arbeit werden zunächst die Sicherheitslücken erörtert, die unter Android zu Datenlecks führen. Die häufigsten Sicherheitslücken werden weiter analysiert, damit sie durch die Proof-of-Concept-Lösung gemindert werden können. Nach den Sicherheitslücken werden die vorhandenen Lösungen und Algorithmen, welche diese Lösungen unterstützen, vorgestellt, die wiederum als Grundlage für die Proof-of-Concept-Lösung dienen. Die Arbeit identifiziert und verbessert die vorhandenen Lösungen. Basierend auf der Forschung des neuesten Stands der Technik wird eine statische Code-Analyse-Lösung entwickelt, die auf zwei vorhandenen Lösungen, DroidRa und FlowDroid, basiert ist, mit dem Ziel, die Funktionen von FlowDroid zur Erkennung von Lecks in dynamisch geladenem Code zu verbessern.

Die Endergebnisse zeigen Verbesserungen in Leistung, da die Proof-of-Concept-Lösung durchschnittlich 3-8% weniger Zeit und durchschnittlich 10-15% weniger Speicher pro Testfall benötigt. Es wird auch gezeigt, dass nicht nur die Verbesserung der Analyseverfahren wichtig ist, sondern auch die Optimierung der richtigen Eingabedaten, nämlich Quellen und Senken (Sources and Sinks) für die Analyseergebnisse wichtig ist. Die Genauigkeit und die Trefferquote der Lösung können, abhängig von den Eingabedaten, um ±20-30% verschoben werden. Die Analyse von echten Apps zeigt auch, dass die Verschleierung für statische Analyselösungen immer noch ein Problem darstellt.

Keywords: *Android OS, Datenschutz, Datenschutzverbessernde Anwendungen, Malware, Android-Sicherheit, Überwachung.*

Abstract

Android is a platform running on millions of devices. The devices store and process private and sensitive data such as GPS data, contacts, health data or bank data. This data can be intentionally or unintentionally leaked to third parties without the users' consent through various methods like Inter-App-Communication, Inter-Component Communication and Third-party library leaks. This thesis researches the state-of-the-art data leak detection solutions and presents an improvement to one of the existing solutions, FlowDroid.

This thesis first discusses the vulnerabilities that cause data leaks on Android and selects the most common ones as the target which the solution aims to mitigate. After vulnerabilities, the existing solutions and algorithms that support these solutions are presented, which are in turn used as a basis for the proof-of-concept solution. The thesis tries to identify places for improvement in the existing solutions and iterate on them. The goal is to develop an improved solution, which uses static code analysis. Based on the research of the state-of-the-art a static code analysis solution based on two existing solutions, DroidRa and FlowDroid is developed, with the goal of enhancing FlowDroid's capabilities to detect leaks in dynamically loaded code.

The final results show improvements in performance, with the proof-of-concept solution taking around 3-8% less time and 10-15% less memory on average per test case. We also show that not only improving the analysis is important, providing the right set of sources and sinks on Android is also important. The differences in these input files can shift the precision and recall of the solution ±20-30%. Real-world application analysis also shows that obfuscation is still a problem for static-analysis solutions.

Keywords: *Android OS, Privacy, Privacy enhancing applications, Malware, Android Security, Surveillance.*

Contents

CHAPTER 1
Introduction

1.1 Problem Description

Smartphones are a part of everyday lives for millions of people due to the diversity of applications on the platform. As there are apps for everything from checking the time and navigation to banking, these smartphones collect and process sensitive data, such as: passwords, PINs, bank card information, location and health data. The lack of transparency about where this data goes, e.g. if it is uploaded somewhere, and how it is further used causes privacy concerns. Therefore, privacy preservation and security have to be parts of every mobile platform.

Android is currently the biggest mobile platform with approximately 73% of the market share[9], and as such it is a target for exploits. While there currently are system mechanisms of Android that help with preventing exploits (e.g. the permission manager), as well as third-party applications helping to limit applications' access to sensitive data (e.g. XPrivacyLua[10]), the vulnerabilities such as privilege escalation and Inter-Component-Communication (ICC) still exist and can lead to data leaks[19].

A data leak in Android could be defined as any data being sent outside of an application without user's consent, with consent being defined as allowing permission to access a resource[57]. These leaks can be both unintentional (e.g. bugs, design flaws) or intentional (e.g. malicious, for profit). If a leak is malicious, leaked passwords and PINs can be used to steal directly from users' bank accounts, while leaked information such as various usage data, location data, sensor data etc. can be collected and sold to third parties which can make use of it, for example: for personalized advertising[51].

To protect against such leaks, detection tools are needed. As it stands, there is no state-of-the-art solution that solves the problem of detecting potential data leaks in Android applications. There are tools such as FlowDroid[15] by Arzt et al. which is a frequently updated data flow analysis tool available freely for researchers and developers. Other solutions proposed in academic research are not available publicly. There are no tools available for Android that can be installed from the Play Store.

Apart from the availability issues, the already available leak detection tools fail to achieve full leak detection. FlowDroid[15] achieves 86% precision and 65% recall in detecting data leaks. That means, that it fails to recognize 35% of existing leaks.

1.2 Expected Results

The goal of this thesis is to develop a solution that can analyze Android Package (APK) files and determine whether the application leaks data. The intent behind the leak is not the focus of the thesis and any leak will be treated as harmful for the user. The end result is a solution that uses an application APK as input and gives a list of potential data leaks as output.

Either a fully independent solution is going to be developed, or a solution that is supported by previous research on the topic, as there are many approaches that have been introduced.

The following research questions have been derived:

- Which vulnerabilities that leak private data exist on Android?
- How do these vulnerabilities function?
- Is there a theoretical way to determine whether an application leaks data before it is installed (analyzing the APK)?
- If yes, can the approach be implemented to work on any real-world application?
- If it can be implemented, how effective is it in practice?

Not only will the final implementation deliver results, but every part of the thesis will deliver intermediate results, including: an overview of current private data leaking vulnerabilities; a collection of methods, techniques and architectures used in previous research, as well as their issues and shortcomings, that will build a foundation for the rest of the thesis; a list of vulnerabilities not addressed by previous research; a concrete architecture, technology stack, algorithms and the full design layout of the solution; an evaluation-ready implementation of the architecture; the answer to the main thesis questions, as well as performance details. In the case that no solution gets developed, the thesis will discuss why it was not possible to develop such a solution.

1.3 Methodological Approach

Each of the parts mentioned in *Expected Results* has its own method:

The initial stage of the thesis will be a literature overview of state-of-the-art research. This will show which vulnerabilities that cause sensitive data leaks exist within Android OS, as well as show which solutions already exist. The problems and limitations of the existing solutions will also be shown. Making an overview of the largest security concerns on Android and how they relate to potential data leaks and privacy violations will show which vulnerabilities exist, and which are most commonly used, in order to determine which of these vulnerabilities should the thesis aim to detect.

A score will be assigned to vulnerabilities based on how they affect user privacy. The vulnerabilities with the highest scores will be the target of the thesis.

Researching the developed solutions will show what kind of approaches are used and which problems and limitations are most commonly encountered, in order to see what can be done to circumvent the same issues and how to improve the detection of the data leaks caused by the already covered vulnerabilities.

The next stage is focused on research of the best technologies for the purpose of detecting data leaks which are caused by the selected vulnerabilities. The causes of the exploits and how they work will be discussed here. By reverse-engineering the exploits, algorithms that find an instance of an exploit within the source code of an application will be developed. Apart from the technologies, the architecture design of the solutions will be made in this stage with the help of prototyping tools, UML-diagrams and wireframing. Algorithms needed for a successful detection of leaks are also researched here.

The final stage starts with implementing a proof-of-concept solution by using the architecture defined in the previous stage.

In case of any obstacles on the way to the usable solution, adjustments will be made in an attempt to create the best possible solution. The issues, if they occur, will be discussed in this stage.

After implementation, the solution will be evaluated. The evaluation will be done using standardized preexisting test suites (e.g. DroidBench[15]). The solution will be tested on real-world applications (i.e. finding new or known leaks in popular applications), as well as applications used exclusively for testing (e.g. Android malware library or benchmark tools). Tools like DroidBench will be used to create an evaluation that can be compared to other results since they cover most known data leak exploits and they have been used by other solutions for evaluation. The final stage will be a comparison of the evaluation results with the existing solutions and a discussion of the main differences.

1.4 Thesis Structure

The rest of the thesis is organized as follows: Chapter 2 focuses on explaining the basics of Android OS, privacy, and data leaks in general through literature research. Chapter 3 researches the current state-of-the-art in regards to data leak detection solutions, showcases which vulnerabilities are covered by these solutions and what their shortcomings are. Chapter 3 also features an analysis of the vulnerabilities and sets the selection of vulnerabilities that are covered by the proof-of-concept solution.

Chapter 4 researches which types of algorithms are adequate for data leak detection, static code analysis and taint analysis. Based on the findings, the overall proof-of-concept solution architecture is proposed in its basic format. Chapter 5 gives a final system overview that is implemented in the proof-of-concept solution and gives a detailed overview of the implementation and each of its components. The solution is thoroughly evaluated and benchmarked in Chapter 6 and the results are analyzed and compared to other solutions. Furthermore, this chapter explains the drawbacks and limitations of the solution.

The thesis ends with a brief discussion and a conclusion, as well as an outlook into future work in Chapter 7.

CHAPTER 2

Fundamentals

To understand the rest of the thesis, some basic concepts need to be introduced, including: What is Android OS and how it works on a software basis, which is explained in Section 2.1. This is needed in order to understand how the vulnerabilities discussed later on are able to exist. The concept of data privacy and why it is needed is discussed in Section 2.2. After that Section 2.3 explains the severity of the data leak issue in general and how it relates to the concept of data privacy explained in Section 2.2.

When it comes to concrete vulnerabilities discussed in this thesis, the references will only go as far back as 2015, since the version of Android launched in 2015, Android 5.0 "Marshmallow", introduced the permission manager. When it comes to research about about privacy, data leaks, and the architecture of Android in general, no limitations regarding the time frame were set, however, resources which are outdated are discarded.

2.1 Android OS

Android is an operating system which runs the Linux kernel, designed for mobile devices. The core of Android is the Android Open Source Project (AOSP), published under the Apache License. Since it's an open source project it is not owned by anyone, but the development is done by the Open Handset Alliance (OHA), to which the largest contributor is Google[70]. The applications developed for Android are either written in Java or more recently, Kotlin, which compiles to Java bytecode. As a platform Android is open. Any manufacturer can join the OHA and any developer can publish an application on the Play Store. This characteristic of the system allows device manufacturers to use various hardware to make a device for Android, which gives consumers choice when purchasing a mobile device[70].

2.1.1 Android Architecture Overview

The basic architecture of the system consists of five layers, as seen in Figure 2.1 explained in [70],[43] and [3]:

1. **System Applications**: Core applications pre-installed on the device, as well as third-party applications.
2. **Java API Framework**: All features of Android are made available by the Java APIs. They provide core elements needed for applications by simplifying the reuse of system components and services, such as: The *View System*, used to create the UI; the *Resource Manager*, used for non-code resources (e.g. text, graphics, layout files); the *Notification Manager* used for alerts; the *Activity Manager*, which manages the lifecycle of applications and navigation.
3. a) **Native C/C++ Libraries**: Core Android elements, such as Runtime and the Hardware Abstraction Layer are built in native code, which needs C and C++ libraries. Android provides access to these libraries through Java APIs, however, some of them can also be used directly if the developed application requires native C/C++ code.

   b) **Android Runtime (ART)**: Android also comes with core libraries that allow for functionality of most core Java libraries. Every application runs its own process and its own virtual machine (VM) instance. It falls in the same layer as native C/C++ libraries, as they both support Java API Framework, and interface with the Hardware Abstraction Layer.
4. **Hardware Abstraction Layer (HAL)**: HAL is the provider of standard interfaces which expose hardware functionality to higher layers. It consists of multiple modules, each of which has an interface for a hardware component.
5. **Linux Kernel**: Android uses the Linux kernel for core system services like threading, memory management, security, process management, network stack, driver model, as well as for abstraction between the hardware and the software layers.

As explained in [43] by Lettner et al. and in [42] by Latifa and Ahmed, the Dalvik VM (i.e. the virtual machine found in the Runtime layer) executes Android applications written in Java or Kotlin which were first compiled into Dalvik Executable format (*.dex*). The VM itself is optimized for minimal memory consumption. The memory consumption is managed automatically by the Dalvik garbage collector (GC). Each process in the application uses a separate GC instance and employ a *mark-and-sweep* algorithm to remove bits that are unreachable (unused) by the application.

2.1.2 Applications on Android

One of the key features of Android is that any application can use the components of another application. To be able to do that, the application must be able to run from

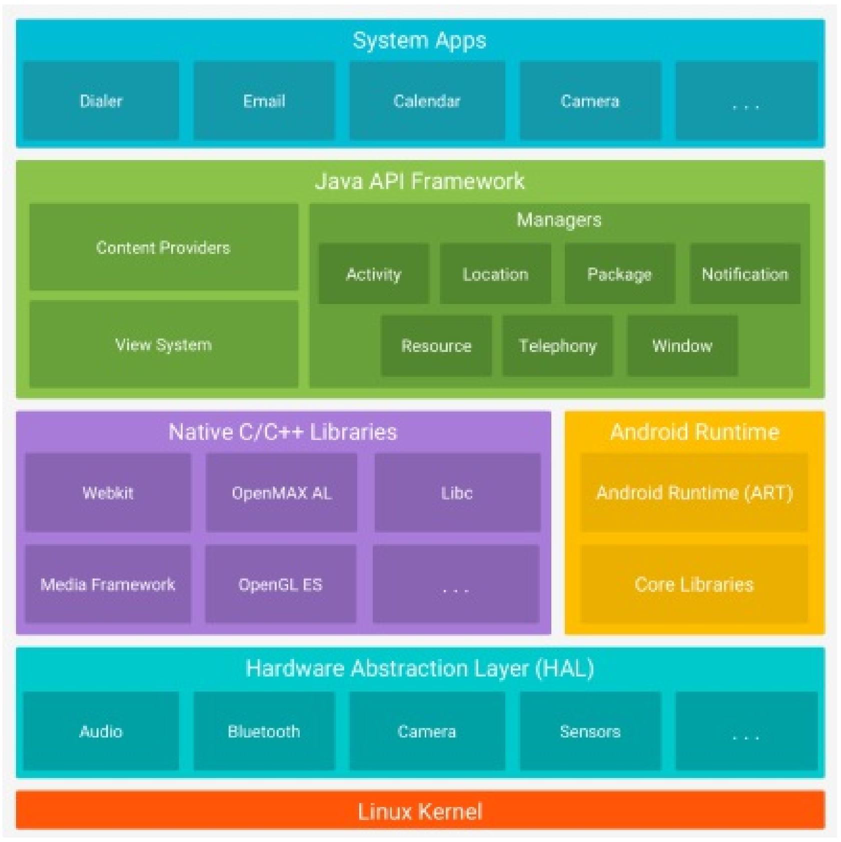


Figure 2.1: Android Architecture Layers (Modified from [3])

multiple entry points (e.g. an E-Mail application can be opened through the home screen which opens the inbox, or through a share menu from another application which opens an already filled in E-Mail draft ready for sending). Android applications have four types of components: **Activities**, **Services**, **Broadcast receivers** and **Content providers**[70].

1. **Activities**: The basic part of an application. Each application consists of one or more independent activities which may or may not communicate with each other. Each of them has a default window to draw in. The visual content of each activity is provided by a view hierarchy.

2. **Services**: The background activities, which do not have a visual user interface, but rather run in the background for a set time. Services, as well as activities run in the main thread of an application, and if they are time-intensive, spawn a separate thread for the task.

3. **Broadcast receivers**: The receivers that take an action request and provide a response. Each application has a number of receivers to respond to actions defined

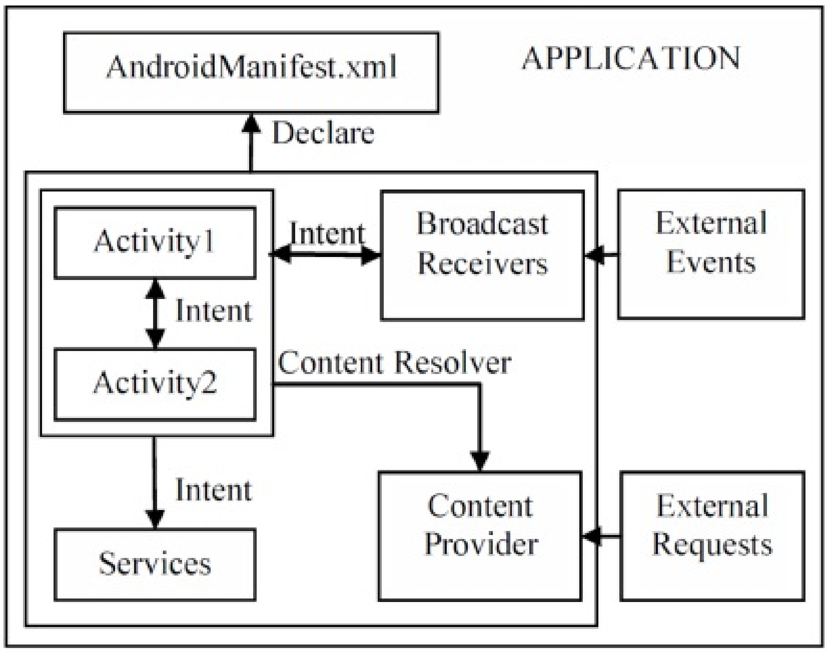


Figure 2.2: Application Components Relationship (Taken from [70])

in the manifest file. The receivers themselves do not have an interface, but they either start an activity upon a request or alert the user through a notification. Broadcast receivers handle *internal* and *external* events through intents.

4. **Content providers**: The providers of specific application data for other applications as well as internal components (*internal* and *external* requests). The data is usually stored in the file system.

5. **Content resolvers**: Resolve a request URI to a specific *content provider*.

In order to communicate with each other, the components use an asynchronous way to do so, through **Intents**, which in essence are objects holding the content of a message meant sent to a component. The components have to be declared in the *AndroidManifest.xml* file. The relations of each component to each other can be seen in Figure 2.2.

2.1.3 Security on Android

Security on Android is handled on two levels[43]:

1. **OS/VM level**: Every application runs a Linux process with its own VM. The code is sandboxed (isolated) from other applications' code, which prevents applications interfering with each other. All application files are only visible to the user and the application itself.

2. **Application level**: Application level security on Android is handled through permissions. The permission system on Android works to protect privacy of the OS user. Each application explicitly declares permissions in the manifest file which allow it to access additional resources outside of the sandbox (e.g. contacts, SMS, camera). The permissions are either granted automatically on installation (normal permissions[29]) or explicitly on first use (dangerous permissions[29]).

   The goal of the permission system is that no application can access resources that affect other applications (e.g. reading files of other applications), the OS (e.g. keeping device awake), or the user (e.g. reading contacts, messages). Normal permissions are permissions that grant access to data outside of the application sandbox, which has low risk of endangering user privacy (e.g. vibrate, Internet, bluetooth, network state). Dangerous permissions are permissions where an application wants a resource that has private user information, affects storage, or other applications (e.g. contacts, SMS, camera). Whether or not a permission is dangerous is decided by the Android OS developers[29].

Aside from the two levels, Android has a set of services which are not a part of the AOSP, but are available for most Android devices, if the manufacturer (OEM) includes it in their version[8]. These include:

- **Google Play:** A set of services that provide a store which users can use to discover and install applications. It is also a platform for user reviews, communication of developers with users, application license verification and application security scanning, among other other things.
- **Android updates:** A service which delivers new features and software fixes as well as monthly security updates to devices.
- **App services:** Frameworks that allow applications to use cloud features, such as backing up application data.
- **Verify apps:** A service which warns or blocks the installation of known or potentially harmful applications. It continually scants the applications on the device.
- **SafetyNet:** An intrusion detection system which tracks and mitigates known, as well as identify new security threats.
- **SafetyNet Attestation:** Third-party API that determines whether a device is Compatibility Test Suite (CTS) compatible. It can also determine if an application is communicating with the application server.
- **Android Device Manager:** A service to locate a lost device.

Of all security services and levels, only dangerous permissions are configurable by the user.

2.1.4 Overview of Security Issues on Android

The Android bug tracker currently has hundreds of tickets issued just for the latest version of the system[1], not counting the framework, system applications and other components. Due to the openness of the platform and the size of the OHA, bug reports are frequent.

Guana et al. have analyzed these reported Android bugs in [31]. They classified the bugs by layer based on the wording, tracked the bug lifetime based on the time elapsed between opening and closing the bug report and analyzed user interest in these bug reports by the number of people following the bug activity. Their finding show that most bugs are found in the **Framework layer**, followed by the **Kernel**, **Libraries** and the **Runtime**. They also report that the lifetime of the bugs is similar across layers, with the layers with more bugs having more long-lasting bugs, due to the fact that around half of the bug closures come in the first month, and newer bugs take priority. With the amount of long-lasting bugs, security holes are unavoidable, and with them come vulnerabilities.

Mazuera-Rozo et al. did an extensive empirical study in [53] to analyze the Android OS stack and its vulnerabilities. They analyzed 1235 Android-related vulnerabilities from five different perspectives:

1. Their type and their hierarchical relationships
2. The most frequent Common Vulnerability Scoring System (CVSS) vectors that describe the vulnerability
3. Layers of the Android OS stack that are affected by the vulnerability
4. Lifespan/Survival of the vulnerability
5. Evolution of the vulnerability through the OS history.

Mazuera-Rozo et al. also analyzed the patches that fixed the vulnerabilities to see the most common types of changes that were necessary by layer. Their study shows a varied set of common problems and misimplementations that lead to vulnerabilities, such as: permissions, privileges, access control, weaknesses that affect memory, data handling, improper checks and handling of exceptional conditions. The main findings of their study are that: Most vulnerabilities come from improper access control (23%), restriction of operations in the bounds of memory buffers (20%) and from issues when processing data (14%), as well as improper input validation (8%); At least 69% of the vulnerabilities are exploitable remotely, without authentication of the system. In at least half of the cases it came to total system integrity compromise where the attacker was able to modify any file in the system, as well as do a total shutdown of the resource and the system; The

[1]`https://issuetracker.google.com/issues?q=componentid:192706` (12.02.2020)

most affected layers were the *kernel* layer, with 52.39% of the vulnerabilities affecting it and the *native libraries* layer with 30.07% of the vulnerabilities affecting it, with *kernel drivers* (77% of the 52.39%) and the *media framework* (78% of the 30.07%) being the most affected subsystems; On average, a vulnerability goes unnoticed for 770 days before being identified, while possibly being exploited. This does not consider the unreported and undiscovered vulnerabilities; The overall number of vulnerabilities has been rising since 2015, which may be a side-effect of the growing user-base, which leads to more vulnerabilities being discovered[53].

2.2 IT Security and Privacy

The NIST Computer Security Handbook defines IT security as "protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability, and confidentiality of information system resources (includes hardware, software, firmware, information/data, and telecommunications)."[64]

Confidentiality is a set of rules that limits access to information, *integrity* assures that the information is trustworthy and accurate, and *availability* guarantees a reliable access to the information.

Altman defines privacy as: "selective control of access to the self or to one's group"[12]. This makes privacy subjective. Aside from that, the amount of access that people allowed also varied constantly. Privacy is needed by every individual, for development, both by young people as well as adults. Trepte explains in [66] that privacy is needed to achieve autonomy to break social norms and allow new thoughts and behaviors. Not every piece of information is confidential but a person has to be able to monitor and moderate who participates in sharing of private information. **?** distinguishes three types of privacy[**?**]:

1. Informational privacy - data is not public unless the person wants to share it
2. Decisional privacy - the right to make decisions and actions while protected from unwanted external influences
3. Local privacy - protection against the entry of unwanted people into private rooms and areas

This thesis focuses on **informational privacy** only.

Respecting a person's privacy means respecting the person as an autonomous being with the freedom to live their own life independently. Corporations having secret knowledge about a person threatens the freedom of expression and assembly[**?**].

Through these definitions, it can be seen that privacy preservation is a challenge in IT security. Wilkowska and Ziefle[71] show privacy and security aspects of a technology ranking highly in importance to customers. The customers in this case are the users of medical assistive technologies which run on Android.

With today's General Data Protection Regulation (GDPR), the obligation of "Privacy by Default"[6] is introduced, which puts privacy as a priority in IT security.

2.2.1 Privacy in System Design

According to Cavoukian et al.[20], in IT, privacy must be an organizational priority, approached from a design perspective and incorporated into networked data systems and technologies by default. Cavoukian et al. state that the objectives of *Privacy by Design* can be achieved through the seven foundational principles:

1. Proactive not Reactive; Preventative not Reactive - all privacy policies and mechanisms should be in place so that they can be observed and resolve any privacy issue before a problem is encountered.
2. Privacy as the Default - all private data should be automatically protected and intact without any action from the individual.
3. Privacy Embedded into Design - privacy not an add-on, but an essential component, which does not diminish functionality.
4. Full Functionality - all interests and design objectives should be accommodated without unnecessary trade-offs.
5. End-to-End Lifecycle Protection - all data remains secure during its lifetime, from collection to destruction.
6. Visibility and Transparency - components and operations should be visible and transparent, subject to independent verification.
7. Respect for User Privacy - data must be collected, used, stored, shared and retired with respect to individual privacy.

To help IT architects support privacy from early stages of software development, as well as evaluate the privacy of existing systems, Hoepman derived eight design strategies, split into four data oriented and four process oriented strategies[35]:

- Data oriented:
  - **Minimize** - the basic privacy design strategy which states that the amount of private data being processed must be reduced to an absolute required minimum.
  - **Hide** - states that any private data and its relation to other data should be hidden from plain view to avoid abuse.

  - **Separate** - states that private data being processed should be processed in a distributed manner to avoid the creation of profiles of a person.
  - **Aggregate** - states that private data should be aggregated whenever possible with the highest level of aggregation and least amount of detail needed.

- Process oriented:
  - **Inform** - states that private data providers should be informed when their data is being processed.
  - **Control** - states that data providers should have the control over the processing of their data.
  - **Enforce** - states that a privacy policy that is compatible with legal requirements needs to be implemented and enforced.
  - **Demonstrate** - states that a data holder has to be able to show compliance with privacy policies and legal requirements

2.2.2 Handling Private Data for Computational Purposes

There are three roles involved in handling private data:

- Data respondent - a person who gives private data, voluntarily or not
- Data holder - or data controller, which collects and stores the data
- Data user - the individual or company which uses the stored data for computations

Even if the data a company collects does not get used for malicious purposes, the data can get stolen and/or published publicly. In cases like these, if the data is not stored correctly, the public and private information can be traced back to the data respondent. The research of privacy studies the approaches to avoid sensitive data disclosure. Most commonly, there are two types, or families of disclosure[63]:

1. Identity disclosure - when it is possible to identify a record of an individual in the database upon the release of information
2. Attribute disclosure - when it is possible to learn something about a property attributed to individuals upon the release of information

Multiple privacy models exist which attempt to ensure user privacy in publicly accessible data, as listed by Said and Torra in [63]:

- Secure multi-party computation - used in case where multiple parties want to compute a joint function of their databases without sharing their data and partial results. The implementations most commonly rely on cryptographic protocols.

- Privacy from re-identification - used to avoid identity disclosure with the help of masking methods which modify the file that contains the released information to make re-identification impossible or difficult.
- k-Anonymity - used for identity disclosure as well. It is a special case of re-identification privacy. The data gets masked in this case as well. The main difference with this model is that it ensures that every combination of identifiable data has at least k records that are indistinguishable from each other.
- Differential privacy - used to ensure that the output of a query does not depend on a single record, its presence or absence.
- Integral privacy - used to ensure that conclusions cannot be drawn based on changes from one database edition to another.

Alongside these models, there are also variations, generalizations and combinations of the models. The data protection mechanisms can be classified multiple ways[63]:

- Based on whose privacy is being sought, we have respondent, holder and user privacy.
- Based on needed computations. Depending on which function will be executed on the data, the data protection can be adapted to protect certain parts of the database. There are *computation-driven* (i.e. specific purpose, function is known), *data-driven* (i.e. general purpose, function is not known) and *results-driven* (i.e. function is known and data to be protected is in the result) protection measures.
- Based on the number of data sources. There are data protection procedures which are defined for a single data source and there are procedures defined for multiple data sources.

The masking methods are the standard approach in data-driven respondent and holder privacy. The main goal is to publish a database of lower quality. This database can still fulfill its purpose in e.g. machine learning algorithms while maintaining privacy. The lower quality is achieved through *perturbative* (e.g. noise addition, multiplication, rank swapping) and *non-perturbative* (e.g. generalization, suppression) methods. With these methods, the most important aspect is the information loss measure, which has to be sufficiently high for all analysis methods to still work[63].

Secure multiparty computation is used for computation-driven methods of preserving respondent and holder privacy with multiple sources in use. To ensure privacy, the specific compute function needs to be known in order to write a cryptographic protocol so that nothing can be found out from the final result. One such example is when competing stores try to find out which product is the most popular without letting each other learn the sales figures.

Differential privacy is used to achieve respondent and holder privacy in computation-driven methods when only one source is used. The typical procedure is to either use perturbative methods on the dataset or the final output of a function.

The user privacy methods protect the data produced by their actions within the data holder's system. Mix networks can protect users with privacy over their connection history. This way, the user is protected from identity disclosure. To protect from attribute disclosure, secrecy of transmitted data is achieved through cryptography. Various methods also exist to protect privacy during information retrieval. P2P community queries aim to protect identities, while agents such as TrackMeNot, GooPIR and systems such as DisPA protect the data[63].

The classification of data protection mechanism is summarized in Table 2.1. The data protection mechanisms are classified according to the three dimensions: whose privacy is being sought, what computation is being done, and number of data sources. For user privacy, classification is done according to the type of application and what we want to protect.

Whose privacy	Computation known?	Num. Sources	Protection methods
Respondent and holder	data-driven	multiple/one	Masking methods
Respondent and holder	computation-driven	multiple	Multiparty computation
Respondent and holder	computation-driven	one	Differential privacy
Holder	result-driven	one	Rule hiding
	Application	What to protect?	
User privacy	communication	identity	Mixes, crowds
User privacy	communication	data	Cryptography
User privacy	information retrieval	identity	P2P communities
User privacy	information retrieval	data	DisPA, TrackMeNot, GooPIR

Table 2.1: Data protection mechanisms (Taken from [63])

2.2.3 Current State of Privacy Research

With the rise of technology use in everyday life, and the rise of the ability to process the data generated by the users of this technology, information processing became an integral part of business strategies of various companies. Companies such as Facebook collect data from their users in order to use it for unknown purposes. One example of Facebook abusing the data they collected was in 2018, when it was revealed that Facebook shared the data of 87 million users, without their consent, with Cambridge Analytica, which the latter used to profile users and sway them in United States political events[36].

Communication data shared through messaging and social media applications can be mined to extract sensitive information that was not communicated directly, i.e. deduced from the context. More traditional and older types of communication such as E-Mail or forums are usually harder to mine. For example, the forums are typically about one thing and creating a profile from that information does not give a full picture about the user, i.e. makes the mined data unusable. However, with large centralized social media applications, the big corporations, especially Facebook with Instagram and WhatsApp, handle such amounts of data that they have the ability to deduce private details about their users. Facebook can learn among other things what each users: ancestry, religion, political stance, wellbeing, whereabouts, gender, age, illegal substance use are, all without asking the questions[40].

With the state of Google's reverse image search, it is possible that Google is able to find all the photos or videos with a specific person in them. It is easy to control whether the photos you take show up online somewhere, but with people taking pictures and recording videos everywhere, it is also possible to show up online without your knowledge. This coupled with the typical photo and video metadata raises concerns on whether it will be possible for a person to have control over their privacy[40].

The state of Android regarding the handling and leaks of private data is covered in Section 2.3.

Another trend of reduced privacy in everyday lives concerns smart home or Internet-of-Things (IoT) applications. Home automation products like the Google home[2], Amazon Alexa[3] and similar, increase users' comfort in their homes by making it easy to control connected devices with voice commands. This however is made possible by collecting extensive data from all users. This data is used for making the service more reliable. It is, however, possible that the data is used for advertising, as Amazon (i.e. Amazon store) and Google (i.e. Google Ads) both have an interest in user analytics. If the data stays locally on the device it is not a privacy concern, however, in the case of digital voice assistants like the Amazon Alexa, their command processing is done in the cloud, since the devices themselves are not capable of processing it[23]. Chung et al.[23] also show that unintentional voice recordings occur, without the "Alexa" keyword on the Echo, which is an additional privacy concern.

Even if a user tries to be anonymous and e.g. use different nicknames for different platforms, the information from these users can be de-anonymised and the platforms can still be linked using e.g. an IP address, residence address and birthday[32].

2.3 Data Leaks

As defined by Shabtai et al.[64], data leakage is *the accidental or unintentional distribution of private or sensitive data to an unauthorized entity.* Sensitive data includes intellectual property, credit-card data, financial information, health information and other information. It poses an issue for individuals and companies as the number of incidents get more frequent and more expensive.

[2]https://assistant.google.com/platforms/speakers/
[3]https://developer.amazon.com/en-US/alexa

2.3.1 Data Leaks in General

Data leakage can occur both due to malicious intent and inadvertent mistakes of insiders and outsiders. The exposure of private information can hurt individuals and organizations in two ways: *directly* and *indirectly*. *Direct* losses are easily measurable and tangible, such as: the loss of sales, investigation costs and fines due to violating customer privacy when it comes to corporations; and financial loss (e.g. credit card information stolen) when it comes to individuals. *Indirect* losses are harder to quantify, examples include: negative publicity, customer abandonment and reputation damage[64].

Data leaks can occur in various forms and places. In [56], Quick et al. keep track of all data breaches which leaked 30 thousand or more records. As of November 2020, a total of 11.38 billion records were compromised in 360 breaches since 2004. A total of 57% of the records leaked in 229 leaks caused by third-party hackers. Lost devices were the reason for 48 breaches, poor security caused 46, and inside jobs and accidents caused 19 and 18 breaches respectively[56].

The leaks caused by outsiders are most commonly: hacker break-ins (e.g. exploiting a system backdoor or misconfigured access control), malware, viruses, and social engineering (e.g. phishing). The leaks caused by insiders are either done deliberately (e.g. for financial gain) or inadvertently (e.g. accidental data sharing without protection)[22].

2.3.2 Data Leak Categorization

Data leaks are characterized based on these attributes[64]:

Where did the leak occur? There are three possible locations for general data leaks: inside of the organization (i.e. leaked from source physically inside the organization), outside of the organization (e.g. from an employee's mobile device) and from a third-party location (e.g. a partner was hacked and used to gain access).

Who caused the leak? A leak can be caused by an outsider, an insider, a vendor/-contractor and a customer. *Insiders* are mostly trusted actors with access privileges and their leaks are split by their nature into accidental and intentional. *Outsiders* are mostly not trusted and have no access privilege. *Vendors and contractors* have some level of trust as they mutually benefit as business partners. Their communication is often secured and encrypted. *Customers* have access privileges for specific applications.

What was leaked? Three types (phases) of data can be leaked: data-at-rest (DAR), data-in-motion (DIM) and data-in-use (DIU). Each type takes a different approach to protect it. Incidents are classified based on the phase the data was in when it leaked.

How was access gained? This question pairs with "Who caused the leak?". The data can be access through various means such as: *Hacking* - which means, among other things: exploiting insecure passwords, misconfigured access control, backdoors, stealing legitimate credentials, using SQL injection, cross-site scripting and buffer overflow attacks. *Malware* - or malicious software can lead to leaks by e.g. recording keystrokes when passwords are entered or sending files containing sensitive information to an external source. It can both be installed from the inside, as well as unknowingly downloaded from an unsafe website. *Social attacks* - which are most commonly executed through observation, assault, physical harm and social engineering. *Physical access* - access is gained through theft, using an unlocked unobserved computer or wire-tapping. *Human errors* - which are caused by the developers, IT technicians and data respondents. This includes misconfiguration, software bugs and improper disposal of sensitive documents.

How did the data leak? The leaks are classified by leakage channel. *Physical leakage channel* - physical media, such as storage drives, laptops and computers were moved from the organization. *Logical leakage channel* - the data was leaked digitally through broadcast, was uploaded or sent to a third party outside of the organization/device.

How is the leak categorization used? This categorization is used after a data leak occurs. Following the categorization questions, the investigators can identify which type of data leak occurred and use that information to:

- Analyze the specifics of the data leak, e.g. determine the entry points of the attackers, what data was breached and what caused the entry point exploit.
- Evaluate the risk, i.e. do risk assessment to determine what damages will occur to the individual or the company whose data was compromised.
- Find a solution to mitigate or minimize the damages, and determine how much time the investigators have to react before the leak is publicly known.

2.3.3 Data Leak Detection and Prevention Solutions

The investigation done by Shabtai et al.[64] shows that a single solution for data leak prevention is not feasible, and that companies should focus on expanding their security beyond classic viruses and network threats. For this, Data Leak Prevention (DLP) solutions need to be implemented. The technology behind enhancing DLP can be split into: standard security measures, advanced security measures, access control and encryption and designated DLP systems, as shown in Figure 2.3. Although the image referenced is from 2012, the categories still apply[64].

- *Designated DLP solutions* are meant to detect and prevent: attempts to copy and send sensitive data, intentionally or unintentionally and without authorization mainly by actors who are authorized to access the sensitive data. One of the main features the DLP solutions is the ability to classify data as sensitive. They use techniques such as: data matching, fingerprinting, statistical methods and rule matching.

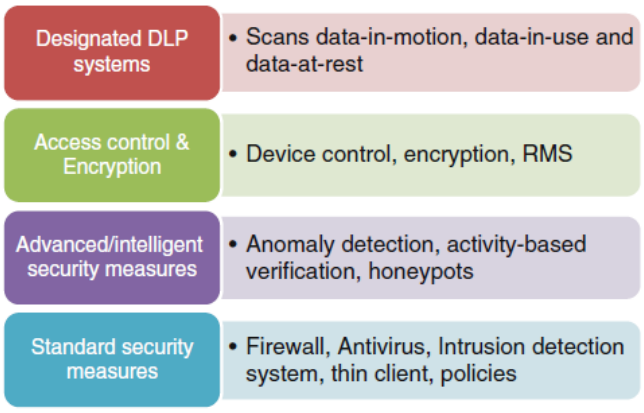

Designated DLP systems
• Scans data-in-motion, data-in-use and data-at-rest
Access control & Encryption
• Device control, encryption, RMS
Advanced/intelligent security measures
• Anomaly detection, activity-based verification, honeypots
Standard security measures
• Firewall, Antivirus, Intrusion detection system, thin client, policies

Figure 2.3: Categories of technological approaches used to provide data leakage detection and prevention. (Taken from [64])

- *Access control and encryption* prevent the unauthorized access, of the listed measures, these are the simplest conceptually and prevent both insider and outsider attacks. Encryption of the data in the databases and data being transfered from one party to another makes attacks which try to read communication data or steal database contents harder since the attackers need to break encryption to read any information. Access control prevents attackers from gaining this data by reducing the number of people who can access it (e.g. through passwords) to a minimum.

- *Advanced security measures* are mechanisms such as machine learning and temporal reasoning algorithms for detecting abnormal data access, activity-based verification (i.e. user activity near sensitive data) as well as applying the honeypot concept (i.e. leaving fake private data that is easier to access than actual private data as bait for attackers).

- *Standard security measures* are common mechanisms like firewalls, Intrusion Detection Systems (IDSs) and antivirus applications. They protect from both the outside and inside attacks.

The authors of the book "A Survey of Data Leakage Detection and Prevention Solutions"[64], Shabtai et al. compiled multiple definitions of DLP solutions:

- "A system that monitors and enforces policies on fingerprinted data that are at-rest (i.e., in storage), in-motion (i.e., across a network) or in-use (i.e., during an operation) on a public or private computer networks"

- "systems that identify, monitor, and protect data-in-use, data-in-motion, and data-at-rest through deep content inspection using a centralized management framework"

- "a set of inspection techniques used to classify data while at-rest, in-use, or in-motion and to apply predefined policies (for example, logging, reporting, relocating, tagging, or encrypting)"
- They themselves define a DLP solution as "a system that is designed to detect and prevent the unauthorized access, use, or transmission of confidential information"

This thesis uses the last definition, while only developing a solution that detects data leaks, and does not prevent them.

2.4 Static Analysis

This section describes the fundamentals of static analysis which are frequently referenced throughout this thesis.

As described by Louridas[52], static code analysis is the process of checking a program for errors without running it, i.e. by checking the code for common error patterns. The code can get analyzed on multiple levels, e.g. a checker can check every line individually for errors, without taking other lines into consideration. A checker can also check whole procedures or more complex flows of the program, depending on its capabilities. In the case of static analysis of Android applications, an application gets analyzed from entry points to exit points.

The following terms are used frequently in the field of static analysis:

- Sensitive data: Any data generally considered private (e.g. sensor data, IMEI, contacts, passwords)
- Source: Application input or API which gives sensitive data. An example would be an Android OS method which retrieves the IMEI.
- Sink: Output point of the application which can send data externally. An example would be a method which triggers the sending of an SMS message.
- Precision: In context of static analysis evaluation, precision represents the number of cases that are correctly labeled among all cases labeled by a solution. To get the get the correct numbers of e.g. leaks, an analysis tool needs to be tested on controlled test cases for which all leaks are known. For example: if we have an application with 12 data leaks, and a tool which discovers 10 leaks, of which 8 are correctly labeled, the precision is 8/10, or 80%.
- Recall: In the context of test case analysis, recall represents the number of cases that are correctly labeled among all relevant elements. In the example listed in the definition of precision, the recall would be 8/12 or 66%.

- F-Score: Is a measure of general accuracy of a solution, calculated from both precision and recall. The highest F-Score is 1 and it indicates perfect precision and recall. The measure is traditionally, and also in this thesis, the harmonic mean between the precision and recall. It is calculated as: $2 \cdot \frac{precision \cdot recall}{precision + recall}$

2.4.1 Comparison to Dynamic Analysis

According to Ball[16], dynamic analysis is the analysis of a running program. It derives properties of a running program which hold for one or more executions, while static analysis analyzes the code for properties which hold in all executions. Dynamic analysis most commonly proves that a program violates defined properties of its behavior.

The information provided by dynamic analysis is typically precise because it shows the e.g. actual execution, memory allocations and data structures created by the program. Because dynamic analysis is also dependent on program inputs, it can detect changes in execution based on inputs and thus detect manipulation of those inputs.

Static and dynamic analysis complement each other:

- *Completeness* - Dynamic analysis can generate a set of invariants which hold for a set of executions, while static analysis can determine whether these invariants hold for all executions. If the two sets of analysis results from static and dynamic analysis are not equal, one of two cases is possible: either the dynamic analysis did not cover all paths, because its inputs did not cover all branches, or the static analysis considered infeasible paths.

- *Scope* - Dynamic analysis has the potential to discover program dependencies along long execution paths in the program, while static analysis has a more limited scope and may miss these dependencies which are far apart, since it covers larger portions of code.

- *Precision* - Dynamic analysis examines the concrete program domain, while static analysis uses abstraction to ensure that it can terminate, since concrete domains produce large run-time and memory over-heads from producing larger amounts of information, and thus static analysis loses information it gathered in the start.

This thesis uses static analysis in the proof-of-concept tool implementation due to its strengths, which are: high code and conditional coverage and the lack of need to actively run the program. In comparison to dynamic analysis, static analysis gives a comprehensive analysis of any application without the need to know which input combinations trigger certain flows to discover bugs or leaks. Dynamic analysis in case of Android applications would introduce a need for an emulator and a runner which mimics a user's input, which adds a level of complexity which falls outside the scope of this thesis.

2.4.2 Static Analysis Support Frameworks and Algorithms

This section describes frameworks and algorithms commonly used in static analysis.

Bytecode Optimization Framework - Soot

Soot[69] is a Java bytecode optimization framework. The purpose of it is to transform compiled Java bytecode into its intermediate decompiled representations and perform optimizations on each step before turning it back into bytecode.

The application bytecode gets transformed into the following representations:

- *Baf* - A near-bytecode representation.
- *Jimple* - A Java-similar representation which only differs from Java due to limited operation types. This representation is commonly used in static analysis tools.
- *Grimp* - An aggregated version of Jimple, used as the basis for the Soot decompiler.

This process can be visualized in Figure 2.4, with *Baf*, *Jimple* and *Grimp* being the bytecode representations. The compiled *.java* files turn into *.class* files, which transform into *Baf*, *Jimple* and *Grimp* representations, with developers being able to create optimizations in each representations and then transform it back to *.class* files.

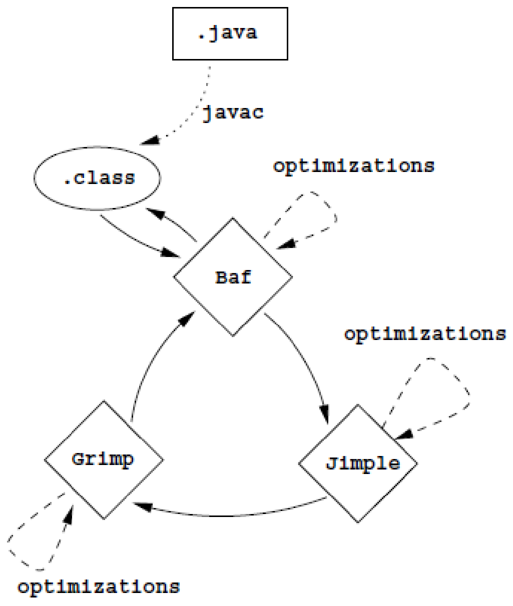


Figure 2.4: Soot optimization flow. (Taken from [69])

The main reason why Soot is used in data leak analysis is that it provides a framework which decompiles the code into a readable representation (Jimple), which can be analyzed without bytecode knowledge. Soot has functionalities which transform a given application input into Java objects, which can then be analyzed for any purpose, from syntactical error checking to data leak analysis.

Taint Analysis

Taint analysis is the analysis of the flow of data within an application which uses marked (tainted) data to recognize whether the change in this data influences the output of a program. For example: if we taint the phone's IMEI and see that passing two different IMEI values through the application changes what the application provides to the sink, we can conclude that there is a potential data leak, without having to understand what each operation does. In this thesis, the term "Forward-taint analysis" is also used. Forward-taint analysis describes the taint analysis which uses a source as its starting point.

Alias Analysis

Alias analysis is the analysis of code which considers all aliases of a value. In programs, a value at some memory location can have multiple pointers directed at it, all these pointers are aliases for the same value. In data flow analysis, the analysis tool should keep track of all the aliases and how they manipulate and use the value to get the full understanding of what is being done withing the program. In the example in Figure 2.5, the analysis tool has to keep track of the aliases to discover the leak of w to a *sink*. In the example: since $x.f = w$ (1 and 2), $x = z.g$ (3) and $z = a$ (5), we know that $a.g.f = w$ (4 and 5). With $b = a.g$ (6), then $b.f = w$ (7) we thus know that w is leaked from *source()* to a sink through *sink(b.f)*.

In its simplest form in static analysis, alias analysis means replacing the aliases with a common name, preferably something that does not conflict with other variable names and only affects one execution flow.

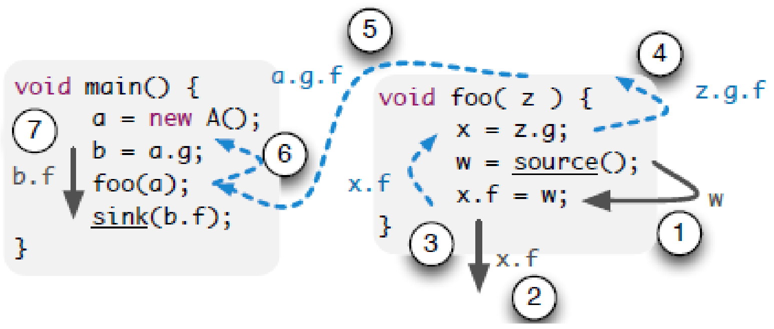

```
void main() {
    a = new A();
    b = a.g;
    foo(a);
    sink(b.f);
}
```
```
void foo( z ) {
    x = z.g;
    w = source();
    x.f = w;
}
```

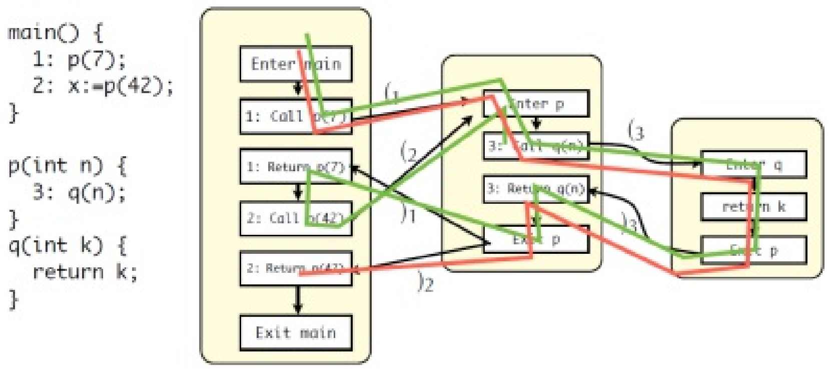
Figure 2.5: Taint analysis under realistic aliasing. (Taken from [15])

In this thesis, the term "On-demand backwards-alias analysis" is used. Backwards-alias analysis describes the alias analysis which uses a sink as its starting point, while "On-demand" means that the analysis only starts for some sinks discovered in an application and not all.

2.4.3 Context Sensitivity

When we want to analyze a flow of an application, we also have to keep the context known. Context is the minimal set of data used by an application that allow it to be interrupted and continue operating after the interruption. The context also saves information about the call and return sites of a method. Connecting all methods that call each other does cover all cases, even if some are not actually executable. However, with increased application complexity, the complexity of these connections is exponentially larger. Static analysis is expensive, since it goes through all possible paths from a source to a sink. Limiting these paths and branches with algorithms is something that FlowDroid developers worked on since the paper ([15]) was released.

Figure 2.6 shows two paths within an application which would get detected as possible paths by a static analysis solution. The green path is realizable, since it goes to the correct return site, and the red one is infeasible, since it goes to an invalid return. By keeping the context known, storing the return sites for every call site and storing the call site when inside the called procedure, static analysis can correctly determine where to return. In the case in Figure 2.6, procedure *p* cannot return to *2:* after being called by *1:*, since the correct return site is *1:*. The static analysis thus knows that the red path cannot be feasible, since it contains the return to an incorrect return site.



```
main() {
  1: p(7);
  2: x:=p(42);
}

p(int n) {
  3: q(n);
}
q(int k) {
  return k;
}
```

Figure 2.6: Example of a feasible (green) and infeasible (red) paths within a program. (Taken from [59])

2.4.4 Flow Sensitivity

Flow sensitivity in program analysis describes the analysis which takes the order of statements into account. Whereas flow-insensitive analysis may say that a statement holds in a program, flow-sensitive analysis will say that e.g. a statement holds after line 20 in procedure p. Any data-flow analysis algorithm is inherently also flow-sensitive[15].

2.4.5 Static Analysis Algorithms

This subsection describes the idea behind algorithms used for static code analysis, and its basic terms.

Static code analysis algorithms in general turn structured code into a graph of nodes and edges which describe the execution(s) of a program, and analyze the properties of this generated graph. Because of this, the code analysis algorithms resemble algorithms for e.g. graph reachability.

The following terms are commonly used throughout the thesis:

- Depth-First Search (DFS) - a search algorithm used for traversing tree and graph structures by going as far as possible from a root node along each branch, before going back and exploring the next branch. In Figure 2.7, the algorithm would traverse the tree as numbered.
- s_p - commonly represents the starting point of a program execution flow.
- N - In general, N represents units or code fragments (statements or instructions), which denote some form of an execution unit.
- D - In general, D represents abstractions. In the case of algorithms which are described in Section 4.1, the values d_i are access paths describing references to tainted values.
- The notation $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ means that there is an edge between the two nodes. In the case of static analysis, it means that d_2 holds at n if d_1 holds at s_p of some procedure p.

2.4.6 Evaluating Static Analysis Data Leak Detection Tools

To evaluate a static analysis data leak detection tool, developers of the tool use controlled test case applications which contain a known number of leaks. For example, to check if a tool can detect a certain leak, it should run on a test case application which contains one or more instances of that leak and analysis output is compared with the baseline or the ground truth. If the two match, then the tool can successfully detect this type of leak. This approach is repeated for every known data leak vulnerability.

In the case of static data leak detection on Android, the most popular benchmarking tool used to evaluate analysis approaches is DroidBench[**?**]. It currently contains 190 test cases and every test case application is well documented and describes how many leaks it contains, as well as which leaks those are.

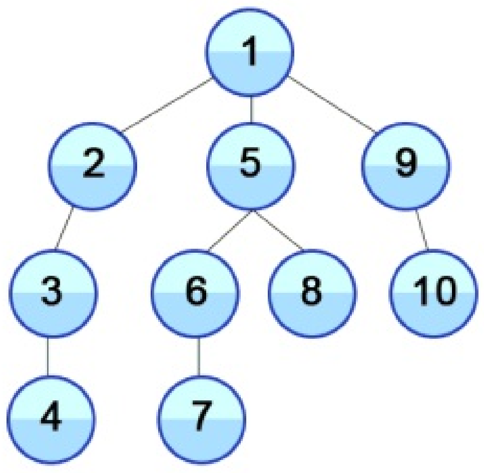


Figure 2.7: Example of depth-first search tree traversal. Numbers represent the order in which the tree is traversed.

To evaluate how well an application scored on a benchmarking tool, the tool's precision, recall and F-Score are calculated based on the output of the analysis compared to the baseline.

CHAPTER 3

Research of Privacy-related Vulnerabilities on Android

In order to determine which private data leak vulnerabilities this thesis covers, the state-of-the-art literature on vulnerabilities was researched. Along with vulnerabilities, the data leak detection and prevention solutions literature was also covered. Section 3.1 explains the basics of Android specific data leaks. Section 3.2 covers the data leak vulnerabilities addressed in current research. Section 3.5 covers the current data leak solutions, how they work and which vulnerabilities they prevent, as well as summarizes their limitations, in order to highlight what else needs to be covered in the Proof-of-concept solution. Finally, Section 3.6 selects the main private data leak vulnerabilities that this thesis aims to cover.

3.1 Data Leaks on Android

Subsection 2.3.3 focused on data leaks in general, the following part focuses on Android-specific data leaks.

In a study done by Reardon et al.[57], the authors developed a system to discover vulnerabilities in order to observe the different ways applications gain access to leaked data. The basic protection of data on Android is done through sandboxing, and granting permissions for actions which need them. However, there are ways to avoid permissions. For example, using covert and side channels with deceptive practices, attackers can mislead users and exploit vulnerabilities. One such example is using the MAC address of the Wi-Fi access point to get the approximate GPS coordinates without getting the Location permission from the OS.

Covert Channel A covert channel is a mean of communication between, in the case of Android, mobile applications. This allows them to transfer information between each other, even if one of them is not authorized by the system to receive it. There are various covert channels possible with Android such as: ultrasonic sound, vibrations and external network servers used for indirect communication[57].

Side Channel A side channel is a communication path which an application can use to gain access to a resource without a permission check. This can be done through unconventional unprivileged functions, or copies of the relevant information being available without permission protection. One such example is the timing attack which can obtain an encryption key due to the fact that they can time how long an encryption algorithm was running. Side channels are more often than not unintentional, and a consequence of the complexity of the operating system. Due to the complexity, some data can be accessed from more than one API point, one of which may be secure, while the other has a missing permission check[57].

The difference between the two Channels is illustrated in Figure 3.1. In the covert channel (a), a security mechanism denies app 2 the access to a resource. App 2 then uses app 1, which has access to the wanted resource, to gain access by communicating through a covert channel with app 1. In the side channel (b), app 1 is denied access to the resource, however, the application then uses a side channel which circumvents the security mechanism[57].

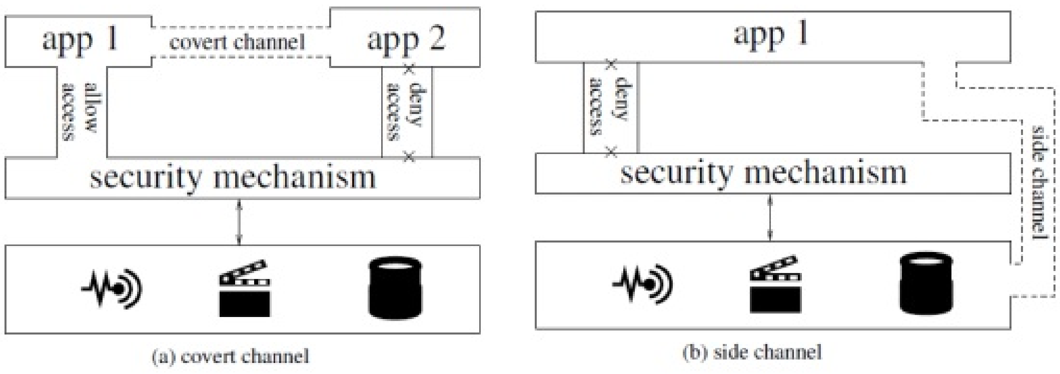


Figure 3.1: Illustration of the differences between (a) covert and (b) side channels. (Taken from [57])

A total of 88,113 applications were tested. The selection of applications was based on popularity. The applications were tested to see if they leak private data, which in their case was defined as any piece of information which can identify an individual or distinguish them from another person. To make sure the applications not only had the ability to (i.e. there is some piece of code which accesses information through a side channel), but actually leaked data (i.e. sends it to a third-party), they de-obfuscated and analyzed the network traffic of these applications[57].

Five different side and covert channels were found[57]:

- **IMEI** - or the International Mobile Equipment Identity is a number which uniquely identifies a phone. One of the main uses is the detection and blocking of stolen phones. It is often used by third parties as a unique identifier of a user to track them across the Internet. One example of an exploit with IMEI is the Salmonads Software Development Kit (SDK). Any application with this SDK has access to a file in which they write the IMEI and other sensitive data. Only one of them needs legitimate access in order for all of them to use it.

- **Network MAC** - similar to IMEI, the Media Access Control or MAC address is a 6-byte unique identifier used for network communication. It is often used by advertisers and analytics software to track a user/device. In Android, the access to the MAC address requires a permission, however, a large number of applications using specific C++ native code to invoke unguarded UNIX system calls allows applications to obtain a MAC address without permission. A total of 12,408 applications used such calls, while 748 did not hold the required permission for the MAC address.

- **Router MAC** - As with Network MAC, router MAC is protected with its own permission. It can be used as a GPS substitute and can link multiple people as related if they are connected to the same network. The analysis found two side channels which give access to Wi-Fi information: *Reading the ARP cache* and *asking the router directly*. The Address Resolution Protocol, or ARP discovers and maps MAC addresses with given IP addresses. To improve performance, historical data is saved in a cache file, which is unsecured and can be accessed by any application. In the case of the router, the flaw is on their side and not all routers behave in the same way. The MAC addresses were gained in an Internet gateway device configuration file which was requested by an application. The router assumed that any application on any device connected to it was trusted.

- **Geolocation** - The researchers discovered that applications sent geolocation data even though they had no location permissions. A part of the applications used IP-based geolocation. Depending on the location it was accurate up to a meter; however it was more commonly less accurate. One real side channel was found in an application called Shutterfly, which used photo EXIF metadata containing a time stamp and coordinates. If a user takes photos often, any application with media storage access can read coordinates without the required permission.

Apart from the physical files and device information being leaked, some application can extract sensitive data from "live" data, i.e. from a camera's viewfinder. Srivastava et al. have analyzed applications using the camera to identify how applications such as augmented-reality applications can violate privacy in non-standard ways[65].

In their main findings, they discovered that over 600 Android applications extract and send private visual data, such as: text from surrounding environment, faces and QR codes with the camera. They also discovered in a survey that the actual application functionality is different than what users were expecting in 61% of the cases, and that the users care about the way their camera data was processed, but were unaware of it due to a lack of transparency. In 19% of applications, the applications extracted information which the participants did not expect.

The exact information which was sent to external servers or third-parties was unknown, but the timestamps of network traffic correlated with the timestamps of photo capture. The data being shared could be images being sent for processing, however, with regards to privacy, such practices present a gray area[65].

Another major investigation of user data leakage was conducted by Ge et al. in [28]. The research was conducted over 9 months using over 180 thousand applications from more than 50 Chinese App Stores. Due to them being from China, Google Play Store was not one of the options, but they found a set of applications that is common between the Chinese App Stores and Google's, the results of applications' access to user information were within a 4% difference. The analysis consisted of observing privacy leakage in API calls to Android System and did not include encrypted transfer of sensitive information, which results in possibly optimistic results. Their discoveries show:

- 0.39% of applications send private contact information to remote servers without notifying the user
- 0.2% of applications send SMS without informing the user of the cost
- 15% of applications bound their execution to download or installation of other applications or advertisements which can be used as a vector of malicious code
- 95% of applications wanting access to private information aren't actually accessing it

Apart from that, some applications also deliberately interfere with other applications after reading the installed application list. What this shows, apart from the malicious aspect, is that requesting access to private information has become the norm, even though it's not needed most of the time, just because it's easy to get[28]. This corresponds with the findings mentioned in Sections 2.3 and 2.2 in which the users are overly generous when granting permissions since they assume benign intent. The opinion the researchers have is that the App Stores should have stricter rules. Something as simple as limiting this access to applications can stop many potential leaks of private information, but would not hinder the majority of applications.

3.2 Research of Vulnerabilities Addressed by State-of-the-Art Research

In the state-of-the-art research, the researchers mostly focus on reporting on an issue briefly, and then approaching the issue with their own solution. Papers which focus on reporting on the issue alone are rare. To give an overview of the state of private data leak vulnerabilities on Android, other sources were used as well. The main focus is on vulnerabilities that are due to flaws in the OS, however, some vulnerabilities require an unsecured application to malfunction in order to work. For this reason, application-specific vulnerabilities are also covered.

To understand how application vulnerabilities evolve over time, Gao et al.[27] conducted a study to analyze the lineages of Android vulnerabilities over time. Around 28 thousand applications with at least ten updates each were analyzed. They used vulnerability-finding tools to create reports, which they studied to analyze which vulnerabilities were most prevalent, how long they survived and if they came back after being patched. The main findings of their study are[27]:

- Most vulnerabilities survived at least three updates.
- Third-party libraries were responsible for most vulnerabilities.
- Almost all types of vulnerabilities were able to get reintroduced, with encryption-related vulnerabilities being the most common ones.
- Some vulnerabilities may imply that an application is malicious.
- Some, previously benign, applications became malicious after an update.

Even if an application is not malicious, the vulnerability it contains may be exploited by a malicious application if it is known to them. For example: if an application stores sensitive data in a publicly accessible file, it can be accessed by a malicious application and forwarded to a third party.

Other malicious applications exploit system-level vulnerabilities to gain root-access or other access to private data. Wu et al.[72] show that "Elevation of privilege" and "Information disclosure" vulnerabilities are the most common types of system-level vulnerabilities on Android. They also show that most vulnerabilities (65%) come from the kernel layer, with the Native Libraries layer being the second most vulnerable with 23%. The most common trait of these vulnerabilities is that they come from C/C++ coded modules, and are mostly in third-party drivers and libraries. This implies that the code quality of Android's own code is of relatively higher quality and less vulnerability-prone. The issues may also come from the fact that potential memory corruption issues (e.g. buffer overflow) are more common in C/C++. Some modules, like media, Wi-Fi and telephony related modules introduce vulnerabilities across multiple layers, from the kernel to applications[72].

To further confirm the findings of Wu et al.[72], Linares-Vasquez et al.[50] conducted an empirical study and found emphasizing results regarding the distribution of vulnerabilities across layers. They show that most vulnerabilities come from improper restrictions on memory buffer operations, issues with data processing, improper access control and input validation. This concurs with the comment that poor code quality is an issue, since most of the problems can be solved with secure coding practices. High code standards could be enforced with quality control techniques, such as *just-in-time* quality control[50].

Like with application vulnerabilities discussed by Gao et al.[27], system-level vulnerabilities also survive multiple updates in the code base[50].

The Common Vulnerabilities and Exposures (CVE) details website[4] gives an overview of all the vulnerabilities ever reported on Android as well. The main information from this datasource, as seen in Figure 3.2 (a), is that the overall number of reported vulnerabilities per year has been declining since peaking in 2017, while being the lowest since 2015. It can also be seen that the rise happened with the rise of smartphone popularity. This does not imply that the system became more insecure, but that a growing number of users became involved in reporting issues. However, the declining number of vulnerabilities could be attributed to an overall more secure platform.

Although Android as a platform is becoming more secure, only around 50% of users have a version from 2018 or later installed on their devices[1], and the other 50% are still prone to issues which may have already been patched.

CVE details[4] also notes that only 2 known exploits of the 2563 reported vulnerabilities[2] have been reported (as of April 2020). This can mean that most vulnerabilities are exploitable in theory but have not yet been exploited in practice. It can also mean that some exploits go unreported. Figure 3.2 (b) shows the total numbers of vulnerabilities by their type. The most common ones are "Code Execution" and "Overflow" vulnerabilities. Both of these types can be used to access and leak sensitive data, however, another two common types, "Gain Privilege" and "Gain Information" which are fourth and fifth most common respectively, directly endanger sensitive data.

Other sources such as androidvulnerabilities.org[3] and OWASP mobile top 10[4] have not been updated since 2015 and 2016 respectively, and as such are not taken into consideration.

[1]`https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide` (14.03.2020)

[2]CVE Details states that there are 2563 known vulnerabilities, however, the number of vulnerabilities in Figure 3.2 (b) is 2304. This may imply that there are uncategorized vulnerabilities missing from the graph.

[3]`http://www.androidvulnerabilities.org/` (14.03.2020)

[4]`https://owasp.org/www-project-mobile-top-10/` (14.03.2020)

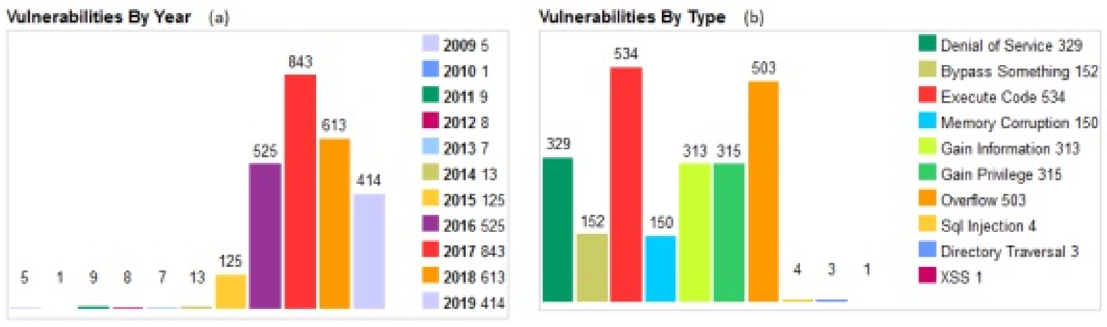


Figure 3.2: Vulnerabilities on Android by year (a) and by type (b) (Modified from [4])

This thesis aims to create a solution which detects possible private data leaks in real applications. Since there are only two known exploits on the CVE details website[4], it cannot be expected that all applications that leak private data exploit these vulnerabilities. Therefore, the vulnerabilities discussed in existing data leak detection solutions have to be considered. These vulnerabilities are discussed in the following subsections.

3.3 Data Leak Vulnerabilities in Android

This section presents the current most common vulnerabilities on Android and how each of them is able to work and be exploited.

The vulnerabilities selected here are not concrete issues in Android source code, but broader issues that cover multiple exploitable vulnerabilities, such as ICC leaks.

3.3.1 V01 - Inter-Component Communication (ICC)

Inter-Component Communication (ICC) is a feature of Android which allows applications to communicate and exchange data to reuse functionality between them. ICC uses intents to request an action from another component. If used maliciously, this can lead to compromised private data. The vulnerability works in such a way that a benign application can be used to leak private data. In order to do so, a malicious application A, which does not have access to a sensitive resource can ask another application B, which has access, to retrieve the information and give it to A. This way, A gets access to a sensitive resource without the required permission due to the lack of security in ICC[18],[46],[74],[21],[19].

3.3.2 V02 - Third-Party Analytics Libraries

While not a classic OS vulnerability, analytics libraries still collect and leak sensitive user data. Analytic libraries are needed so that developers can understand how their users preform in-app functions, e.g. what they do and what they do not do. Applications can either use their own code, or use third-party libraries. Since these third-party libraries share resources with the host application, they have the same access to sensitive data. When a library collects needed information from a tracking point, it sends the data to the analytics company first, then to the developer. At this point, there is a possible data leak, where data is collected by the analytics company without the user's permission. Even if a user gave permission to the host application, it is possible that they have no knowledge about the analytics company. This classifies as a leak[51]. A possible scenario is that the host application calls a third-party library function, which retrieves a sensitive resource, the library then gives the private data to the host application and the host passes the data externally without knowing that it contains. The third-party library can in this case also be the leaker if it has malicious intent[34].

Aside from the analytics company collecting the data from this application, they may also have libraries in multiple applications on the same device, and may link the data in one profile using the IMEI number, for example, as a unique ID. Unlike web analytics where cookies and supercookies can be deleted, and IP address can be changed, it is relatively hard to change an IMEI number[51].

The general overview of how an analytics library works can be seen in Figure 3.3. To use an analytics library, a developer needs to register with the analytics company, then give their information, as well as information about their application. After registration, they get a unique key which identifies their application, and the analytics SDK which is added to the application (2). The SDK is added to the build path, and the data about the SDK are added to the Android manifest file along with the required permissions for the SDK (3). The library is initialized on application start and APIs of the library are invoked typically on each onResume() and onPause() of the application as well as any specific analytics the developer wants (5). When the users use the application (4), their in-app actions are recorded and sent to the analytics company (6), which typically has a user interface for the developer to view the records (7), which can be used to enhance user experience (1). Optionally, the analytics company may send the collected data to an advertising company (8) which can send targeted advertisements to the application (9).

3.3.3 V03 - Inter-App Privacy Leaks

If an application is not permitted to access a sensitive resource, but it gains information through a data medium of another application and sends it to an external site, it came to a inter-app privacy leak. Examples of such data mediums are unsecured SQLite databases, shared preferences, content providers and internal storage. Shared preferences are the information which an application saves so that every component that the application has, can access it at any time, even if the application restarts, or the device reboots (e.g. cached data). Even though the use of readable and writable mode (i.e. accessible by other applications) has been deprecated since API level 17 (current level as of April 2020 is 29), there still are applications which use this mode[38]. Therefore, malicious applications can look for such files and extract the possibly sensitive data, which can be virtually anything.

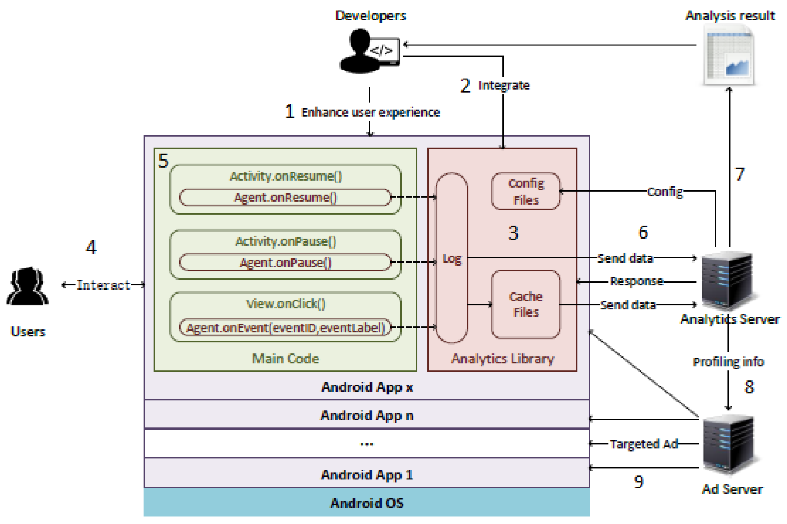


Figure 3.3: Structural overview of a mobile third-party analytics library. (Taken from [51])

One example of a private data leak through shared preferences can be seen in Figure 3.4. The myLocation application has access to fine and course location permissions, reads the location and writes it to a preference .xml file. The recorder application reads the location from the preference .xml file and writes it to external storage.

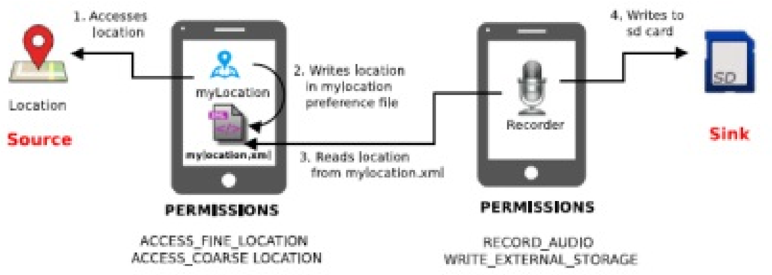


Figure 3.4: Privacy leak during inter-app information sharing via readable shared preferences (Taken from [38])

3.3.4 V04 - Cache File Privacy Leakage

Android creates a directory in the system for each application, that is only accessible by the application. The data generated by an application will be stored in this directory. However, due to improper handling of data, any data can be stored outside of this directory and in a publicly accessible one. For example: an application developer may call the wrong API during the storing of data, or if the device has limited internal storage and thus needs an external SD card, some cache of the application may be stored there. The data on the SD card is accessible by all applications. If the data is not encrypted or obfuscated in any way, it may be read by a malicious application, and lead to a leak of data[44]. In this case, the vulnerability lies mostly in the application storing the data, and not Android itself, however, the exploitation of this vulnerability can be detected.

3.4 Leak Hiding Techniques

To avoid detection by static analysis tools, malicious applications use obfuscation techniques to hide their behavior. Faruki et al. researched the so-called "stealth" malware techniques[25]. There are also legitimate uses for each obfuscation technique, such as protection of source code from reverse-engineering and copyright infringement, that is why the detection of a hiding technique on its own is not enough to label a potential leak. The following list is a collection of code obfuscation/hiding techniques, however, it is not a comprehensive list and only contains some techniques which are commonly employed by malware:

- H01 - Junk Code Insertion and Opcode Reordering - these techniques alter the signature of the application by adding non-executable code and re-ordering existing code to avoid signature-based malware detection.

- H02 - Altering Control-Flow - since some anti-malware uses application data flow as a signature, these techniques are used to alter the flow by adding extra steps to avoid detection.

- H03 - String Encryption - string literals used in code are encrypted and cannot be read unless the application is running and the strings are decrypted, thus making static analysis harder.

- H04 - Class Encryption - a technique which is commonly used to hide license-checks and paid downloads. It encrypts entire classes which work on these parts of the application.

- H05 - Resource Encryption - the resources folder, assets and native libraries can also be encrypted and used to hide information which can only be decrypted at runtime.

- H06 - Reflection APIs or dynamic code loading - this technique can be used to hide sensitive Android API calls within malicious applications. Reflection is normally allowed in order to make method invocation using strings, and the creation of programmatic class instances possible. However, the literal strings used can be encrypted and thus make it infeasible to detect malicious API calls statically.

- H07 - Package, Class and Method Renaming - similar to the previous example in H06, by renaming parts of the application which use known malware methods, this technique avoids signature-detection.

An example of an application using hiding techniques, in this case obfuscation and encryption to hide privacy leaks can be seen in Listing 3.1. The code first gets the Android ID through Reflection in lines 3 and 4. Then it hashes the ID (lines 7-9), XORs it with a randomly generated key (lines 11-16) and stores it in a JSON file (lines 18-20), which is then encrypted (lines 22-24). The enrypted file and the XOR key are then sent via an HTTP POST request (lines 26-29)[24]. The detection of this leak can be missed, if for example: the code is loaded dynamically, is executed through multiple components or misses the data flow through hash functions[24].

Listing 3.1: Snippet of code leaking Android ID using obfuscation and encryption (Taken from [24])

```
 1 StringBuilder json = new StringBuilder();
 2 // get Android ID using the Java Reflection API
 3 Class class = Class.forName("PlatformId")
 4 String aid = class.getDeclaredMethod("getAndroidId",
 5    Context.class).invoke(context);
 6 // hash Android ID
 7 MessageDigest sha1 = getInstance("SHA-1");
 8 sha1.update(aid.getBytes());
 9 byte[] digest = sha1.digest();
10 // generate random key
11 Random r = new Random();
12 int key = r.nextint();
13 // XOR Android ID with the randomly generated key
14 byte[] xored = customXOR(digest, key);
15 // encode with Base64
16 String encoded = Base64.encode(xored);
17 // append to JSON string
18 json.append("O1:\'");
19 json.append(encoded);
20 json.append("\'");
21 // encrypt JSON using RSA
22 Cipher rsa = getInstance("RSA/ECB/nopadding");
23 rsa.init(ENCRYPT\_MODE, (Key) publicKey);
24 encr = new String(rsa.doFinal(json.getBytes()));
25 // send the encrypted value and key to ad server
26 HttpURLConnection conn = url.openConnection();
27 OutputStream os = conn.getOutputStream();
28 os.write(Base64.encode(encr).getBytes());
29 os.write(("key=" + key).getBytes());
```

An example of a typical reflective call that defeats static analysis can be seen in Figure 3.5. In this case, Class A calls the reflection targets, Classes B, C or D through reflection API, which loads the required class dynamically and returns an instance to Class A. This way, the call cannot be resolved through static analysis as the it is not part of the application bytecode, but is loaded dynamically[11].

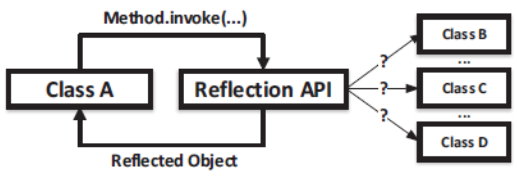


Figure 3.5: An example of reflective calls used to defeat static analysis (Taken from [11])

3.5 Research of Existing Solutions Which Mitigate Data Leak Vulnerabilities

This section presents existing data leak detection solutions. For every solution: the basics of its functionality, what results they achieved, as well as their main limitations are presented. The limitations of every solutions are mentioned because they will serve as a reference point as to which parts of certain solutions need improvement. The goal of the proof-of-concept solution developed for this thesis is to improve on an existing solution. For every solution, these limitations are noted and if it feasible to improve upon them, then they are selected as a basis for the proof-of-concept solution.

3.5.1 S01 - FlowDroid

FlowDroid by Arzt et al.[15] is a fully context, object and flow-sensitive static taint analysis tool. It is one of the first and most influential tools in the field of data leak detection. FlowDroid extends the Soot[41], a popular framework to analyze Java based applications, which gives it a foundation for precise analysis. FlowDroid further uses a scalable, multi-threaded implementation of the Interprocedural finite distributive subset (IFDS) framework.

The architecture of FlowDroid can be seen in Figure 3.6. After unpacking the APK, it analyzes the application to find lifecycle and callback methods and calls to sources and sinks. This is achieved through the parsing of various Android-specific files, such as: the Android manifest file (1), the .dex files (2) and the layout XML files (3). After the analysis, FlowDroid generates a dummy main method using the compiled list of lifecycle and callback methods (4). The dummy main method is needed, since Android applications do not have a single entry point (e.g. like a Java program), but multiple activities and services which serve as entry points. Through the main method, a call graph and an inter-procedural control-flow graph (ICFG) are generated (5), which simulate real-world use, creating random paths through the activities and services.

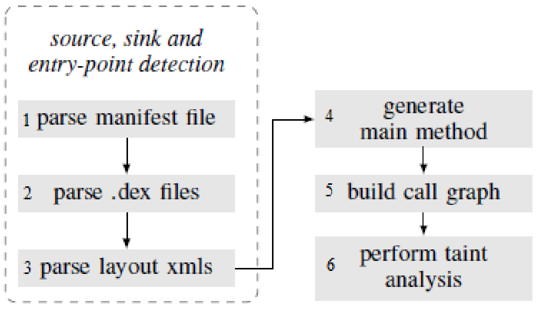


Figure 3.6: Overview of FlowDroid (Taken from [15])

An example of a control-flow graph (CFG) of a dummy main method can be seen in Figure 3.7. Only the parts of the lifecycle which can occur in runtime according to the configuration XML files are included. Disabled activities and callback methods which belong to them are not included. The example shows a generic activity lifecycle with a *sendMessage* callback. The p represents a *predicate* which FlowDroid cannot evaluate statically, so it considers both branches which depend on the predicate equally.

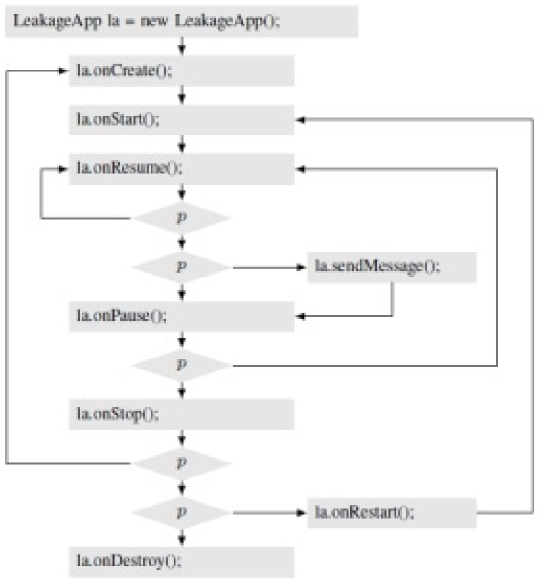


Figure 3.7: FlowDroid CFG for dummy main method (Taken from [15])

Beginning with the detected sources, FlowDroid's taint analysis (6) then tracks tainted data by going through the ICFG. FlowDroid generates a list of sources and sinks using a machine learning approach Arzt et al. developed previously. Finally, FlowDroid reports all flows from sources to sinks it discovered. These reports contain full path information. To obtain this information, the implementation links data-flow abstraction objects to their predecessors and to their generating statements. The FlowDroid reporting component can then fully reconstruct a graph of all assignment statements that might have caused a taint violation (i.e. a data leak) at a sink[15].

Evaluation: FlowDroid achieved 86% precision and 93% recall in DroidBench tests[15].

Limitations: due to it being static analysis only, it can only resolve reflective calls if the arguments are string constants. There are also callbacks FlowDroid is unaware of, e.g. callbacks through native methods. It is also unaware of multi-threading, since it assumes all threads to execute in a sequential order.

Due to this, improving upon FlowDroid's native and reflective call resolution is one potential goal the proof-of-concept solution could set.

3.5.2 S02 - IccTA

IccTA by Li et al.[46] is a tool which aims to detect ICC leaks. It uses a two-part approach: ICC link extraction and taint flow analysis.

The overview of IccTA can be seen in Figure 3.8. It uses the same foundation as FlowDroid[15], using Soot[41] with Jimple (Soot's internal representation of a Java/Android application)[46].

In the first step (1), IccTA transforms the .dex bytecode into Jimple. In the second step (2.*), IccTA extracts the methods using ICC (2.1), and the application's target components (2.2) to create ICC links (2.3). In step 3, IccTA stores the links and all the collected data, such as: the ICC call parameters or Intent Filter values, into a database. Using the ICC links, step 4.1, modifies the Jimple representation to connect the components to enable data-flow analysis between components directly. In step 4.2, IccTA uses a modified version of FlowDroid[15], to build a complete control-flow graph of the whole Android application. This allows propagating the context, for example: the value of intents, between Android components and achieving a highly precise data-flow analysis. Finally step 5, stores the reported tainted paths (i.e. data leaks) into a database[46].

Both steps (3) and (5), store all the results like the ICC methods with their attribute values such as the URIs and intents, target components with their intent filter values, the ICC links and the discovered ICC leaks into a database. By saving this data, the application needs to be analyzed only once and later, the results can be reused (e.g. if an application is deleted and reinstalled)[46].

Evaluation: In the experimental results, IccTA achieved a precision score of 96.6% and a recall of 96.6% on DroidBench[15].

Limitations: Same as with FlowDroid[15], IccTA can only resolve reflective calls which use string constants, it disregards multi-threading and fails on native calls. It also does not handle all ICC methods, and cannot resolve complicated string operations, and string analysis may cause false alarms.

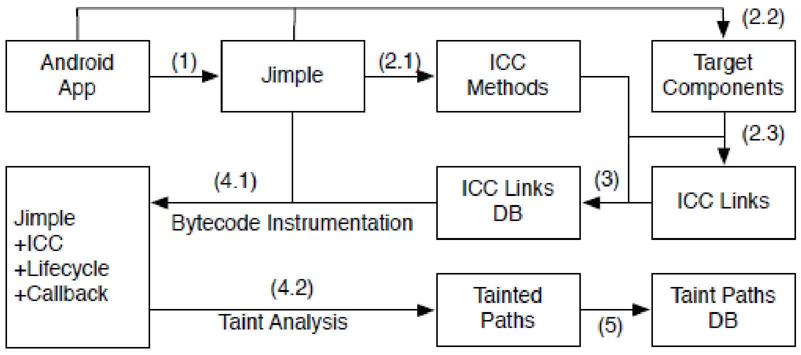


Figure 3.8: Overview of IccTA (Taken from [46])

Similar as with FlowDroid's limitations, these are also potential goals for the proof-of-concept solution. It also only detects one type of vulnerability, which the proof-of-concept solution can improve upon.

3.5.3 S03 - MirrorDroid

MirrorDroid by Rumee et al.[62] aims to cover all data leak vulnerabilities that are leaked by a single application. It does not cover a specific vulnerability. To achieve data leak detection, the MirrorDroid executes two instances (i.e. main and mirrored instance) of an application in parallel. They both get the same input, except for sensitive data (e.g. IMEI, Contacts), which is altered in the mirrored instance. If the output of the application differs between the two instances, there must be a leak of information about the sensitive input. MirrorDroid's approach is summarized in Figure 3.9. There we can see that the two instances are fed different data in key resource access, and if the sink output differs, application must be malicious.

In order to run the mirrored execution, Rumee et al. modified the Dalvik interpreter so that it can handle new instructions. The main feature is that each instruction is repeated to the mirrored instance before a new instruction is fetched. This is done to ensure that the only difference between the two executions is the sensitive data and that even non deterministic inputs (e.g. random number generators) stay the same.

Evaluation: MirrorDroid recorded successful detection both within applications with known data leaks (Malware Dataset with 449 samples), and popular Play Store applications (50 randomly selected applications from top 500 free applications). In both sets, they recorded no false positives. In theory, the main drawback of MirrorDroid should be the overhead of running two instances, however, by duplicating only the needed parts of applications, the overhead is recorded to be only 8.2%.

Limitations: one of the main drawbacks is the required overhead, however, it is kept to a minimum. MirrorDroid also needs to run once per sensitive data type which makes it easy to miss a leak if one data type is not tested.

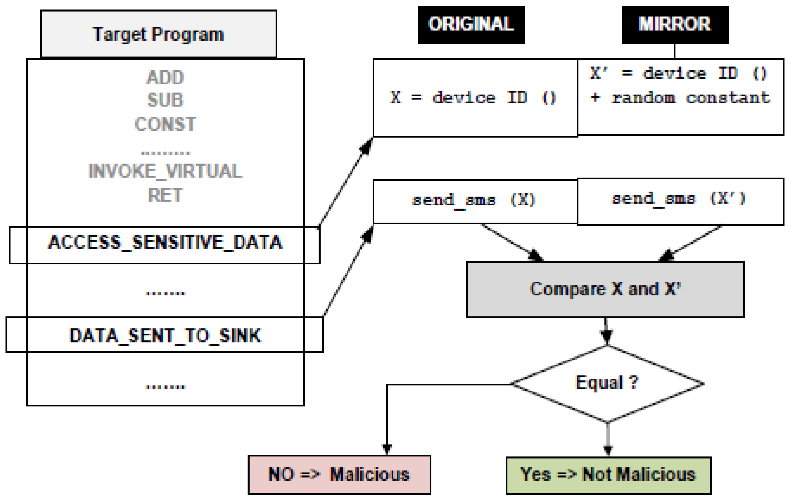


Figure 3.9: MirrorDroid approach (Taken from [62])

Since MirrorDroid is a dynamic analysis solution, the limitations are not considered potential goals for the proof-of-concept solution.

3.5.4 S04 - HybriDroid

HybriDroid[21] is a data leak detection approach which combines *static* and *dynamic* detection techniques into a *hybrid* approach. Its architecture has four steps: 1. Basic app model extraction (static analysis); 2. Run-time information collection; 3. Advanced app model building; 4. Private data leakage detection (Figure 3.10). Steps 1-3 build an application behavior model, and step 4 detects vulnerabilities. The app model in this case is defined as a set of: Components, intents, intent filters, permissions and paths (i.e. data flows). Both the basic and the advanced app model follow this definition.

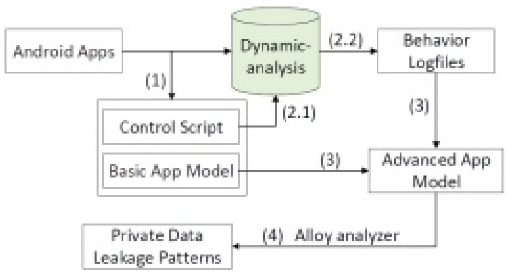


Figure 3.10: HybriDroid architecture [21]

The basic model is the result of a static analysis in which the application APK file is decompiled and model elements are extracted, however, due to static analysis shortcomings in dynamic application features (e.g. resolving string operations and reflection techniques), the model has defects, most notably the paths. To collect run-time information needed for dynamic analysis, the authors modified the Dalvik VM in order to run monitor modules in Android emulator, which wrote the behavior into log files. The API Hook Module intercepted function calls and collected function argument values, while the Taint Tracer Module collected information flow paths using tainted data. Upon collecting data with these two monitors, the logs are analyzed and the advanced model is built i.e. the basic model is extended. The two models can have conflicting values of some elements (e.g. Intents), so the refining policy, that decides which parts to keep is crucial. In this case, HybriDroid defines a context for the components and decides whether to add a new element to the model, to refine it, or to discard it.

To detect concrete data leaks, HybriDroid uses lightweight formal analysis techniques. These techniques automatically detect private data leaks. If one is detected, it is reported to the user together with the root cause, source and receiver components of the private data.

Evaluation: HybriDroid ran comparisons to static and dynamic analysis tools separately. Static analysis tools have a standardized benchmarking tool: DroidBench[15], a set of applications with fully known vulnerabilities, which was developed for the purposes of FlowDroid[15]. HybriDroid achieved 97.8% precision and 90% recall. For dynamic analysis, Chen et al. tested HybriDroid on 50 top ranked applications on Google Play Store, detecting leaks in 5 of them.

Limitations: The article[21] mentions the inability of analyzing native code as the only limitation. However, it is feasible that there are limitations which the authors left out, which are not known due to lack of openly available source code.

HybriDroid's main limitation are native calls, which are selected as potential goals for the proof-of-concept solution. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.5 S05 - AndroidLeaker

AndroidLeaker[74] is another *hybrid* detection approach. Figure 3.11 showcases the main elements. First, static taint analysis is conducted. The APK file is decompiled and analyzed for possible privacy leaks (1 and 2). If it finds "Leakage Points" it logs them and gives a failure grade to the application (3). If no leaks are found, message security labels (4) and dependent send permission (5) are generated by the taint analysis. These elements are used during instrumentation (6), i.e. the construction of an instrumented APK (7), on which the dynamic analysis is conducted. The runtime analysis (8) checks inter-component communications, detects and prevents potential private data leaks. The final behavior is recorded and reported to the user (10).

AndroidLeaker uses the framework it is based on to create a call graph from an application's bytecode to conduct data-flow analysis. It also uses a generic inter-procedural,

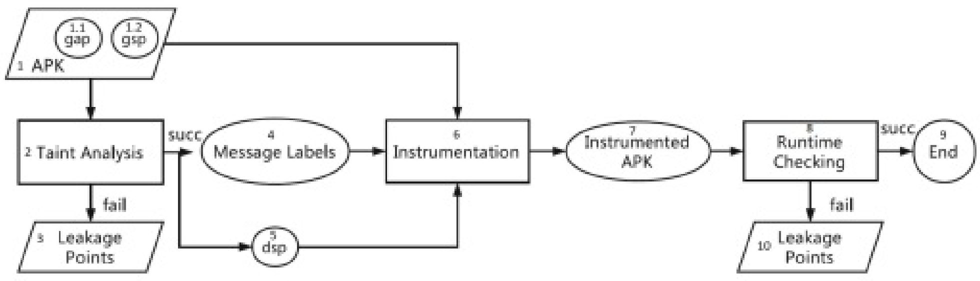


Figure 3.11: AndroidLeaker architecture overview [74]

finite, distributed subset problem for inter-procedural data flow analysis. To conduct the taint analysis: sources and sinks; callback functions; alias information; and entries and permissions are collected. This is then fed to a flow-, context-, field- and object-sensitive taint analysis. For the runtime analysis, AndroidLeaker runs checks for every inter-component communication call to see if an application is trying to access a permission it itself does not have, through the second application. If this check fails, the communication is canceled.

Evaluation: To evaluate the results, AndroidLeaker used DroidBench as well. Their overall results are: 82% precision and 69% recall. To test the leaks detected dynamically, they used proprietary tests, which they passed.

Limitations: compared to other solutions, it has a relatively low recall.

AndroidLeaker's main limitation is the relatively low recall even with dynamic analysis. The proof-of-concept solution aims to achieve equal or greater overall results in DroidBench with static analysis only. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.6 S06 - APPLADroid

SniffDroid

SniffDroid by Jain et al.[38] aims to detect inter-app privacy leaks on Android.

The architecture of SniffDroid is shown in Figure 3.12. *Smali Code Generator (1):* First, the APK of an application is disassembled and converted into smali[5] and the Android manifest file is extracted. The *Parser (2)* extracts the components, permissions and package names of the application from the manifest file. The *Extractor (3)* takes information about the shared preference files (e.g. their modes, methods and calls). *Graph Generator (4):* Graphs are used to model how applications store and retrieve information from shared preferences. The generator takes application components from

[5]Smali is a readable, intermediate representation of .dex bytecode

the parser and information about shared preference files from the extractor and generates a graph for each application connecting application components with the corresponding preference files they call. *Union of App Graphs (5):* Application graphs are used as input and they are unified to check if they share information through preference files. *Pruning the Union of App Graphs (6):* The graph is then pruned to only leave components connecting with shared preference files. *Construction of Application Automaton (7):* The pruned graph is converted into the Application Automaton by classifying graph vertices as the initial and final states. Similarly to FlowDroid[15], SniffDroid creates a dummy main state as the entry point. The remaining vertices are defined as final states. The main state is connected to every other state in the Automaton. The state transitions are labeled with permissions. *Policy Automaton (8)* is represented as an Automaton with an initial, intermediate and final states which describes the potential flow of sensitive data from source to sink permission, including intermediate permissions. *Intersection of Automaton (9):* The Application Automaton is intersected with Policy Automaton to detect leaks. If the intersected automaton possesses a final state, the applications are suspected of having a leak[38].

Evaluation: SniffDroid's experimental evaluation did not include any standardized benchmarks. They did however discover that 81 of the 240 applications tested used private and 8 used readable shared preferences[38].

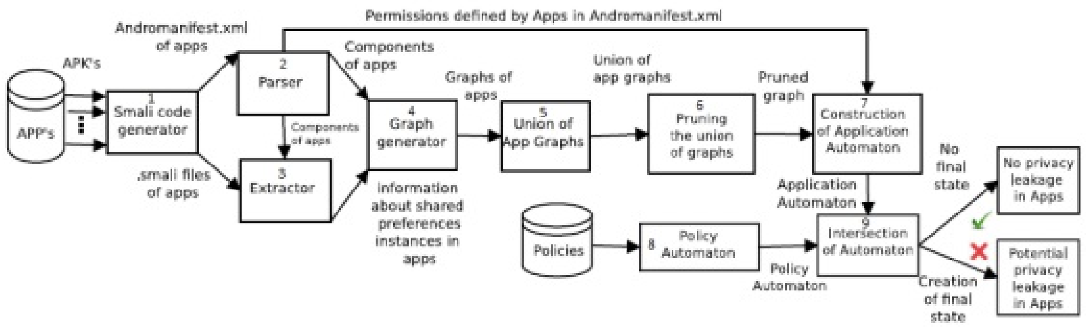


Figure 3.12: Workflow of SniffDroid (Taken from [38])

Limitations: SniffDroid takes a major assumption with assuming that the applications they analyze are free of obfuscation, encryption and reflection. However, malicious applications are likely to use such techniques to avoid detection[38].

APPLADroid

APPLADroid is an extension of SniffDroid[38] developed in 2019 by Jain et al.[39]. The core architecture of the approach is the same, however it extends SniffDroid in following ways:

- It adds support for all ICC mediums (compared to only shared preference files in SniffDroid).

- Improves performance to be more scalable and fast by constructing a more high-level representation of data flows.
- It stores the malicious flows it discovered in a database, thus supporting reusability and further saving time.
- It conducts control and data flow, as well as taint analysis to capture malicious flows of sensitive data.

Evaluation: They claim results of 100% recall and precision in DroidBench[15] tests[39].

Limitations: APPLADroid does not mention any inherent limitations it has, and neither does it mention fixing the SniffDroid limitations of assuming that the applications are free of obfuscation, encryption and reflection, which implies that they still stand[39].

For SniffDroid/APPLADroid, the main limitations are that they do not analyze obfuscation, encryption and reflection, all of which are considered potential goals for the proof-of-concept solution. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.7 S07 - X-Decaf

X-Decaf or Xposed based Detection of Cache File[44] by Li et al. is a tool developed for detecting private data leaks caused by unsecured cache files. Xposed[6] is a framework for modules that can modify the Android OS and applications without modifying any APK files.

The architecture of X-Decaf can be seen in Figure 3.13. It consists of the following major parts: *Establishing Sensitive Library (1)*: To analyze the relationship between privacy data and Android system API, X-Decaf decompiles applications and finds the system APIs which create and propagate sensitive data. This establishes a sensitive library (1.1). The sensitive library includes APIs which involve voice, pictures and video data, since they mainly analyzed social networking applications. Each API is tagged with the following information: type of private data, the needed permission, class containing API call, method containing API call. *Dynamic Tracking (2):* The sensitive function in the sensitive library is the object being tracked during runtime in this step. This module first sends a request to the function with specific data, and monitors this data through a hook created by Xposed.

Taint Marking (3): This module is mainly used for cache file filtering and data tainting. The major steps are presented as follows: *Cache File Filter:* X-Decaf filters cache files based on privacy data to improve efficiency and accuracy (e.g. cache files with ".jpg" extension are classified as sensitive). *Taint Mark:* X-Decaf taints the cache file

[6]`https://repo.xposed.info/module/de.robv.android.xposed.installer` (22.03.2020)

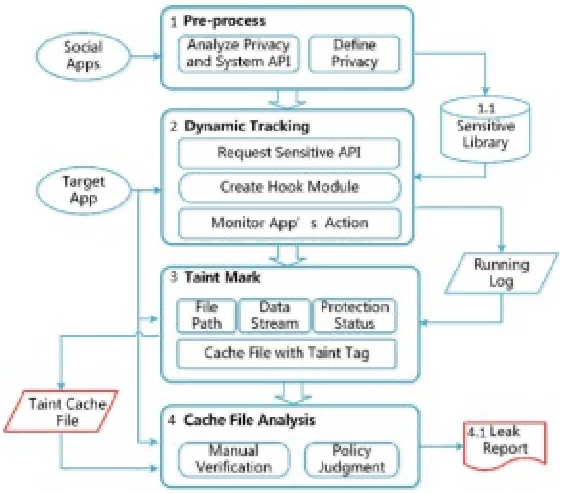


Figure 3.13: X-Decaf architecture (Taken from [44])

with three attributes: privacy type, file hash and file protection status (protected/unprotected). *Cache File Analysis (4)* is done through *Manual Verification* and *Policy Judgment*. *Manual Verification:* X-Decaf first taints data and manual checks determine if the application managed the cache files. Then, cache files are monitored to see if they exist or have been removed. According to results, the correct policy is performed. *Policy Judgment:* X-Decaf monitors changes of protection status, file path, life-cycle of these cache files with taint, and outputs a leak report (4.1) corresponding to leakage criteria[44].

Evaluation: Their experimental analysis did not use any standardized benchmarks. They did however show that most social networking applications have privacy leakage issues[44].

Limitations: The paper[44] does not mention any limitations.

The limitation of X-Decaf is the fact that it only targets one vulnerability. The proof-of-concept solution's potential goal is to be versatile and detect two or more vulnerabilities along with hiding techniques. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.8 S08 - DroidRA

DroidRA by Li et al.[47] is a tool focusing on detecting reflection based hiding techniques mentioned in Section 3.4. It is not a standalone data leak detection solution, and it serves as a tool which helps other data leak detection solutions discover more leaks. The tool is developed by the authors of IccTa[46] and has first been proposed in [45].

Figure 3.14 show the architecture of DroidRA. The approach consist of three modules:

(1) The Jimple Preprocessing Module or JPM prepares the application to be inspected, i.e. it decompiles the application, constructs all the entry points and prepares the heuristics for class loading.

(2) The Reflection Analysis Module, or RAM, finds reflective calls and retrieves the values of their parameters (i.e., class/method/field names). All reflection target values that are discovered are saved for the researchers who use DroidRA in their own tools and approaches.

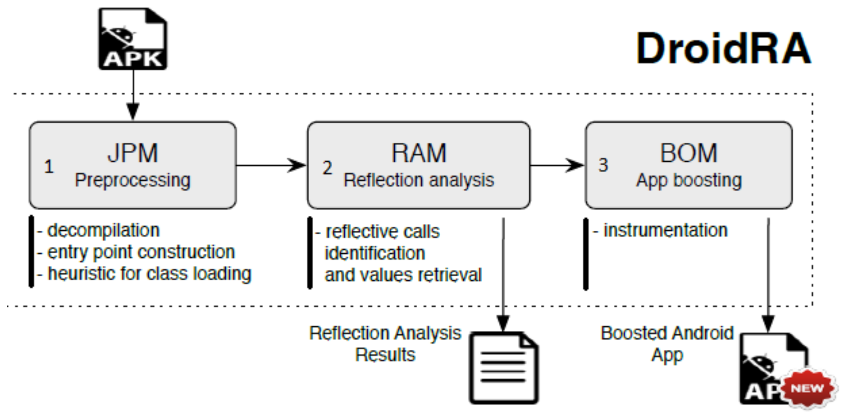

APK
DroidRA
1 JPM Preprocessing
2 RAM Reflection analysis
3 BOM App boosting
- decompilation
- entry point construction
- heuristic for class loading
- reflective calls identification and values retrieval
- instrumentation
Reflection Analysis Results
Boosted Android App
AP NEW

Figure 3.14: The overview of DroidRA (Taken from [47])

(3) Using the information from the RAM module, the Booster Module or BOM instruments the application and creates a new one where reflective calls are "boosted" or enhanced with standard Java calls. The goal of this module is to create an equivalent application which can be analyzed more precisely by the state-of-the-art tools[47].

Evaluation: DroidRA on its own is not a leak detection solution so it cannot be evaluated as such without it being used as part of another solution.

Limitations: The main limitations of DroidRA are that it cannot detect dynamically loaded code that is not present in the APK (i.e. if it is loaded from an external server) or code that is obfuscated in some other way (e.g. through encryption).

Since DroidRA is not a data leak detection solution, a potential goal is to incorporate DroidRA's features into an existing solution.

3.5.9 S09 - DroidRista

DroidRista by Alzaidi et al.[13] is a static data flow analysis tool for Android applications. It aims to solve ICC links, reflective calls, and implicit data flows. DroidRista is a tool that combines three approaches (S08 - DroidRA[47], Soot[41] and S01 - FlowDroid[15]), modifies one approach (IC3-Modified[55]), and adds a new component (Instrumentor).

The DroidRista workflow can be seen as in Figure 3.15. First, the application's APK file is used as input for DroidRA (1), generates a boosted APK through instrumentation. Then, it uses IC3-Modified (2), a version of the IC3 tool, to extract the ICC links from Android applications. IC3-Modified first extracts all methods used for inter-component communication with all potential components by parsing the Android manifest file. Then, it matches the components with the destinations and stores them in a database to allow for reuse. Next, through Soot (3), Dalvik bytecode of the boosted APK is converted into Jimple. It modifies the Jimple code to immediately connect components for data flow analysis and enable the handling of ICC and reflection techniques. To support reflection, DroidRista uses DroidRA again, since DroidRA can use instrumentation to resolve reflective call targets, reveal application behaviors and map the calls to the corresponding Java calls, while maintaining the existing call graphs.

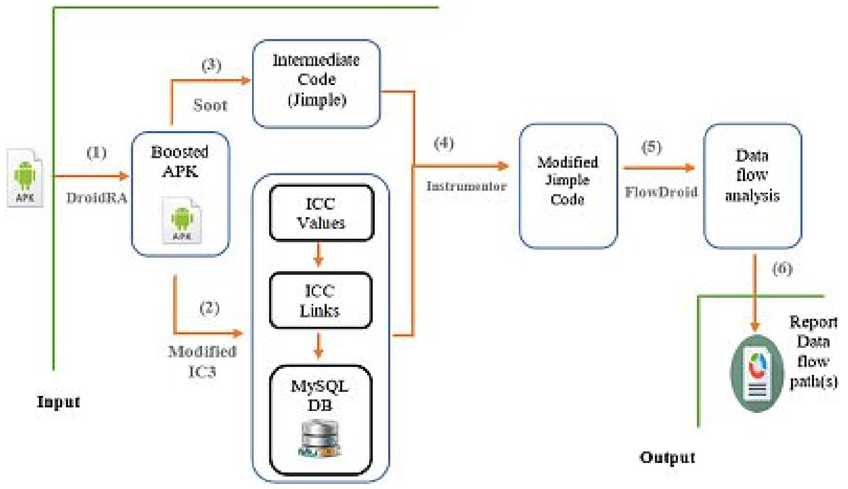


Figure 3.15: DroidRista workflow (Taken from [13])

To support ICC, DroidRista uses the Instrumentor (4), which modifies the Jimple code and substitutes each ICC method with an instance of the destination component with the given intent. The modified code is then given to FlowDroid (5), which generates a flow control graph of the application, which enables data flow analysis. Once the analysis is completed, the data-flow paths are reported (6)[13].

Evaluation: To evaluate, DroidRista used the IccTA[46] branch of the DroidBench test suite[15], the original DroidBench and ICC-Bench[2] and achieved 98.4% precision and 96.9% recall overall.

Limitations: DroidRista mentioned the inability to detect leaks in native code as a limitation[13].

As with FlowDroid and HybriDroid, DroidRista also does not detect leaks in native code, so it is a potential goal for the proof-of-concept solution. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.10 S10 - AppLance

AppLance by Liang et al.[48] is a leak detection solution that focuses on analyzing packed applications. Application packing is a technique which changes the bytecode of an application into new bytecode which is infeasible to analyze statically. Packed applications are also infeasible to modify or repackage, which makes instrumentation harder.

The approach of AppLance is to control variables. In each step, AppLance changes one variable and keeps the rest the same. This changes a multi-variable problem into a single-variable problem and it allows studying the effects of one variable on the whole application data flow. In this case, sources and interference factors are the variables and the sinks are the results. To detect a connection between a source and a sink, AppLance changes the value of a source, and if the value of the sink changes, there is a connection. This enables detection of data flow even if the communication in-between is encrypted. AppLance thus uses a black-box approach and only looks at how private data is accessed and where it is leaked to determine if the application has a possible private data leak.

The approach can be seen in Figure 3.16. It runs on a real Android device (i.e. not an emulator) and monitors the source and sink APIs for the analyzed application. It consists of three modules: the Reference Module (1), the Comparison Module (2) and the Analysis Module (3).

The reference module (1) uses dynamic binary instrumentation to insert a probe into the application and collect the source and sink APIs information when the application runs. This is used as a reference for the comparison module.

The comparison module (2) is the same as the reference module with the only difference being that the private data obtained from source APIs is modified. The reference and the comparison module collect information about the sink APIs and give it to the analysis module.

The analysis module (3) checks if the application leaks private data based on the difference between the reports of the two modules.

Evaluation: AppLance claims 96.2% accuracy in a hybrid test consisting of test cases from DroidBench[15], IccTA[46] and DroidRA[47], scoring higher than FlowDroid[15] which achieved 55% accuracy in non-packed applications. AppLance was not affected by packed applications and reached the same accuracy, however, FlowDroid was unable to detect any leaks in applications packed by all packers except one.

Limitations: Due to the black-box approach, AppLance leak detection is coarse, meaning it cannot specify which data flow leaks information. Code coverage is also an issue, and as it stands, their runner can only cover simple use cases. It also does not cover all source and sink APIs, but it does provide an interface for easier extension.

To improve on AppLance's limitation, a potential goal of the proof-of-concept solution is to be able to detect which types of data are being leaked. However, since this approach

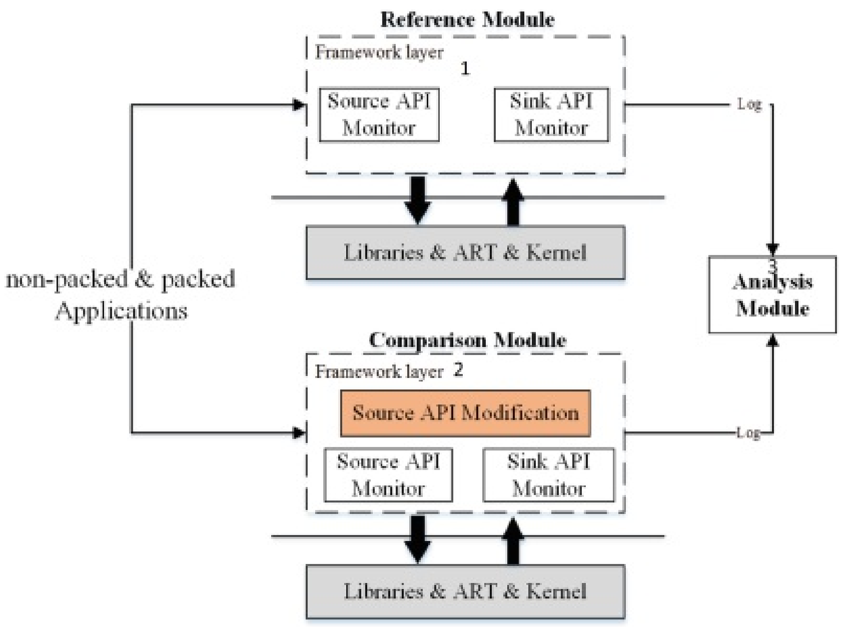


Figure 3.16: The overview of AppLance (Taken from [48])

does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.11 S11 - Fog Computing Solution

The Fog Computing Solution by Gu et al.[30] is a hybrid context-based privacy leakage detection method. Their main focus are healthcare networks and Personal Healthcare Devices (PHDs) based on Android.

The approach uses context analysis to create a detection scheme consisting of static privacy leakage analysis and dynamic privacy disclosure monitoring. The overview can be seen in Figure 3.17

First, static analysis is used to analyze how the permissions are mapped to the API functions, analyze the system, function calls, interface events and static taint propagation paths. The analysis itself is based on FlowDroid[15] and adapted for PHDs.

Then, dynamic monitoring uses four steps:

1. User information and system working status are collected in the user terminal, which constructs the context information.
2. In the fog, the private data is extracted and the data from the previous stem is analyzed to perform privacy leakage detection through access control.

3. If a leak is detected, the next transmission of the device is blocked, the flow of the detected leak is intercepted and the user is notified.

4. The information about the user and the leak is uploaded for future reference.

Evaluation: The approach used DroidBox[1] to evaluate the detection capabilities and managed to detect 2876 leaks, which is higher than the baseline of 2431 leaks detected by DroidBox itself.

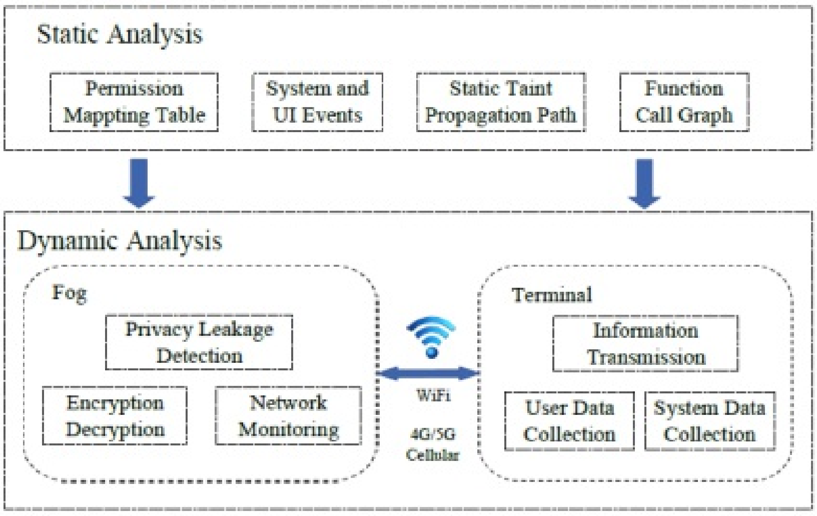


Figure 3.17: Framework of privacy leakage detection. (Taken from [30])

Limitations: Gu et al. mentioned no inherent limitations of the approach[27].

However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, because it cannot be used as a basis for the proof-of-concept solution.

3.5.12 S12 - Leak Detection Through API Call Logs

Leak detection through API call logs is a solution proposed by Ito et al.[37]. The overview of the approach can be seen in Figure 3.18. In order to record API call logs Ito et al. modified the Android OS, namely an emulator of Android so that an installed application (1) can output text files during its runtime (2). Their assumption is that malicious applications frequently access local data and communicate with outside servers, revealing data they might be leaking. The API call logs are analyzed through static analysis (3), details of which are not discussed in the paper. Based on the call logs, the applications are classified (4) in three categories: *No-Access*, *Proper-Access* and *Improper-Access*. *No-Access* means the application does not call any sensitive APIs, *Proper-Access* means an applications accesses private information along with its proper functionality and *Improper-Access* means that the goal of private data access is to leak the sensitive information.

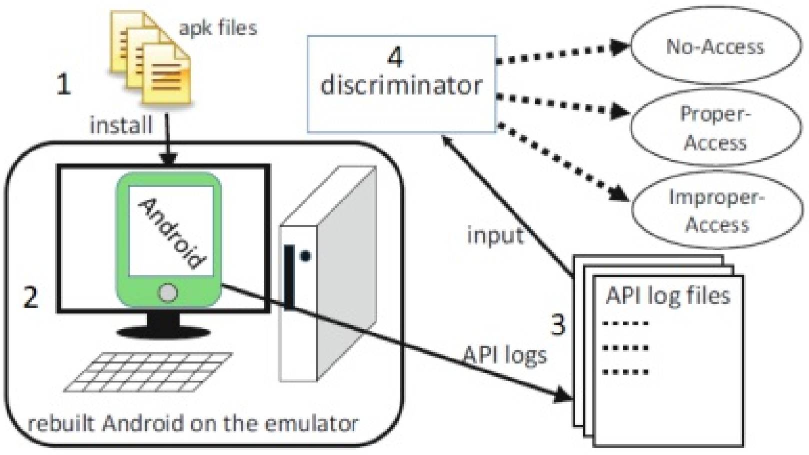


Figure 3.18: Overview of the solution. (Taken from [37])

Evaluation: No standardized benchmarks were used. Their experimental setup was proprietary and it yielded following results: on 42 tested applications Ito et al. noticed that 0.09% of the sensitive API *access* calls, 0.22% of sensitive API *send* calls and 1.59% of the sensitive *response* (i.e. display on screen) API calls were with the aim to leak information.

Limitations: Since this work was a proof-of-concept without an extensive evaluation, the limitations of the solution were not researched by Ito et al.[37].

With unknown limitations, it is not feasible to improve upon the solution, since it is unknown what needs improvement.

3.5.13 S13 - Agrigento

Agrigento by Continella et al.[24] is an approach developed in an attempt to detect leaks in applications which use obfuscation techniques to hide it. The overview of the approach can be seen in Figure 3.19.

Agrigento consists of two main phases:

1. The *network behavior summary extraction* is performed, i.e. the application is executed multiple times in an instrumented environment, while using the same values for sensitive values, to gather *network traces* and *contextual information.* This is done so that the context behind the differences in the runs can be determined and the *contextualized trace* can be created which says what the reason for the difference is. The differences can be due to e.g. timing values, random values and network values, and do not mean that there is a malicious intent behind them.

2. The application is run once again, with only the sensitive information being changed. By comparing the *contextualized trace* of the final run with the *network behavior summary* of the previous runs to determine whether there is significant difference. Agrigento first performs *differential analysis* to determine all differences, and a risk analysis to determine the severity of the differences and determine whether a sensitive data leak occurred.

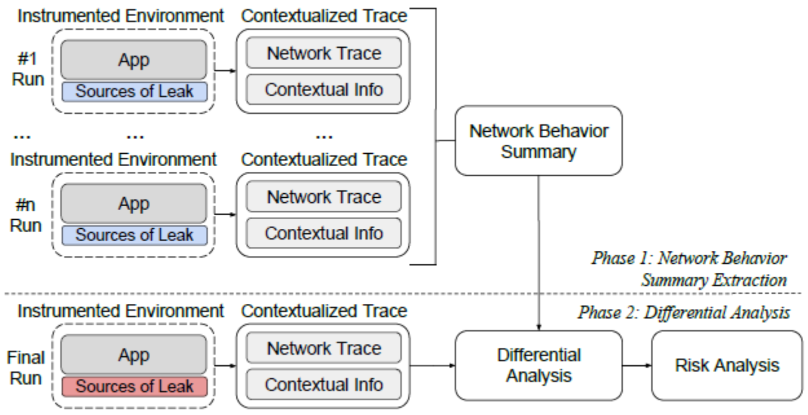


Figure 3.19: High-level overview of Agrigento. (Taken from [24])

Evaluation: The researchers did not use DroidBench for evaluation, as at the time of writing their paper, it did not have many dynamic-based test cases. They evaluated Agrigento by comparing it to what Continella et al. considered similar solutions - ReCon[58] and BayesDroid[67]. Compared to ReCon itself, Agrigento detected 165 applications ReCon did not, and ReCon detected 42 applications Agrigento did not detect to leak private data. Running both tools on the same network traffic dump, Agrigento detected 278 leaks, while ReCon detected 229. Compared to BayesDroid, Agrigento detected 21 applications leaking data, while BayesDroid detected 15, 10 of which were the same ones. A static analysis only tool, such as FlowDroid[15] only detected 44 leaks in the ReCon test case.

Limitations: Due to requiring the application to run, code coverage is an issue since the part of the application which is malicious may not get executed. This leads to false negatives. Agrigento does not handle covert channels as well, and the need for multiple runs gives it a higher comparison time than other approaches mentioned in this thesis[24].

A potential improvement that the proof-of-concept solution can make is high code coverage.

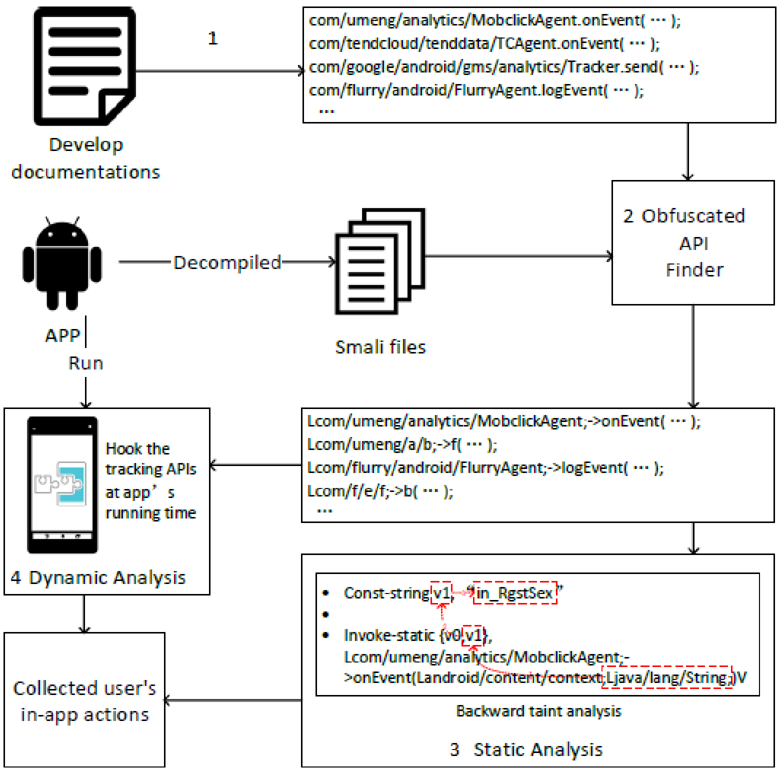
3.5.14 S14 - Alde

Alde by Liu et al.[51] was developed to detect data leaks caused by analytics libraries. It uses both static and dynamic analysis to detect analytics API calls and record the collected in-app data collected.

The overview of the solution can be seen in Figure 3.20.

The first step (1) in Alde is to collect the APIs used by the analytics library used in an application. This is done by downloading the library and decompiling it to find all the API functions.

The second step (2) is the Obfuscated API finder. The obfuscated tracking APIs are found through the method call graphs, since obfuscating method identifiers does not affect the graph. Alde first discovers the call graphs of the analytics library when it is not obfuscated, then it decompiles the target application, generates a call graph, prunes all calls that are certainly not part of analytics API (e.g. part of other known third-party library), and for each API from the original library, it compares the calls graphs.



```
com/umeng/analytics/MobclickAgent.onEvent( ··· );
com/tendcloud/tenddata/TCAgent.onEvent( ··· );
com/google/android/gms/analytics/Tracker.send( ··· );
com/flurry/android/FlurryAgent.logEvent( ··· );
 ···
```

```
Lcom/umeng/analytics/MobclickAgent;->onEvent( ··· );
Lcom/umeng/a/b;->f( ··· );
Lcom/flurry/android/FlurryAgent;->logEvent( ··· );
Lcom/f/e/f;->b( ··· );
 ···
```

```
• Const-string v1, "in_RgstSex"
•
• Invoke-static {v0,v1},
  Lcom/umeng/analytics/MobclickAgent;-
  >onEvent(Landroid/content/context;Ljava/lang/String;)V
```

Figure 3.20: System overview of Alde. (Taken from [51])

The third step (3) is the static analysis. The goal of this analysis is to find hardcoded information left by the analytics library. The analysis first decompiles the application, and analyzes the code to generate a control-flow graph. Then, it finds the code of the tracking APIs and the registers that store the function parameters. After that, it runs backward taint analysis by searching the decompiled code in reverse order. The analysis stops when Alde finds a constant assigned to the register or reaches a method that cannot be analyzed.

The fourth and final step (5) is the dynamic analysis. To analyze the application during runtime, it runs it with simulated use for five minutes, while monitoring the processes through an Xposed module which hooks into the tracking APIs of the analytics library. All function calls and parameter values are captured and stored. The results of the static and dynamic analysis are combined to get the final result.

Evaluation: Liu et al. did not use any standardized benchmarking tools, they did however have relevant findings. They found out that analytics libraries do collect e.g. sensor, contact data and the list of installed applications and send them to the analytics servers, however, the data was not shared with developers in raw format.

Limitations: Liu et al. share the most common drawbacks of hybrid solutions: They can only analyze the applications that their decompiler can decompile; their static analysis can't handle inter-component and inter-process leaks; their dynamic analysis cannot handle all the execution paths[51].

The proof-of-concept solution could improve on the code coverage and inter-process leaks, however, this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, since it cannot be used as a basis for the proof-of-concept solution.

3.5.15 S15 - Witness

Unlike all other solutions mentioned in this section, which detect generic vulnerabilities (e.g. ICC), Witness by [49] detects eight concrete vulnerabilities. These vulnerabilities include: *Content Provider Directory Traversal*, which allows attackers to access arbitrary files if the content provider of the target application uses openFile() interface and it does not check the URI; *Content Provider SQL Injection*, which allows attackers to inject malicious input into databases if the content provider lacks parameter validation; and *Content Provider Data Leaks*, which enables attackers to obtain internal data of a content provider if it does not define protection levels or permissions properly[49].

Witness itself is a hybrid approach to detecting vulnerabilities, consisting of static and dynamic analysis. The overview of the approach can be seen in Figure 3.21.

The static analysis consists of pattern construction (1), which is done based on the vulnerability's heuristic information, such as CVE descriptions and documentation for developers (since it detects various vulnerabilities, and is extensible for more vulnerabilities). The pattern itself is used to describe a vulnerability on seven aspects: components, attributes, entry functions, target functions, concerned functions, test cases template and trigger template. Entry functions, target functions and concerned functions create the API list (2). Similarly to other approaches, Witness extracts application features (3) from the metadata in the Android manifest file, smali files, control flow graph and the call graph. If witness fails to find the attributes of a vulnerability in the extracted features, the vulnerability is excluded from the analysis. If it finds attributes in the features, Witness constructs test cases (4) and a trigger (5) using the test and trigger templates of the vulnerability as well as the data extracted from the application.

The dynamic analysis runs the trigger to invoke the application (6). While the analysis is running, Witness changes the test cases that the trigger sends (7), while monitoring the entry and target functions of the application to trace the behavior through binary instrumentation techniques (8).

After running both phases, Witness analyzes the collected behavior log to verify if the application contains a vulnerability (9) and outputs a report (10)[49].

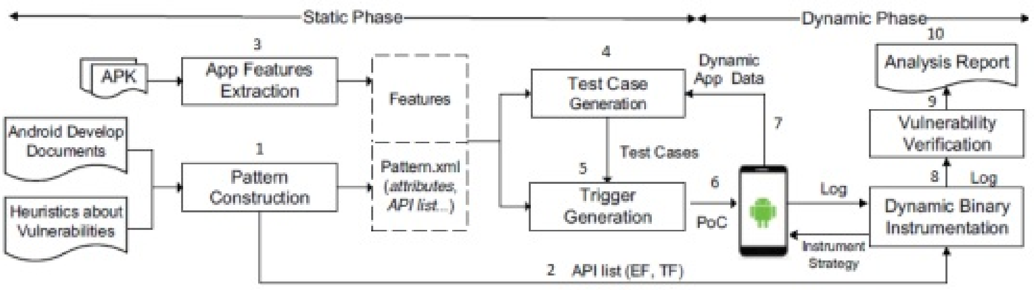


Figure 3.21: The architecture of Witness. (Taken from [49])

Evaluation: The authors did not use standardized benchmarks. Their preliminary analysis on 3211 applications, where they detected 243 vulnerabilities, showed that Witness was able to detect vulnerabilities without any false positives and false negatives, which was verified through manual inspection[49].

Limitations: The authors list two main limitations of Witness. The first one is that the analysis for a specific vulnerability is limited to the available public knowledge and manual pattern construction. This leads to false positives and a non-exhaustive search. The second limitations is that packed applications are not able to be analyzed with Witness[49].

The limitation to specific vulnerabilities could be improved upon by generalizing the vulnerabilities. However, since this approach does not have openly available algorithm description or source code, it is not feasible to improve upon this solution, since it cannot be used as a basis for the proof-of-concept solution.

3.5.16 S16 - DINA

DINA by [11] aims to detect inter-application (i.e. collusive) attacks. It is a graph-centered hybrid approach which consists of: a static analysis module that analyzes multiple applications at the same time to generate an Inter-Application Communication (IAC) graph and define Dynamic Code Loading (DCL) and reflection call sites, which are a target for dynamic analysis; a dynamic analysis module that augments the graphs created in the static analysis by adding new nodes and edges created dynamically; and the vulnerability analysis module that uses the graphs to identify potentially vulnerable paths[11].

An overview of the approach can be seen in Figure 3.22.

The **static analysis module** (1) tries to identify reflection, DCL and all IAC functionality of all the applications in the analysis. It is a classloader-based analysis system based on JITANA[68], that is more scalable than compiler-based approaches, such as the ones based on Soot[41]. DINA generates two graphs for each application, the Method Call Graph (MCG), which shows the call relationships between methods in the applications, and the Instruction Graph (IG), which shows the control and data flow information for a method.

The analysis is done on the Dalvik bytecode and it consists of: *Preprocessing*, which decompiles the APKs and extracts intent filter information and generates the MCG and IG; *Reflection/DCL analyzer*, which identifies the DCL and reflective calls using the MCG, and identifies the applications that need to be executed in the dynamic analysis module; and *Static IAC analyzer*, which identifies the IAC paths through string matching over IGs to match the intent action strings of all applications. If a match is found, an edge connecting multiple applications is added to the graphs.

The **dynamic analysis module** (2) performs incremental dynamic analysis for each application that contains reflective or DCL calls. The analysis consists of two steps: *Resolving reflection and loading new codes*, which runs the applications and captures their dynamic behavior using the reflection details extracted previously. This is used to augment the control and data flow graphs; and *the dynamic IAC analyzer*, which executes components to create the dynamic analysis graphs by analyzing all new dex files and classes loaded through DCL. Then it performs incremental analysis to detect IAC activities continuously during the augmentation process, which results in the final versions of the graphs.

The **vulnerability analysis module** (3) identifies if the nodes in the final graph create a vulnerable path that leaks private data. It runs its analysis on all IAC paths in the graph. Going through the graph, it identifies nodes as senders or receivers and whether they can reach sensitive source or sink methods. Then it marks all paths going from sensitive sources to sensitive sinks across multiple applications[11].

Evaluation: Their evaluation approach used DroidBench and achieved 100% precision and 91.6% recall. The results of the rest of their experiments show successful detection

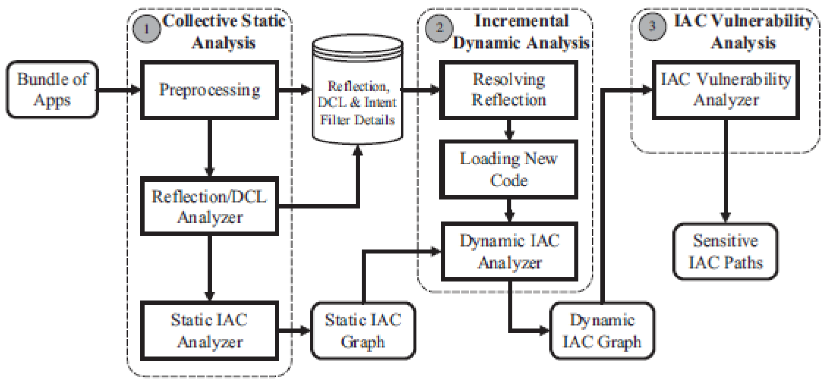


Figure 3.22: The architecture of DINA. (Taken from [11])

of ICC leaks, IAC leaks and reflective (dynamic code loading) calls, which makes it one of the most robust solutions listed here.

Limitations: As with other dynamic leak detection approaches, code coverage is an issue and may lead to false negatives. DINA also does not analyze the full data flows tracking analysis, which leads to imprecision during static analysis, however, the scope of their dynamic analysis alleviates this issue[11].

A potential improvement that the proof-of-concept solution can make is to improve code coverage and reduce false positives.

3.6 Selecting Vulnerabilities for Further Analysis and Solving

The following Table 3.1 shows which vulnerabilities are addressed by which solution. The tools marked with covering "Unknown" vulnerabilities or hiding techniques are marked as such, since it is unknown to what degree they cover them. On the other hand, an approach like Witness[49] detects none of the listed high-level vulnerabilities and covers only 8 specific vulnerabilities listed in CVE[4], however, some of them do fall under the listed high-level vulnerabilities and Witness is thus marked as being able to detect them.

Overall, this section has shown that concrete vulnerabilities that lead to broader issues are rarely analyzed and fixed by third parties. The CVE Details website[4] shows 2563 vulnerabilities, but there is no mention of most of them in academic research. However, the broader categories of vulnerabilities are often studied, and the trends in articles discussing the solutions to these broad issues imply that one solution may cover hundreds of vulnerabilities if developed correctly.

[7]Witness does not cover the full range of ICC vulnerabilities, but only certain ones.

Solution	Vulnerability	Hiding technique
S01 - FlowDroid	V01 - ICC, V03 - Inter-app leaks	None
S02 - IccTA	V01 - ICC	None
S03 - MirrorDroid	V01 - ICC, V02 - Third-party libraries V03 - Inter-app leaks, V04 - Cache file leaks	Unknown
S04 - HybriDroid	V01 - ICC, V03 - Inter-app leaks	H06 - Reflection techniques
S05 - AndroidLeaker	V01 - ICC, V03 - Inter-app leaks	H06 - Reflection techniques
S06 - APPLADroid	V01 - ICC, V03 - Inter-app leaks	Unknown
S07 - X-Decaf	V04 - Cache file leaks	None
S08 - DroidRA	None	H06 - Reflection techniques
S09 - DroidRista	V01 - ICC, V03 - Inter-app leaks	H06 - Reflection techniques
S10 - AppLance	V01 - ICC, V02 - Third-party libraries V03 - Inter-app leaks, V04 - Cache file leaks	H07 - Packing techniques
S11 - Fog Computing Solution	V01 - ICC, V02 - Third-party libraries V03 - Inter-app leaks, V04 - Cache file leaks	H07 - Packing techniques
S12 - Leak detection through API call logs	V01 - ICC, V02 - Third-party libraries V03 - Inter-app leaks, V04 - Cache file leaks	Unknown
S13 - Agrigento	V01 - ICC, V02 - Third-party libraries V03 - Inter-app leaks, V04 - Cache file leaks	Unknown
S14 - Alde	V02 - Third-party analytics libraries	H06 - Reflection techniques
S15 - Witness	V01 - ICC[7]	Unknown
S16 - DINA	V01 - ICC	H06 - Reflection techniques

Table 3.1: Table of vulnerabilities addressed by discussed solutions

3.6.1 Criteria for the Selection of Solutions

To develop a solution for detecting data leaks, a further study of existing solutions and their algorithms is needed. To select which solutions will be further analyzed, a list of criteria was developed.

The main criteria for selecting the solutions for this thesis are regarding the ability to detect leaks without actually running the application, i.e. static detection. The reasoning why static analysis specifically is chosen, as opposed to dynamic analysis is discussed in Section 2.4. This does, however, cause an issue, since any of the vulnerabilities mentioned in Section 3.2 could theoretically be "enhanced" through hiding code in e.g. reflection-based calls and DCL.

The reasoning for the criteria selection is mainly in the limitations section of each solution. Solutions had common limitations, and to further improve on them, these limitations are generalized into selection criteria. For further analysis, a solution which has the best basis for improvement is chosen and the proof-of-concept solution then tries to improve on that limitation.

The criteria for the selection of solutions for the development of the proof-of-concept solution in this thesis are following:

- **SC01** - Employs a static analysis algorithm. The solution itself does not have to be fully static, i.e. it can be a hybrid solution, but it must employ a static analysis algorithm at one point which contributes to overall leak detection. This criteria is defined due to the fact that multiple dynamic analysis solutions had low code coverage and high rate of false negatives.
- **SC02** - Detects at least two data leak vulnerabilities or hiding techniques. The solution has to be versatile and not only detect one type of vulnerability/hiding technique. This criteria is defined because there are multiple solutions which only detect one data leak vulnerability or hiding technique.
- **SC03** - Detects or circumvents hiding techniques mentioned in 3.4. Since many malicious applications try to hide their behavior, it is important that the solution is also able to detect leaks hidden in e.g. reflection or dynamically loaded code. Similar to SC02, solutions would commonly skip hiding techniques, which are often employed by real-world applications to hide behavior.
- **SC04** - Has a well described algorithm in the research article or freely available source code. This criteria is defined because the proof-of-concept solution cannot improve upon a solution which works in an unknown way.

3.6.2 Assessment and Selection of Solutions

Based on the criteria, the solutions selected for analysis are the following:

- **S01 FlowDroid** - FlowDroid fills all criteria and most importantly has open-source code and well described algorithms which can be improved upon.
- **S08 DroidRA** - DroidRA specializes in detection of hiding techniques (reflection), and it can be used as a pre-processing step of any solution.
- **S13 Agrigento** - Although Agrigento is a dynamic analysis solution, it employs some static analysis algorithms which may prove useful for the proof-of-concept solution.
- **S16 DINA** - DINA is another hybrid solution which employs both static and dynamic analysis, however the static analysis algorithm can be used in the proof-of-concept solution.

The following Table 3.2 shows the summary of vulnerability assessment. If a criteria is marked with both ✓and ✗, it means that even though it can be fulfilled, the current solutions do not have an efficient approach to solving it.

Solution	SC01	SC02	SC03	SC04
S01 - FlowDroid	✓	✓	✓	✓
S02 - IccTA	✓	✗	✗	✓
S03 - MirrorDroid	✗	✓	✓	✗
S04 - HybriDroid	✓	✓	✓	✗
S05 - AndroidLeaker	✓	✓	✓	✗
S06 - APPLADroid	✓	✓	✓	✗
S07 - X-Decaf	✗	✗	✗	✗
S08 - DroidRA	✓	✓	✓	✓
S09 - DroidRista	✓	✓	✓	✗
S10 - AppLance	✗	✓	✓	✗
S11 - Fog Computing Solution	✗	✓	✓	✗
S12 - Leak detection through API call logs	✗	✓	✓	✗
S13 - Agrigento	✓	✓	✓	✓
S14 - Alde	✓	✗	✓	✗
S15 - Witness	✓	✗	✗	✗
S16 - DINA	✓	✓	✓	✓

Table 3.2: Table of solution selection criteria and solutions

CHAPTER 4

Research of Static Analysis Algorithms

This chapter gives an overview and an assessment of static analysis algorithms. Since the algorithms can be generic and not specific to Android, a part of this chapter is dedicated to adapting the algorithms to fit the use-case.

4.1 Overview of the State-of-the-Art Static Analysis Algorithms

There are various static analysis algorithms designed for Android using approaches similar to a varying degree. Many of them build and improve on each other, e.g. FlowDroid is an example of an analysis technique that multiple solutions mentioned in Section 3.5 tried to improve. This section gives a general overview of the concrete algorithms used for detecting leaks, and highlights the ones that provided the best results.

The algorithms analyzed here are either extracted from the solutions mentioned in Section 3.6.2, as well as additional standalone algorithms used for static analysis.

Table 4.1 shows which algorithm corresponds to which solution selected in Section 3.5. S08 - DroidRA is not in the table, as this section focuses on algorithms analyzing data flow, while DroidRA focuses on finding specific calls within the code. The two standalone algorithms mentioned in Table 4.1 are also discussed in this section. A01 - IFDS is an algorithm used both in A02 - FlowDroid and A03 - Static control-flow analysis algorithm, while A03 - Static control-flow analysis algorithm is considered an alternative to A02 - FlowDroid and thus selected for further analysis.

[1]Even though IFDS is a standalone algorithm, it is used within other solutions, e.g. FlowDroid

Solution	Algorithm
S01 - FlowDroid	A02 - FlowDroid algorithms
S08 - DroidRA	A06 - DroidRA reflection detection algorithm
S13 - Agrigento	A04 - Agrigento algorithms
S16 - DINA	A05 - DINA algorithms
Standalone algorithms	A01 - IFDS algorithm[1], A03 - Static control-flow analysis algorithm

Table 4.1: Table of solution selection criteria and solutions

4.1.1 A01 - Interprocedural Finite Distributive Subset (IFDS) Algorithm

The original Interprocedural Finite Distributive Subset (IFDS) algorithm was developed by Reps et al.[60], however, the algorithm was not fully shown in the paper. The following Algorithm 4.1 was extracted from the paper by Naeem et al.[54], with Naeem et al. making further extensions to the algorithm in [54].

The Original IFDS

The original IFDS is a dynamic algorithm that deals with the interprocedural, finite, distributive subset problems and calculates a merge-over-all-valid paths solution to the problem. The merge goes over all procedure calls and returns a correctly matched analysis (i.e. context sensitive). The algorithm also requires that the domain data flow of facts is expressed as the powerset of a finite set D, with set union being the merge operator. The data flow should also be distributive over set union.

The algorithm has the summary function approach to context-sensitive analysis, i.e. it computes $P(D) \rightarrow P(D)$ which summarizes the effect of long code sections on any subset of D. The algorithm gets it's efficiency from the compact representations of functions. For example, if we have a subset of D, $S = a, b, c$, the function $f(S)$ can be computed as $f(S) = f() \cup f(a) \cup f(b) \cup f(c)$. The function can be defined through a bipartite graph $\langle D \cup 0, D, E\rangle$, where E is a set of edges from $D \cup 0$ to (a second copy of) D. The graph has an edge from d_1 to d_2 if and only if $d_2 \in f(d1)$. The vertex 0 is an empty set. The function represented by a graph is $f(S) = b : (a, b) \in E \wedge (a = 0 \vee a \in S)$. By combining two graphs, merging the nodes of the range of g with the correct nodes of the domain of f, and computing the reachability from the nodes of g to the nodes of f, composition of two graphs is achieved. A relational product of the sets of edges representing two functions gives a set of edges representing their composition.

The input for the algorithm is an exploded supergraph, which represents the program being analyzed and the data flow functions. The supergraph consists of the interprocedural control flow graph (ICFG) of the program where each instruction is replaced by a graph representation of the flow function. The vertices of the supergraph are pairs $\langle l, d\rangle$, with l being a label in the program and $d \in D \cup 0$. The edge $\langle l, d\rangle \rightarrow \langle l', d'\rangle$ is found in the supergraph if the ICFG has an edge $l \rightarrow l'$ and $d' \in f(d)$ (or $d' \in f()$ when $d = 0$), with f is the flow function at label l. The supergraph contains a set of edges representing the flow function of every interprocedural call or return. The flow function on the call edge maps facts about the caller actuals to facts about callee formals. For every valid path from $\langle s, 0 \rangle$ to $\langle l, d \rangle$ in the supergraph the solution contains the elements d of D. The data flow analysis reduces to valid-path reachability on the supergraph.

The algorithm incrementally constructs two tables, *PathEdge* and *SummaryEdge*, which represent the flow functions of long sequences of code. *PathEdge* is a table of triples $\langle d, l, d' \rangle$ (Sometimes written as $\langle s_p, d \rangle \rightarrow \langle l, d' \rangle$ for clarity), which show a path $\langle s_p, d \rangle$ to $\langle l, d' \rangle$, with s_p is the start node of a procedure. *SummaryEdge* is a table containing triples $\langle c, d, d' \rangle$(Sometimes written as $\langle c, d \rangle \rightarrow \langle r, d' \rangle$, with r being the instruction following c), with c indicating a call site. The triple indicates $d' \in f(d)$ with f being a flow function which summarizes the effect of the procedure called from a call site. The representation of the algorithm assumes that in the ICFG, every call site has a no-op "return site" node r as a successor.

The two tables are interdependent. If we have an edge from a start node to the exit node added to *PathEdge*, for every call site of the procedure, a corresponding triple, or several of them due to composition, must be added to *SummaryEdge*, as the effect of the new call site. This composition is computed by combining the graphs representing f_c and f_r from the supergraph with the new edge. That is, for each d_4 and d_5 such that $\langle d_4, d_1 \rangle \in f_c$ and $\langle d_2, d_5 \rangle \in f_r$, the triple $\langle c, d_4, d_5 \rangle$ is added to SummaryEdge. This is performed in lines 29 to 31 of the Algorithm 4.1.

If we have a triple added to *SummaryEdge* with a new effect of the call at c, for each data flow from the start node to the call site, there is a valid path from s to r, where r is the successor of c, thus the edge from the start node to the successor of the calls site must be added to *PathEdge*. This is performed in lines 32 to 34 of the Algorithm 4.1.

IFDS Extensions

The extended IFDS algorithm with all four extensions made by Naeem et al.[54] is shown in Algorithm 4.2. The altered or added lines are underlined.

- The first extension only constructs the nodes in the supergraph on demand, to limit the size of it. The size of the graph needs to be limited in order to keep computational complexity of the analysis low so that it can still complete even on large applications.
- The second extension gives a procedure-return flow function more information about the program state prior to a procedure call.
- The third extension improves the precision with which the instructions are modeled by analyzing the program in Static Single Assignment (SSA) form. The SSA form describes every definition of a variable as a new variable (i.e. it creates a versioned

Algorithm 4.1: The extracted original IFDS algorithm (taken from [54])

```
declare PathEdge, WorkList, SummaryEdge: global edge set
Algorithm Tabulate($G^{\#}_{IP}$)
    Let $(N^{\#}, E^{\#}) = G^{\#}_{IP}$
    PathEdge := $\langle s_{main}, 0\rangle \rightarrow \langle s_{main}, 0\rangle$
    WorkList := $\langle s_{main}, 0\rangle \rightarrow \langle s_{main}, 0\rangle$
    SummaryEdge := $\emptyset$
    ForwardTabulateSLRPs()
    foreach $n \in N^{\#}$ do
        $X_n :=\ d2 \in D | \exists d1 \in (D \cup 0) s.t. \langle s_{procOf(n)}, d1\rangle \rightarrow \langle n, d2\rangle \in$ PathEdge
    end
Procedure Propagate($e$)
    if $e \notin$ PathEdge then
        Insert $e$ into PathEdge; Insert $e$ into WorkList
    end
Procedure ForwardTabulateSLRPs()
    while $WorkList \neq \emptyset$ do
        Select and remove an edge $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ from WorkList
        switch $n$ do
            case $n \in Call_p$: do
                foreach $d_3 s.t. \langle n, d_2\rangle \rightarrow \langle s_{calledProc(n)}, d_3\rangle \in E^{\#}$ do
                    Propagate($\langle s_{calledProc(n)}, d_3\rangle \rightarrow \langle s_{calledProc(n)}, d_3\rangle$)
                end
                foreach $d_3 s.t. \langle n, d_2\rangle \rightarrow \langle$ returnSite($n$)
                 $, d_3\rangle \in (E^{\#} \cup$SummaryEdge ) do
                    Propagate($\langle s_p, d_1\rangle \rightarrow \langle$ returnSite($n$) $, d_3\rangle$)
                end
            end
            case $n \in e_p$ : do
                foreach $c \in$ callers($p$) do
                    foreach $d_4, d_5 s.t. \langle c, d_4\rangle \rightarrow \langle s_p, d_1\rangle \in E^{\#} and \langle e_p, d_2\rangle \rightarrow \langle$
                     returnSite($c$) $, d_5\rangle \in E^{\#}$ do
                        if $\langle c, d_4\rangle \rightarrow \langle$ returnSite($c$) $, d_5\rangle \notin$ SummaryEdge then
                            Insert $\langle c, d_4\rangle \rightarrow \langle$ returnSite($c$) $, d_5\rangle$ into
                             SummaryEdge
                            foreach $d_3 s.t. \langle s_{procOf(c)}, d_3\rangle \rightarrow \langle c, d_4\rangle \in$ PathEdge do
                                Propagate($\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle$ returnSite($c$)
                                 $, d_5\rangle$)
                            end
                        end
                    end
                end
            end
            case $n \in N_p - Call_p - e_p$: do
                foreach $\langle m, d_3\rangle s.t. \langle n, d_2\rangle \rightarrow \langle m, d_3\rangle \in E^{\#}$ do
                    Propagate($\langle s_p, d_1\rangle \rightarrow \langle m, d_3\rangle$)
                end
            end
        end
    end
```

Algorithm 4.2: The extended IFDS algorithm, with the changes underlined (taken from [54])

```
declare PathEdge, WorkList, SummaryEdge, $\underline{\text{Incoming, EndSummary}}$: global
  edge set
Algorithm Tabulate ($\underline{flow,\ passArgs,\ returnVal,\ callFlow}$)
    ...
Procedure ForwardTabulateSLRPs()
    while $WorkList \neq \emptyset$ do
        Select and remove an edge $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ from WorkList
        switch $n$ do
            case $n \in Call_p$: do
                foreach $\underline{d_3 \in passArgs(\langle n, d_2 + \rangle)}$ do
                    Propagate($\langle s_{calledProc(n)}, d_3\rangle \rightarrow \langle s_{calledProc(n)}, d_3\rangle$)
                    $\underline{\text{Incoming } [\langle s_{calledProc(n)}, d_3\rangle] \cup = \langle n, d_2\rangle}$
                    foreach $\underline{\langle e_p, d_4\rangle \in \text{EndSummary } [\langle s_{calledProc(n)}, d_3\rangle]}$ do
                        foreach $\underline{d_5 \in \text{returnVal}(\langle e_p, d_4\rangle, \langle n, d_2\rangle)}$ do
                            $\underline{\text{Insert } \langle n, d_2\rangle \rightarrow \langle \text{returnSite}(n), d_5\rangle \text{ into SummaryEdge}}$
                        end
                    end
                end
                foreach $d_3 s.t.$ $\underline{d_3 \in callFlow(\langle n, d_2\rangle)\ or}$ $\langle n, d_2\rangle \rightarrow \langle$
                  returnSite(n) $, d_3\rangle \in (E^{\#} \cup \text{SummaryEdge})$ do
                    Propagate($\langle s_p, d_1\rangle \rightarrow \langle$ returnSite(n) $, d_3\rangle$)
                end
            end
            case $n \in e_p$ : do
                $\underline{\text{EndSummary } [\langle s_p, d_1\rangle] \cup = \langle e_p, d_2\rangle}$
                foreach $\underline{\langle c, d_4\rangle \in \text{Incoming } [\langle s_p, d_1\rangle]}$ do
                    foreach $\underline{d_5 \in \text{returnVal}(\langle e_p, d_2\rangle, \langle c, d_4\rangle)}$ do
                        if $\langle c, d_4\rangle \rightarrow \langle$ returnSite(c) $, d_5\rangle \notin$ SummaryEdge then
                            Insert $\langle c, d_4\rangle \rightarrow \langle$ returnSite(c) $, d_5\rangle$ into
                              SummaryEdge
                            foreach $d_3 s.t. \langle s_{procOf(c)}, d_3\rangle \rightarrow \langle c, d_4\rangle \in$ PathEdge do
                                Propagate($\langle s_{procOf(c)}, d_3\rangle \rightarrow \langle$ returnSite(c)
                                  $, d_5\rangle$)
                            end
                        end
                    end
                end
            end
            case $n \in N_p - Call_p - e_p$: do
                foreach $\underline{\langle m, d_3\rangle s.t.\ n \rightarrow m \in CFG\ and\ d_3 \in flow(\langle n, d_2\rangle, \pi)}$ do
                    Propagate($\langle s_p, d_1\rangle \rightarrow \langle m, d_3\rangle$)
                end
            end
        end
    end
```

variable). This way a variable is only assigned once and it has to be declared before use. This simplifies variable properties so that various compiler optimization algorithms, such as dead code removal, can be enabled[61].

- The fourth extension improves efficiency on domains where some data flow facts subsume each other.

4.1.2 A02 - FlowDroid's Analysis Algorithms

As mentioned in Section 3.5, FlowDroid[15][26] is a static analysis solution, which many solutions mentioned in Section 3.5 used as a basis. FlowDroid uses forward-taint and on-demand backward-alias analysis techniques. It models its taint-analysis within the IFDS framework. The A01 - IFDS algorithm used is the extended algorithm from Section 4.1.1.

The forward and backward solvers are shown in Algorithms 4.3 and 4.4 respectively.

Forward Taint Analysis. The forward and backward analysis return access paths. An access path is a triple of a local variable and two fields. The access path describes the set of all objects reachable through this path.

If any of the right-hand side operands are tainted, the left-hand side is tainted as well through the transfer function for assigning. Assignments to arrays taint the whole array. Any taints rooted at a variable get erased when a "new" assignment is made. Method calls use formal parameters to translate access paths to callee's context, while the inverse happens on method returns. FlowDroid also has a call-to-return flow function, which bypasses the method calls on the caller's side and thus propagates taints irrelevant for the call, at sources generates new taints, at sinks reports taints and propagates native call taints.

On-demand Backwards-Alias Analysis. When a tainted value gets assigned to the heap, FlowDroid runs a backwards analysis and searches for aliases of the target variable and taints them. This spawns a forward analysis and potentially finds a new leak.

Maintaining context sensitivity. Algorithms 4.3 and 4.4 show the forward and backward analyzers. This representation assumes that the reader knows how the IFDS algorithm works. The IFDS algorithm is explained in Section 4.1.1. Each solver has their own worklist, a set of path-edges which summarize the flows that were computed up until the current node.

An edge $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$ means that the analysis states that d_2 holds at n if d_1 holds at the start point s_p of procedure p. In this case, the values d_i are access paths which describe the references to the tainted values. To achieve context sensitivity, the handover between the two analyses has to be done precisely, so that the result does not produce conflicting contexts and unrealizable paths. FlowDroid avoids false positives by injecting context from one analysis to the other.

Algorithm 4.3: FlowDroid main loop of forward solver (taken from [15])

```
1  while $WorkList_{FW} \neq \emptyset$ do
2      $pop\langle s_p, d_1\rangle \to \langle n, d_2\rangle\ off\ WorkList_{FW}$;
3      switch $n$ do
4          case $n$ is call statement: do
5              if summary exists for call then
6                  apply summary
7              end
8              else
9                  map actual parameters to formal parameters
10             end
11         end
12         case $n$ is exit statement: do
13             install summary $\langle s_p, d_1\rangle \to \langle n, d_2\rangle$
14             map formal parameters to actual parameters
15             map return value back to caller's context
16         end
17         case $n$ is assignment $lgs = rhs$: do
18             $d_3 :=$ replace $rhs$ by $lhs$ in $d_2$
19             insert $\langle s_p, d_1\rangle \to \langle n, d_3\rangle$ into $WorkList_{BW}$
20         end
21     end
22     extend path-edges via the propagate-method of the classical IFDS algorithm
23 end
```

When processing the assignment of a variable, the forward analysis spawns a backwards alias analysis. FlowDroid spawns the analysis by injecting the context of the forward analysis into the backwards analysis. FlowDroid consults the "path edge", which the IFDS algorithm stores as a summary computation side-effect and FlowDroid injects the entire edge into the backward solver (Algorithm 4.3 line 19). The same injection happens the other way around (Algorithm 4.4 line 19).

To avoid unrealizable paths, the returning into contexts which were not analyzed by the forward analysis needs to be prevented. The backward analysis never returns into the caller, instead, whenever it searches for an alias, it triggers the forward analysis. The forward analysis then maps the relevant taints to the caller's context. All returns are handled by the forward analysis and the backwards analysis can go into callees but not to callers. Algorithm 4.4 line 14 shows that the backwards analysis spawns a forward analysis when it reaches a method header.

Maintaining flow sensitivity. To maintain flow sensitivity, FlowDroid keeps track of all activation statements. When a backward analysis is spawned, the access path is

Algorithm 4.4: FlowDroid main loop of backward solver (taken from [15])

```
while $WorkList_{BW} \neq \emptyset$ do
    $pop\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ $off$ $WorkList_{BW}$;
    switch $n$ do
        case $n$ is call statement: do
            if summary exists for call then
                apply summary
            else
                map actual parameters to formal parameters
            end
            extend path-edges via the propagate-method of the classical IFDS
              algorithm
        end
        case $n$ is method's first statement: do
            install summary $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$
            insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_2\rangle$ into $WorkList_{FW}$
            do not extend path-edges via the propagate-method of the classical
              IFDS algorithm, killing current taint $d_2$
        end
        case $n$ is assignment $lgs = rhs$: do
            $d_3$ := replace $rhs$ by $lhs$ in $d_2$
            insert $\langle s_p, d_1\rangle \rightarrow \langle n, d_3\rangle$ into $WorkList_{FW}$
            extend path-edges via the propagate-method of the classical IFDS
              algorithm
        end
    end
end
```

augmented with the alias' activation statement and the tainted alias is marked inactive. Inactive taints are aliases to yet-to-be-tainted memory locations. Only active taints can cause leaks on the sink. When a forward analysis is spawned and an aliased taint is propagated over its activation statement, the taint gets activated and can cause a leak. The activation statements represent call trees, i.e. they are used for looking up call trees where they occur in order to translate them back into transitive callers.

The extended IFDS which FlowDroid uses, computes the application super-graph on the fly. This means that the only taint information being computed is for the variables that are actually tainted. The two algorithms of FlowDroid (i.e. Algorithm 4.3 and 4.4) are actually two instances of the IFDS solver with slight adjustments. Each algorithm has its own summary functions table, which is used to avoid re-computing for the same callees in the same context.

4.1.3 A03 - Static Control-Flow Analysis Algorithm

The algorithm developed by Yang et al.[73] analyzes the user-event-driven components and the sequences of callbacks (i.e. lifecycle and event handler callbacks) from the Android framework to the application code. The algorithm was not designed with the main goal of detecting leaks, but as a general data-flow analysis algorithm which can be used to see how data flows from start to end of the program. However, it can be used as a basis for any application analysis.

Control-Flow Analysis of a Callback Method

A key component of this approach is the context-sensitive callback analysis. The method uses static abstractions for windows and views. For an event handler, the context is a view v, while for a lifecycle callback, the context is a window w. The concrete analysis is shown in Algorithm 4.5. The algorithm is used by the main control-flow analysis, describes in Section **CCFG Construction**.

Input and output. The algorithm goes through all valid ICFG paths, starting with the entry node of a method's CFG and ending when a trigger node is reached. A trigger node is one that may execute a new callback. The set of all possible trigger nodes is an input for the algorithm. One output of the algorithm is the set of all trigger nodes which were reached during the traversal.

The algorithm also determines whether the exit node of a CFG is reachable from the entry node through a valid, trigger-free path. The execution of a method can then avoid executing triggers. Another output of the algorithm is the boolean value which marks if a trigger-free path exists.

Context sensitivity. A pre-analysis call to *ComputeFeasibleEdges* which determines the feasible ICFG edges from a method and the transitively called methods is used to achieve context-sensitivity. Only feasible edges are followed in the trafersal (lines 5–25 in Algorithm 4.5).

The feasibility pre-analysis choice is dependant on the callback method and the context. The pre-analysis is further shown in Section 20. The analysis can identify virtual calls where a window is the only possible receiver for any lifecycle callback in its context, and thus determine the feasible edges.

Algorithm design. The algorithm is based on the IFDS algorithm, similarly to FlowDroid. The problem is formulated as a lattice with two elements: $\emptyset$ and a singleton set $entryNode(m)$. The data-flow functions are $\lambda x.x$ for non-trigger nodes (identity function) and $\lambda x.\emptyset$, for trigger nodes.

The set *avoidingMethods* is a set of all methods that contain a trigger-free valid path from the entry to the exit node. This means that a same-level valid path has a call site which has a matching return site and the execution of these methods avoids triggers. If a call-site node is reachable and it invokes an *avoidingMethods* method, the return-site

Algorithm 4.5: AnalyzeCallbackMethod(m,c) of the static control-flow analysis (taken from [73])

```
input  : $m$ : callback method
input  : $c$ : context
input  : $triggerNodes$ : set of ICFG nodes
output: $reachedTriggers \leftarrow \emptyset$ : set of ICFG nodes
output: $avoidsTriggers$ : boolean

$feasibleEdges \leftarrow$ ComputeFeasibleEdges($m, c$)
$visitedNodes \leftarrow$ entryNode($m$)
$nodeWorklist \leftarrow$ entryNode($m$)
$avoidingMethods \leftarrow \emptyset$
while $nodeWorklist \neq \emptyset$ do
    $n \leftarrow$ removeElement($nodeWorklist$)
    if $n \in triggerNodes$ then
        $reachedTriggers \leftarrow reachedTriggers \cup n$
    else if $n$ is not a call-site node and not an exit node then
        foreach $ICFG\ edge\ n \rightarrow k \in feasibleEdges$ do
            Propagate($k$)
        end
    else if $n$ is a call-site node and $n \rightarrow entryNode(p) \in feasibleEdges$ then
        Propagate(entryNode($p$))
        if $p \in avoidingMethods$ then
            Propagate(returnSite($n$))
        end
    else if $n$ is exitNode($p$) and $p \notin avoidingMethods$ then
        $avoidingMethods \leftarrow avoidingMethods \cup p$
        foreach $c \rightarrow$ entryNode($p$) $\in feasibleEdges$ do
            if $c \in visitedNodes$ then
                Propagate(returnSite($c$))
            end
        end
    end
end
$avoidsTriggers \leftarrow m \in avoidingMethods$
Procedure Propagate($k$)
    if $k \notin visitedNodes$ then
        $visitedNodes \leftarrow visitedNodes \cup k$
        $nodeWorklist \leftarrow nodeWorklist \cup k$
    end
```

node is also reachable (lines 15-16). Whenever an exit node is reached for the first time (line 18), the method is added to *avoidingMethods*, and all call sites which invoke it are considered for possible reachability (lines 20-24). The set *avoidingMethods* is a variation of the IFDS *SummaryEdges*.

CCFG Construction

The Callback Control-Flow Graph (CCFG) construction uses the output of Gator[5], an Android analysis toolkit. In the output, an activity is associated with widgets (i.e. views) or an options menu. Each menu is a separate window with widgets, which typically contain items in a list. Dialogs are separate windows with related choices and they are associated with their own widgets. A widget can be associated with multiple event handlers.

The CCFG construction creates nodes for the relevant callbacks of each window. The lifecycle methods for creation and termination of windows are based on standard API. Further description assumes that a window defines both creation and termination callbacks, while the implementation does not make this assumption. For any handler of a widget, there is a CCFG node (h,v). A branch node b_w and a join node j_w are also introduced.

Edge creation. Algorithm 4.6 defines the edges of a window w.

Edges $(l^c, w) \rightarrow b_w \rightarrow (l^t, w)$ show the lifetime callback invocations (line 2). The second edge represents the window termination through the *back* button. The window can also be terminated by event handlers. If a termination trigger node is reached an edge is created (line 11). If the termination trigger can be avoided through an ICFG path, a new edge is created (line 13).

For each handler of a view from the window widget set, an edge $b_w \rightarrow (h, v)$ is created to show possible user actions and invoked handlers (line 9). The back edge created at line 17 creates a structure which has arbitrary ordering of user-triggered events. If the window is a menu, the item selection closes and an edge to the termination callback is created.

Every handler is analyzed under the context of the view using Algorithm 4.5 (called in line 10 of Algorithm 4.6). If *avoidsTriggers* is true, an edge is added to show that the handler retains the current window and that user events will continue to trigger handlers of the same window. Other outgoing edges for the handler node are determined by *reachedTriggers* and are created by *TriggerEdges*.

Algorithm 4.5 is invoked for the creation callback (line 2) to determine which triggers are reachable. The edge creation for the creation callback (lines 3-6) is similar to the one for handlers (11-14). Termination callbacks have no such triggers. Edges created by *TriggerEdges* are based on the analysis of trigger statements. Activity-launch calls are analyzed with a flow- and context-insensitive intent analysis, accounting for statement feasibility. The analysis is focused on explicit intents. Menu and dialog launch calls are resolved by Gator. All of these statements launch new windows. Correspondingly, function *TriggerEdges* produces edges from an event handler node to a creation callback, and from a termination callback to a join node when invoked at line 11 and edges from a creation node of one window to a creation node of a new window, as well as a edge from a termination callback to a branch node when invoked at line 3.

TriggerEdges also checks for the possibility that triggers contain a statement which terminates the current window. If the set contains a termination statement, an edge from the handler or a creation callback node to the termination statement is created to represent possible control-flow

Algorithm 4.6: CreateEdges(w) of the static control-flow analysis (taken from [73])

```
    input  : $w$ : window
    input  : $(l^c, w), (l^t, w)$ : lifecycle nodes for $w$
    input  : $(h_1, v_1), (h_2, v_2), \dots$ : event handler nodes for $w$
    input  : $b_w, j_w$ : branch/join nodes for $w$
    output: $newEdges$ : set of CCFG edges for $w$

 1  $newEdges \leftarrow \emptyset$
 2  $\langle triggers, avoids \rangle \leftarrow$ AnalyzeCallbackMethod($l^c, w$)
 3  $newEdges \leftarrow newEdges \cup$ TriggerEdges($triggers, l^c, w$)
 4  if $avoids$ then
 5  |   $newEdges \leftarrow newEdges \cup (l^c, w) \to b_w$
 6  end
 7  $newEdges \leftarrow newEdges \cup b_w \to (l^t, w)$
 8  foreach $event\ handler\ node\ (h, v)$ do
 9  |   $newEdges \leftarrow newEdges \cup b_w \to (h, v)$
10  |   $\langle triggers, avoids \rangle \leftarrow$ AnalyzeCallbackMethod($h, v$)
11  |   $newEdges \leftarrow newEdges \cup$ TriggerEdges($triggers, h, v$)
12  |   if $avoids$ then
13  |   |   $newEdges \leftarrow newEdges \cup (h, v) \to j_w$
14  |   end
15  end
16  if $w\ is\ not\ a\ menu$ then
17  |   $newEdges \leftarrow newEdges \cup j_w \to b_w$
18  else
19  |   $newEdges \leftarrow newEdges \cup j_w \to |(l^t, w)$
20  end
```

Detection of Feasible Edges

Constant propagation formulated as Interprocedural Distributed Environment (IDE) analysis problems is used to detect feasible edges within a context.

In the *ComputeFeasibleEdges* analysis of a callback method under a context (widget v, or window w), first, the analysis uses interprocedural constant propagation to find out which local variables refer to only one object. This determines that a particular parameter of a method definitely refers to the given context and it obtains additional reference information. The analysis also considers all methods transitively invoked by the given method. Virtual calls are solved through hierarchy information. After constant propagation, the new information refines virtual call resolution, i.e. if only one receiver is determined to be possible, the call is resolved. A second interprocedural constant propagation analysis determines the constant values of integers and booleans. The final step uses branch nodes whose conditions are constants to determine infeasible ICFG edges, which together with infeasible edges of refined virtual calls define the output of *ComputeFeasibleEdges*.

Valid CCFG Paths

Not every path created in CCFG simulates a real sequence of callback methods. Overall, a valid CCFG path has matching edges $\cdots \to (lc, w)$ and $(lt, w) \to \ldots$, which are created by *TriggerEdges* and recorded as a matching pair. This condition and the static analysis implications are similar to traditional ICFG control-flow analysis. One can focus only on valid CCFG paths with standard techniques, such as explicitly maintaining the sequence of unmatched edges $(\cdots \to (lc, w))$ or creating approximations of them.

4.1.4 A04 - Agrigento's Algorithms

Agrigento's[24] approach, as explained in Section 3.5.13, runs a baseline of the application, and then it modifies the sources of private information and reruns the application. After running it, it compares the network behavior summaries to identify the changes and potential leaks.

The base of this analysis is the differential analysis algorithm shown in Algorithm 4.7. For each HTTP flow in the contextualized trace in the final run, Agrigento checks the tree from the base runs to see if the call is a part of it. If it does not find a match, it searches for the most similar HTTP flow by comparing the position in the tree (i.e. same domain, key). It also recognizes patterns of known data structures (e.g. JSON). If it finds any, it parses them and compares the subfields. Before comparing subfields, it also decodes known encodings (e.g. Base64). To obtain the alignment of the fields under comparison, Agrigento uses the Needleman-Wunsch algorithm. This identifies the similarity regions between fields and inserts gaps so that the same characters are at the same positions. This produces a regular expression (i.e. regex) where consecutive gaps are marked with a wildcard (i.e. *). To obtain the differences between calls, Agrigento extracts the substrings that match the wildcards of a regex. It discards any differences caused by previous network request and whitelists known benign differences caused by known libraries (e.g. Google Ads).

Algorithm 4.7: Agrigento differential analysis (taken from [24])

```
Procedure DifferentialAnalysis($context-trace$, $summary$)
    $diffs \leftarrow \emptyset$
    for $http-flow \in context-trace$ do
        if $http-flow \notin summary$ then
            $field \leftarrow$ getMissingField($http-flow, summary$)
            $fields \leftarrow$ getSamePositionField($field, summary$)
            $diffs.add($Compare($field, fields$)$)$
        end
    end
    return $diffs$
Procedure Compare($field$, $fields$)
    $diffs \leftarrow \emptyset$
    $most-similar \leftarrow$ mostSimilar($field, fields$)
    if $isKnownDataStructure(field, most-similar)$ then
        $subfields \leftarrow$ parseDataStructure($field$)
        $similar-subfields \leftarrow$ parseDataStructure($most-similar$)
        for $i \in subfields$ do
            $diffs.add($Compare($subfields_i, similar-subfields_i$)$)$
        end
        return $diffs$
    end
    if $isKnownEncoding(field, most-similar)$ then
        $field \leftarrow$ decode($field$)
        $most-similar \leftarrow$ decode($most-similar$)
    end
    $alignment \leftarrow$ align($field, most-similar$)
    $regex \leftarrow$ getRegex($alignment$)
    $diffs \leftarrow$ getRegexMatches($field$)
    $diffs \leftarrow$ removeNetworkValues($diffs$)
    $diffs \leftarrow$ whitelistBenignLibaries($diffs$)
    return $diffs$
```

4.1.5 A05 - DINA's Algorithms

The algorithms which are used in DINA are explained in more detail here. Overall, DINA consists of three algorithms, a static analysis, dynamic analysis and a vulnerability analysis. The static and vulnerability analysis are executed without the application running, and will be considered for this thesis, as even though they are not enough on their own, they can be used as a basis for expansion.

Collective Static Analysis

Algorithm 4.8 shows the static analysis process:

Preprocessing. The APKs of each application are decompiled and the manifest files are extacted, along with the intent filter information from the manifest file (lines 3-7). The MCG (line 9) and IG (line 14) are also generated.

Algorithm 4.8: DINA Collective Static Analysis (taken from [11])

```
   input  : Bundle of Apps: $B, Ref\_DCL\_API\_List$
   output: $static\_IAC, Intent\_Filter\_App_i, Ref\_Details$
   /* Preprocessing                                                     */
 1 $static\_IAC \leftarrow$ CreateNodes($|B|$)
   /* initialize Intent filter list                                     */
 2 $Intent\_Filter\_App_i \leftarrow \{\}$
 3 foreach $App_i \in B$ do
 4     Decompile($App_i$)
 5     parse_manifest($App_i$)
 6     update ($Intent\_Filter\_App_i$) $\leftarrow \{(App_i$, class-name, intent-action-string$)\}$
 7 end
 8 foreach $App_i \in B$ do
 9     Generate $MCG(App_i)$
       /* Reflection analyzer                                           */
10     foreach $method \in MCG(App_i)$ do
11         if $method_j \in Ref\_DCL\_API\_List$ then
12             update($Ref\_Details$) $\leftarrow \{(App_i$, class-name, method-name$)\}$
13         end
14         Generate $IG(method_j)$
           /* Static IAC Analyzer                                       */
15         foreach $instruction \in IG(method_j)$ do
16             foreach $String\ str\ in\ instruction_k$ do
17                 if $str \in Intent\_Filter\_App_r.intent - action - string$ then
18                     Update $static\_IAC \leftarrow$ addEdge ($App_i, App_r$)
19                 end
20             end
21         end
22     end
23 end
```

Reflection/DCL analyzer. DCL and reflective calls are detected using APIs and the class and method names are extracted as well (lines 10-13). The list of APIs is used as input of the algorithm. This also determines which applications need to be analyzed dynamically.

Static IAC analyzer. This analyzer identifies IAC paths through string matching of intent action strings of multiple applications. The static IAC graph is then augmented with edges between components of multiple applications (lines 17-19).

IAC Vulnerability Analysis

Algorithm 4.9: DINA IAC Vulnerability Analysis (taken from [11])

```
    input  : $dynamic\_IAC, Sensitive\_API\_List$
    output: $node_i.sensitive$
1  foreach $node$ of $App_m \in dynamic\_IAC$ do
       /* Identify sensitive methods in the sender node      */
2      if $node_i$ is sender then
3          foreach $method \in DFS(node_i.method - name)$ do
4              if $method_j \in Sensitive\_API\_List$ then
5                  $node_i.sensitive$ = True
6              else
7                  $node_i.sensitive$ = False
8              end
9          end
       /* Identify sensitive methods in the receiver node    */
10     else if $node_i$ is receiver then
11         foreach $method \in MG(App_m)$ do
12             if $method_j \in$ {onCreate, onReceive, onStartCommand } &&
                 ($class$-$name$ of $method_j$ == $class$-$name$ of $node_i$) then
13                 foreach $method \in DFS(method_j)$ do
14                     if $method_j \in Sensitive\_API\_List$ then
15                         $node_i.sensitive$ = True
16                     else
17                         $node_i.sensitive$ = False
18                     end
19                 end
20             end
21         end
22     end
23 end
```

Algorithm 4.9 shows the vulnerability analysis. It identifies whether the nodes in the generated graph contain a vulnerability which leaks private information. The analyzer runs on all IAC paths in the graph. For each path, every one is analyzed whether it is a receiver or a sender. If the node is a sender, an inverted DFS is conducted to determine whether it can reach a sensitive source node (lines 2-9). If the node is a receiver, a DFS search is conducted from the Intent-receiving method to determine whether it can reach a sensitive sink node (lines 10-22). To determine which sources and sinks are sensitive, an API list (i.e. *Sensitive_API_List*) is used.

The complete sensitive paths from the source to the sink are marked across multiple applications.

4.1.6 A06 - DroidRA Reflection Detection Algorithm

DroidRA's[47] algorithm is not a data flow or a data leak analysis algorithm, but it is a static analysis algorithm. There is no formal definition of the algorithm provided in [47], but the following steps can be deduced through the source code analysis:

1. Load all decompiled classes and methods.
2. For every statement in every method, check if it is a reflective call.
3. If it is a reflective call, save the metadata of the method call for later use.

4.2 Assessment and Comparison of the Static Analysis Algorithms

This section will go over the previously discussed algorithms and compare them, as well as determine which ones are used in the proof-of-concept solution.

4.2.1 Assessment Criteria for Selecting a Static Analysis Algorithm

The main criteria for algorithm selection are similar to criteria for Solutions in Section 3.6.1:

- **AC01** - The algorithm is usable in a data leak detection solution on its own (i.e. without supporting algorithms, like a dynamic data leak detection algorithm).
- **AC02** - The algorithm has a way of determining which type of data is being leaked (e.g. through tainting).
- **AC03** - The algorithm is tested and effective in detecting data leaks or analyzing data flows.

An algorithm does not need to fulfill all the mentioned criteria if it is able to be combined with other similar algorithms which fulfill the missing criteria.

Hybrid leak detection solutions also contain static algorithm, which sometimes do not provide any definitive output (e.g. an application has a leak), but they may contain the base elements for a fully static analysis and are thus considered as well, but only if the essential parts of the algorithm do not depend on the dynamic part (AC01).

To determine whether the data leaked contains private information, the algorithm also needs to be able to contain information about the data being leaked, such as it's source or content (A02).

For AC03, only the algorithms that have well documented evaluation, since taking an algorithm that claims to detect data leaks, and in practice performs poorly moves research in the wrong direction.

4.2.2 Selecting Static Analysis Algorithms

Out of the five algorithms mentioned in Section 4.1, one is IFDS[60], and two (FlowDroid[15], Static control-flow analysis algorithm[73]) are based on it. Multiple solutions mentioned in Section 3.5 are also based on FlowDroid, which makes the influence of IFDS widely spread.

FlowDroid's algorithm has the reputation and the proven results on Android and is thus, along with the extended IFDS algorithm, chosen as the basis for the solution. FlowDroid is also an example of an algorithm which is not standalone, since it required IFDS, but since IFDS is also a static algorithm, it passes AC01.

While Algorithm A03 is also IFDS based, the main reason why it was not selected is that it was not tested how well it performs regarding data leak detection and thus failed AC03.

Algorithms from A04 - Agrigento and A05 - DINA both rely on their dynamic algorithm counterparts for full utilization so they fail AC01.

Since the criteria for the algorithm selection are focusing on data flow and data leak analysis, the A06 - reflection detection algorithm from DroidRA is excluded from the criteria and it is treated as a special case. It is selected for the proof-of-concept solution due to the fact that it can be used as a step to enhance the data leak capabilities of another algorithm, like the FlowDroid algorithm.

The following Table 4.2 summarizes the selection criteria of the algorithms.

Algorithm	AC01	AC02	AC03
A01 - IFDS	✓	✓	✓
A02 - FlowDroid	✓	✓	✓
A03 - Static control-flow algorithm	✓	✓	✗
A04 - Agrigento	✗	✓	✓
A05 - DINA	✗	✓	✓
A06 - DroidRA algorithm	N/A	N/A	N/A

Table 4.2: Table of algorithm selection criteria and algorithms

4.3 Static Analysis Algorithm Configuration

Since IFDS was developed in 1995 and extended in 2010[54], programming has changed. However, the base ideas behind procedure/method calls are the same and thus they may be used mostly unmodified, with Android-specific modifications and pre-processing steps which allow analysis. IFDS and the algorithms from FlowDroid are used with standard out-of-the-box configuration.

CHAPTER 5

Proof-of-Concept Solution for Private Data Leak Detection

The solutions presented in Section 3.5 showcased different approaches to the problem of data leaks on Android. The idea of this thesis is to overview existing solutions, and try to improve upon their issues. The focus is also on showing why these limitations exist. FlowDroid[15] has been in active development since the original paper came out, implementing various improvements and fixes for the original tool.

Of all the solutions presented, FlowDroid was the solution which was the most approachable, since it is open-source and well documented. The thesis identified one limitation in it that can be improved upon, and that is the lack of dynamic code analysis. One of the other solutions covered in this thesis, DroidRA[47] tackles this exact issue. It tries to improve the capabilities of static analysis to detect leaks in dynamically loaded code.

The proof-of-concept solution presented in this thesis tries to combine DroidRA and FlowDroid to improve FlowDroid's data leak detection capabilities. One thing to note is however, that FlowDroid has been in active development since the paper released and it is not fully known what the current limitations of that solution are, since it has not been documented in a follow-up article. Since FlowDroid mentions lack of analysis of reflective calls[15], and this issue has not been tackled academically yet, the proof-of-concept solution in this thesis tries to improve upon this limitation. Since DroidRA's main goal is to take reflective calls and transform them into direct calls, their APK modification should improve FlowDroid's reflection analysis results.

Comparison to existing tools The approach by this thesis is similar to the one made by Alzaidi et al. in DroidRista[13], since it combines DroidRA and FlowDroid, however, the proof-of-concept solution is simpler and shows the direct impact of DroidRA's APK booster on FlowDroid, without the intermediate steps. DroidRista[13] employs multiple modifications to not only help with dynamic code, but ICC in general as well, so it is not known to what degree does DroidRA help in data leak detection, and this proof-of-concept solution aims to find that out, because if this combination improve FlowDroid, there is then a freely available improved version of FlowDroid available to anyone, while currently for data-leak detection on Android, there is limited availability within open-source tools. The main advantage of this proof-of-concept solution is that it also combines *freely available tools*, which anyone can use for their own needs, while DroidRista has no openly available tool. This thesis also provides the exact setup and commands used in the development and evaluation phases, which the reader can use to use the tool on their own.

5.1 System Overview

The proof-of-concept solution follows the basic idea of other static analysis solutions, such as DroidRista[13], with the following components:

- **DroidRA**: APK boosting[47] to introduce the improvement of detecting leaks in dynamically loaded code to FlowDroid.
- **Source sink extraction**: SuSi[14] to get all sources and sinks in Android framework, which are used to identify whether a flow of data is a data leak.
- **Decompiler**: Soot's Jimple decompiler[41], which prepares the APK for analysis by transforming it into a format which can be analyzed by a Java program.
- **IFDS solver**: FlowDroid's extension of the Soot IFDS solver[7]. It contains the Android metadata parsing/extraction module, the call graph builder and the taint analysis module, all of which are used to detect links from sources to sinks in any given application.
- **Print results**: Simple output printing to console, which is used to report analysis results to a readable, structured format.
- **Evaluation**: The evaluation component runs the application against DroidBench[**?**], i.e. it runs the proof-of-concept solution against all test-cases in DroidBench and compares the results to FlowDroid and DroidRista.

The general flow of the tool can be seen in Figure 5.1 and the steps are described below:

- APK of the application is enhanced to better detect reflection (1)
- The boosted application is decompiled and ready for analysis (2)
- The sources and sinks file is extracted by SuSi (3)

- Metadata of the application is parsed; go through all classes and save all instances of source calls; build a call graph to visualize the flow of the application; in case a sink is reached when using a source as a starting node, mark it as a data path; analyze the path using tainted data and if the data read at source influences the output at the sink, data leak is recorded (4).
- Print all found data leaks (5)

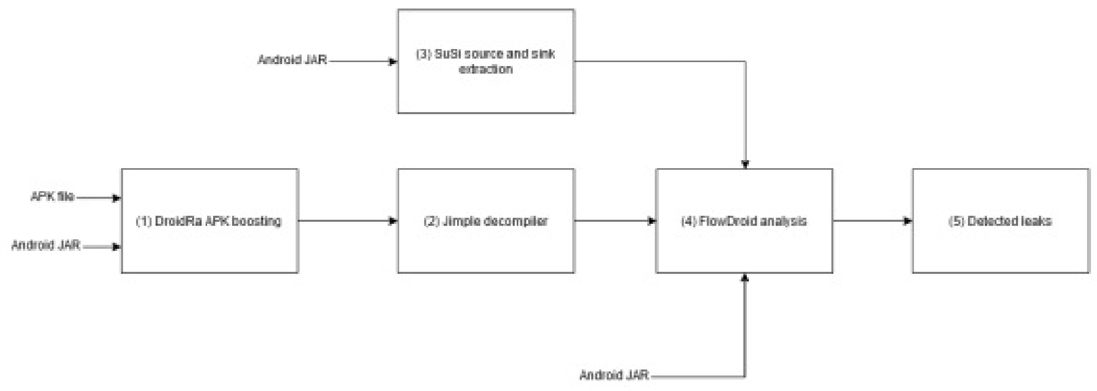


Figure 5.1: Component graph of the proof-of-concept solution.

5.2 Solution Components

This section lists and explains the core functionality of elements of the final proof-of-concept solution used for this thesis, as well as the modifications made to these tools for the purposes of the thesis.

5.2.1 DroidRA

The basics of how DroidRA[47] works have been explained in Section 3.5.8, and the system overview can be seen again in Figure 3.14. In summary, DroidRA analyzes an application looking for reflective calls and tries to replace the respective calls with standard Java calls which can get recognized by static analysis tools.

The example in Figure 5.2 shows the generic reflection pattern. Obtaining the methods and fields statically can occur directly (solid arrows), or can be obtained through initialization of an object of a class through a reflective constructor (dotter arrows). By using this pattern, DroidRA models most of the reflection usages in applications.

How the boosting of the code looks like is briefly demonstrated in Listings 5.1 and 5.2. The RAM module maps reflective calls in an application with the actual method calls, in the example shown, it would map *class.getMethod("setImei")* with *object.setImei()*. Every reflective call is additionally made explicitly in the boosted application. This way, when the static analysis starts, it does not have to deal with string literals. We see

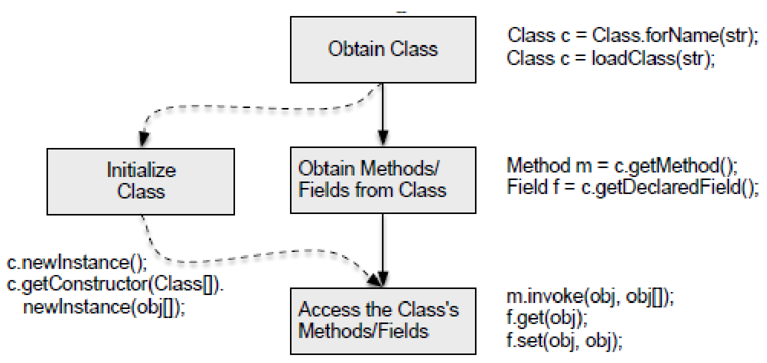

Obtain Class
Class c = Class.forName(str);
Class c = loadClass(str);
Initialize Class
Obtain Methods/ Fields from Class
Method m = c.getMethod();
Field f = c.getDeclaredField();
c.newInstance();
c.getConstructor(Class[]).
newInstance(obj[]);
Access the Class's Methods/Fields
m.invoke(obj, obj[]);
f.get(obj);
f.set(obj, obj);

Figure 5.2: Abstract pattern of reflection usage and some possible examples. (Taken from [47])

Object o = c.newInstance();, *m.invoke(o, imei); String s = (String) m2.invoke(o);* all followed up with a conditional check *if (1 == BoM.check())* and a new call, e.g. *o = new ReflectiveClass ();*. This is because the booster "enhanced" the original call with a direct method (constructor) call. The code behind the conditional will never trigger in a real executions, since *BoM.check()* will always return *false* when the application is running, however, the analysis tools do not know that and they analyze the flow with the code behind the conditionals taken into consideration.

Listing 5.1: Code excerpt of de.ecspride.MainActivity from DroidBench's Reflection3.apk. (Taken from [47])

```
TelephonyManager telephonyManager = //default;
String imei = telephonyManager.getDeviceId();
Class c = Class.forName("de.ecspride.ReflectiveClass");
Object o = c.newInstance();
Method m = c.getMethod("setImei" + "i", String.class);
m.invoke(o, imei);
Method m2 = c.getMethod ("getImei");
String s = (String) m2.invoke(o);
SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage("+49 1234", null, s, null, null);
```

Listing 5.2: The boosting results of the previous example. (Taken from [47])

```
Class c = Class.forName("de.ecspride.ReflectiveClass");
Object o = c.newInstance();
if (1 == BoM.check())
   o = new ReflectiveClass ();
m.invoke(o, imei);
if (1 == BoM.check())
   o.setImei(imei);
```

```
String s = (String) m2.invoke(o);
if (1 == BoM.check())
   s = (String) o.getImei();
```

In the following part, the detailed explanation of how DroidRA works is given:

JPM – Jimple Preprocessing Module Like with FlowDroid, DroidRA also uses Soot/Jimple as the decompiler and format of the analysis. After decompilation of the APK, DroidRA also employs the method of a dummy main method to use as a starting point of the analysis, which is used to build a control-flow graph and to traverse the whole code of an application.

DroidRA prepares all the code it can for analysis, including dynamically loaded code. They only focus on code present in the APK itself and not the code loaded from remote locations at runtime, since that is not feasible to do by static-analysis. The authors argue that the local dynamic code is far more common.

This dynamic code loading analysis is done via a heuristic. For every application, DroidRA traverses all embedded files and checks their file format. If it is a valid format (e.g. dat, bin, db) they look whether the file contains a .dex file and all retrieved .dex files are taken into consideration when analyzing the application.

RAM – Reflection Analysis Module The purpose of the Reflection Analysis Module is to map reflective calls of an application to their direct call counterparts. In the example given in Listing 5.1, the aim is to extract *de.ecspride.ReflectiveClass.getImei()* from *m2.invoke(o)*, *Method m2 = c.getMethod ("getImei")* and *Class c = Class .forName("de.ecspride.ReflectiveClass");*, i.e. to associate *m2* with the origin class.

According to their study, the authors model the reflection problem as a constant propagation problem within a CFG. This problem turns into resolving parameter values of reflective calls through data-flow analysis. It is also important that the analysis is context- and flow-sensitive to avoid issues such as "NoSuchMethodException" due to improperly mapped methods to classes.

To map reflective calls to the target values, DroidRA employs an analysis approach using COAL, the Constant Propagation Language[55] to specify the reflection problem. To use COAL, the reflection analysis problem is modeled in a generic way using common reflective call patterns (e.g. *Class.forName() -> getMethod() -> invoke()*). In COAL, the reflective methods are specified as objects with method name and class names being its fields. After modeling the reflection analysis, DroidRA authors built the reflection analysis on top of COAL to enable it run composite constant propagation to infer reflective call target values.

The example in Listing 5.3 shows the constant propagation of reflection values for class Method. Based on the specification, COAL generates a semi-lattice that shows the analysis domain. In the case of this example, Method has two fields where class types are fully qualified class names. In COAL, each value in the path is a tuple, and each tuple is a field value. For example, COAL solver models the value of object m at line 10 of Listing 5.3: *(first.Type; method1); (second.Type; method2)*. The first tuple represents the value of Method m from the first branch, the second tuple models the else branch.

To generate the transfer function for the getMethod calls, the solver works as presented in lines 15-17 in Listing 5.3. The *mod* statement specifies the signature of the *getMethod* and describes how it modifies the state of the program. The *gen* keyword tells that the method generates a new Method object. The field *declaringClass_method* indicates the reference to the class to which the method call belongs to, and the *replace* command replaces the reference *c* with the class call, while the *replace name_method* indicates that the *getMethod* call is replaced with the real call.

At the start of the analysis, all reflective values are marked and the COAL solver generates transfer functions for these values. The query statement on lines 18-19 indicate when the values of these objects of interest will be computed. In the case of Listing 5.3, the query indicates that values are computed when Method *m* calls *invoke*, i.e. on line 10.

Listing 5.3: Example of COAL-based reflection analysis. (Taken from [47])

```
1  // Java / Android code
2  Class c; Method m;
3  if (b) {
4    c = first.Type.class;
5    m = c.getMethod("method1");
6  } else {
7    c = second.Type.class;
8    m = c.getMethod("method2");
9  }
10 m.invoke(someArguments);
11 // Simplified COAL specification (partial)
12 class Method {
13   Class declaringClass_method;
14   String name_method;
15   mod gen <Class : Method getMethod(String, Class[])>{
16     -1: replace declaringClass_method;
17     0: replace name_method;}
18   query <Method : Object invoke (Object , Object[])>{
19     -1: type java.lang.reflect.Method;}
```

One issue with this approach is that if a parameter of a *m.invoke()* call is not specific (e.g.
m.invoke(Object[])) DroidRA cannot determine which element of the given array is the parameter and thus cannot infer the method call.

BOM – Booster Module The Booster Module takes the original APK as the input, as well as the results of the RAM module and outputs a new version of the APK where reflective calls are augmented with standard Java calls, while the original reflective calls remain untouched so that the application does not break during runtime. The augmented calls are only present to allow enhanced static analysis.

The approach made by BOM is straightforward. If a reflective call initializes a class, BOM adds a constructor call. If the reflective calls invokes a method, BOM adds a direct call. This is thanks to the mappings that RAM provides linking reflective calls with correct methods and classes.

The additional conditional checks (i.e. *check()*) added around every boosted call are there to avoid actually executing this code during runtime, since the only purpose of it is to be recognized by static analysis solutions, which do not consider the value *check()* returns.

BOM also performs additional instrumentations to help static analyzers. Since some static analyzers simply stop when encountering encrypted classes or other classes loaded from outside of the device, DroidRA mocks the classes that are called at points where they cannot be further analyzed. For example, instead of completely skipping a library which contains an encrypted archive, DroidRA mocks the encrypted archive and keeps the rest of the library to be analyzed.

Limitations Because the tool wasn't updated since 2017, the dependencies within it were outdated, so to avoid dependency management between DroidRA and FlowDroid, as they share many dependencies, DroidRA was run separately from the rest of the components.

The outdated dependencies also impacted the ability to actually run the tool, so we had to make modifications in order to get it to run, such as: fixing Soot *RuntimeExceptions* when parsing the APK and updating the Android.jar used.

Use in the Proof-of-Concept Solution For the proof-of-concept solution, DroidRA was mostly used out-of-the-box, but some modifications had to made so that it can run in the test environment.

The source code was taken from the original repository[1]. To simplify the use, a JAR of the tool was built, which can be used as a library for the proof-of-concept solution..

In its dependencies DroidRA also used old versions of Soot[41], which threw exceptions when running on DroidBench test cases. To resolve this, the Soot code was modified directly, since it was not given as e.g. an external dependency, but provided as a library in the source code. The modification removed a constraint which was blocking APK boosting. This constraint detected some methods of the input APK files as the wrong resolving level and blocked further analysis, so removing it meant that every method gets

[1]https://github.com/serval-snt-uni-lu/DroidRA

fully analyzed. This has potentially introduced performance issues, but for the purposes of this thesis, DroidRA has worked as intended.

The tool was then run with the following parameters:

```
Main.main(new String[]{apk.getAbsolutePath(), "./android-29.jar"});
```

There were no custom parameters set to limit the scope of the boosting. The only other parameter was the Android JAR of Android version 29 (10.0).

The JAR file is used as a compilation of all Android framework methods, which is used as an extension of the APK file. Because every application calls native Android methods at some point, the analysis of the APK file itself is not enough and we have to follow the flow of data into the Android framework methods as well. For this purpose, we need the JAR file containing all Android framework methods for a certain version so that the full data flow can be mapped and analyzed, or in this case, boosted.

5.2.2 Source and Sink Extraction

In order for the solution to be able to recognize if an application is accessing or sharing an Android device's data, it needs to know which method calls are source, and which are sink calls. The simplest solution is to give a comprehensive list of sources and sinks to the solution and when reading method calls, mapping them to sources and sinks.

To calculate this comprehensive list of sources and sinks, SuSi[14] was used. SuSi takes a JAR file of the full Android OS as the input and extracts the sources and sinks. This JAR has to be extracted from the Android emulator or a real device, since the SDK provided by Google contains stubbed methods and does not have the full implementations of all Android framework methods. SuSi uses machine learning on the OS JAR, based on a training sample of hand-annotated sources and sinks to find out which parts of the system are actual sources and which are sinks.

The following part explains how the machine-learning approach works in detail.

SuSi addresses two classification problems: for every Android method, SuSi decides if it is a source, sink or neither; the second part is refining the first step. Every method classified as neither source nor sink is ignored in the second step.

Machine Learning Primer SuSi uses standard supervised learning with a small subset of manually-annotated examples to train a classifier. The classification is performed with a set of features.

The classifier used is the margin classifier called Support Vector Machines (SVM), specifically the SMO classifier implementation used in Weka[33] optimized for minimal error. The basic principle is to represent training examples in two classes using vectors in a vector space and the trying to find a hyper-plane which separates the examples. Whether

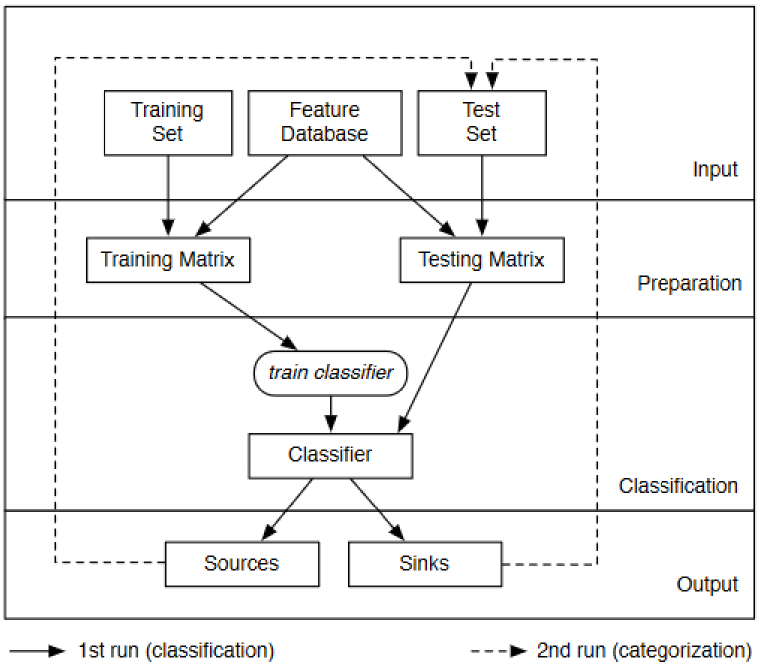


Figure 5.3: Machine learning approach of SuSi (taken from [14]).

or not a method is e.g. a sink or not is determined by the side of the plane on which the method is on.

The classifier is only capable of determining between two classes, but there are more than two in the first and second problem. This is solved by going recursively through the classes with a "one-against-all" principle where we determine e.g. first if a class is a sink or not in the first step, then every negative instance is checked whether it is a source or not.

Design of the Approach In Figure 5.3 we see the architecture of SuSi. It consists of four layers: input, preparation, classification and output. The approach runs in two rounds: first classifying methods as sources, sinks or neither, and second, categorizing them. The only difference between the rounds is that SuSi takes the output of the first round as additional input in the second round.

The input of the first classification problem is the training data, the unclassified set of Android API methods and the database of features for classification and categorization. After taking the input, SuSi builds two matrices: the first one shows which features does the training set have, and the second matrix shows which features do the uncategorized methods have. SuSi then trains the classifier based on the training matrix which then runs on the testing matrix to determine whether the methods in the set are sources, sinks or neither.

There are methods in Android which are both sources and sinks, but are scarce and there is no separate category for them and are thus marked as either one or the other based on the classifier.

In the second step, SuSi separates the test set into source test set and sink test set to further determine if a method is a source or a sink, with the training inputs also respectively being subsets of the training set from the first step. The feature set is also different and focuses on a set of sufficiently meaningful API methods. For sources 14 categories are used, some of which are: account, bluetooth, calendar, file, International Mobile Equipment Identity (IMEI), location and network. For the sinks, 17 categories are used, some of which are: account, audio, email, location, Short Message Service (SMS)/Multimedia Messaging Service (MMS) and Near-Field-Communication (NFC). If a method does not belong or does not fit any category, it is put into a no-category class.

The final output are a file containing the categorized sources and a file containing the categorized sinks.

Feature Database SuSi uses a set of 144 syntactic and semantic features to classify methods. All features together can give enough information to train a precise classifier. SuSi also takes advantage of regularity and redundancy in developer coding styles to discover sources and sinks since many are coded in a similar fashion.

Although there are 144 features, most of them are instances of a parameterized class. The following classes of features are used by SuSi: Method Name, Method has Parameters, Return Value Type, Parameter Type, Parameter Is An Interface, Method Modifiers, Class Modifiers, Class Name, Dataflow to Return, Dataflow to Sink, Dataflow to Abstract Sink, Required Permissions. All these may indicate whether a method is a source or a sink.

For all methods, each feature assumes one of three values: True - feature applies, False - feature does not apply, Not supported - feature cannot be decided.

SuSi's features for categorizing sources and sinks can be grouped as follows:

- Class Name - The method is in a class with a specific name substring.
- Method Invocation - The method invokes another method whose name has a specific string.
- Body Contents - The method body has a reference to a specific object type.
- Parameter Type - The method receives a specific type.
- Return Value Type - The method returns a specific type.

Permission-based features are not used for categorization, since requesting a permission does not directly relate to a functionality.

Dataflow Features Considering a method signature and syntax of its body is insufficient to detect sources and sinks and leads to low precision and recall. Taking data flows within the method into consideration improves both. The approach to analyze the data flows in the Android JAR similarly to how application files are analyzed by FlowDroid took to long since Android is larger than any application and the analysis would take too long, thus SuSi adopts a coarse-grained intra-procedural approximation. The results of this data-flow analysis are only one part of SuSi so it is enough for it to be imprecise to a given degree.

The data-flow features are based on the taint tracking within the Android method which needs classification. Depending on the feature, the method can be analyzed in one of the following ways: treating all method parameters as sources and calls to methods with a specific string as sinks; treating all parameters as sources and calls to abstract methods as sinks; treating all calls to specific methods as sources and the return call as the sink.

Based on how the analysis initialized, a fixed-point iteration runs with these rules: if the right hand side of an assignment is tainted, so is the left one; if one parameter of a known transformer method is tainted, the result is tainted; if one parameter of a writer method is tainted, the invoker object is tainted; if a method is invoked in a tainted object, the return value is tainted; if a tainted value is written to a field, the whole object is tainted.

If the analysis finds a source-to-sink connection, the iteration aborts and returns *true* for the analyzed method. If the analysis completes without finding any connections, it returns *false*.

Implicit Annotations for Virtual Dispatch SuSi is based on Weka[33], which does not have any internal knowledge about Java semantics, however, when annotating training data, SuSi also propagated the annotations up and down the hierarchy which discovered around 300 additional sources and sinks.

Prefiltering SuSi also performs prefiltering to skip analyzing unnecessary methods. This includes; removing abstract methods, methods which have no definition available in the Android platform version used, all private methods and methods in private classes. SuSi skipped over these private methods as they are only accessible through reflection. This however presents a problem as it was developed when no static analysis solution was analyzing reflective calls, which they now do.

Limitations Since this tool is was not updated for newer Android versions, it was ran on an older version of Android (4.2). However, this list of source calls has not changed significantly since and still provides a baseline for the tool.

Since SuSi has not been updated in 3+ years and FlowDroid changed how it takes the sources and sinks as input, the output of the list needs to be updated. SuSi provides separate lists for sources and sinks, while FlowDroid takes a single list of lines, of a certain RegEx pattern. To match the pattern, for this thesis a small script was written to adapt the lists into the given format.

Use in the Proof-of-Concept Solution SuSi is used for this thesis to provide the sources and sinks files for the static analysis. The tool was unable to run on newer Android versions, and the latest successful one was Android 17 (4.2). The IFDS solver also takes a single sources and sinks file as an input, while SuSi produces separate files for both, so for this thesis a post-processing step had to be introduced, which marked every entry in the sources file with *_SOURCE_* and every entry in the sinks file with *_SINK_*, and combined the two lists. If there was any conflict and an entry was marked with both *_SOURCE_* and *_SINK_*, then one of the entries was removed and the remaining one was marked with *_BOTH_*. The final, merged, file was used as an input for the static analysis.

The fact that an old version of the Android JAR file was used does not impact the goal of the proof-of-concept solution, which is to see improvement over the base FlowDroid tool, and if both tools use the same sources and sinks file as input, the difference can still be seen. The older version of Android can impact the overall precision and recall of the proof-of-concept tools, so as a comparison, the sources and sinks file extracted from the FlowDroid release version JAR is also used as an alternative input.

5.2.3 Decompiler

The decompiler which provides code for analysis is part of the Soot framework[41].

The overview of Soot in general can be seen in Figure 5.4, and the part relevant for the proof-of-concept solution is outlined in blue. The compiled class files, in this case extracted from the APK file, get input into the Jimple decompiler, which generates the 3-address intermediate bytecode representation. This intermediate representation gets analyzed, optimized and tagged, and in the end turned into Java source files. The source files also get mapped to object representations of the code. For example: the attributes of methods, structures and declarations get mapped to fields in their respective objects (e.g. SootMethod, SootField, SootClass).

For Android specific files, such as *.dex* files, a Soot plugin - **Dexpler**[17] is used.

As can be seen in Figure 5.4, this is only a part of what the framework offers. Soot also provides the basis for the static program analysis with *soot-infoflow*, as well as *call graph* generation.

Use in the Proof-of-Concept Solution The decompiler is actually used in both DroidRA steps and IFDS solver steps, and it is a module integrated into both, so there are no custom decompiler settings or changes made for this thesis.

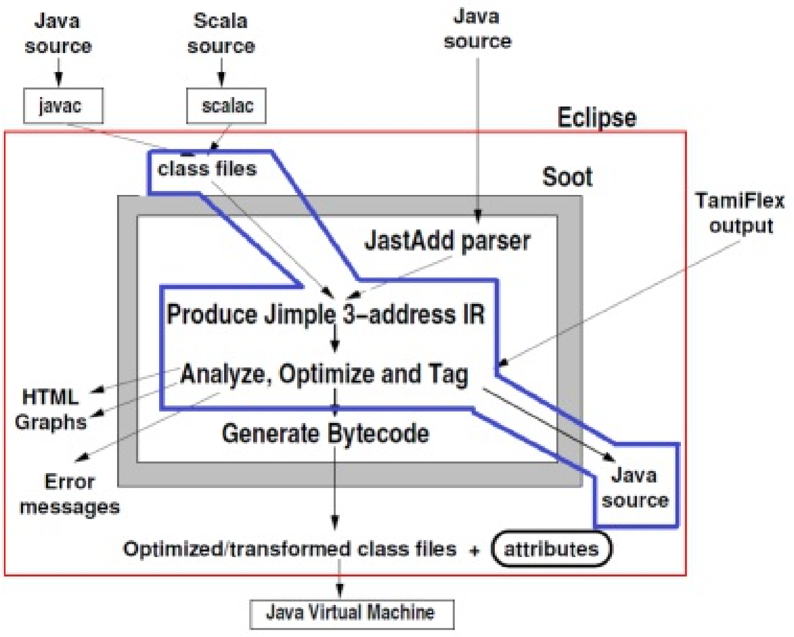


Figure 5.4: Overview of the Soot workflow. The thesis relevant part is outlined in blue. (Modified from [41])

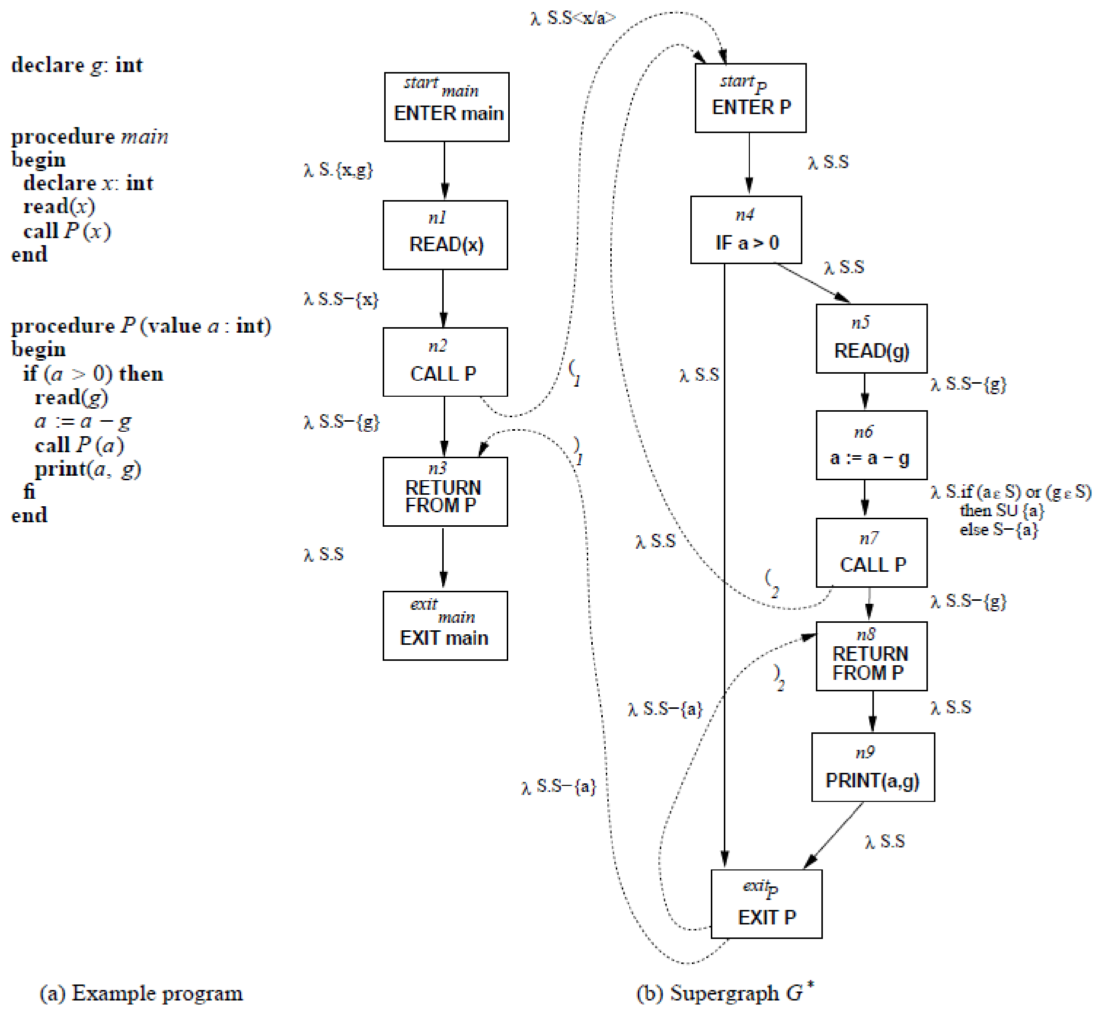
5.2.4 IFDS Solver

The base of the static analysis in this tool is the IFDS solver of FlowDroid. The solver itself is an extension of the native Soot solver. DroidRista[13] already combined DroidRA and FlowDroid in their solution, but in the meantime FlowDroid received four major releases including improvements to the analysis, IccTA and StubDroid. The algorithm did not change, it is the same extended IFDS algorithm as presented in Section 41, however, general optimizations of the tool have been made, such as the inclusion of a memory manager, dead code removal and multiple versions of the solver optimized for e.g. better garbage collection and faster solving (flow-insensitive).

At the time when FlowDroid's research was first published, the tool simply ignored all reflective calls. Since then they developed analysis of reflective calls, however, since this is not documented in any research articles, we do not know the effectiveness of their implementation. This is a point that is also considered in the final evaluation.

In its essence, what FlowDroid does is the approach suggested in the paper from Reps[59]: transforming a program analysis problem into a graph reachability problem. The goal there is to map every program statement and method call into nodes of a graph, and finding out whether a source can reach a sink is a matter of finding out whether there exists a path from source that reaches a sink within the constructed graph.

An example of the transformation is given in Figure 5.5. The transformation orders the program statements into a graph with a start and an end node. If we compare the example program, its procedures and calls to the constructed supergraph, we can see that there is a path from *read(x)* to *print(a,g)*, with *a* being an alias for *x*. This would constitute a leak.



```
declare g: int

procedure main
begin
  declare x: int
  read(x)
  call P(x)
end

procedure P(value a : int)
begin
  if (a > 0) then
    read(g)
    a := a − g
    call P(a)
    print(a, g)
  fi
end
```

Figure 5.5: An example of a program (a) and its supergraph (b). (Taken from [59])

Use in the Proof-of-Concept Solution The IFDS solver is the analysis module of the proof-of-concept solution and it is mostly used out-of-the-box as provided by Arzt et al.[2]. Every application analysis is run with the following command:

```
MainClass.main(new String[]{"-a", apk.getAbsolutePath(),
```

[2]https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.8

```
"-p", "./android-29.jar", "-s", "./SourcesSinks.txt",
"-o", "./report.xml"});
```

This is the basic run command for FlowDroid, without any extra parameters to limit any functionality and it provides a comprehensive analysis. The JAR of Android version 29 is used, for the same reasons mention in the DroidRA section 5.2.1.

5.2.5 Result Printer

For every application, the solution produces a report similar to the one in Listing 5.4, with the following information: analysis success, source-sink links discovered, performance data (e.g. memory consumption, runtime, total found sources and sinks). In the example below, the solution found:

- A total of 3 sources and 5 sinks in the analyzed application
- One leak: Activity2 reads the deviceId (IMEI) from a source and stores it, while Activity1 reads and sends it to a sink via SMS. The number of *<Result>* tags represents the number of total leaks found in an application.

The report also contains performance data consisting of memory consumption in MB and total analysis runtime in seconds.

Listing 5.4: Example analysis report for ActivityCommunication1.apk

```
-<DataFlowResults TerminationState="Success" FileFormatVersion=
    ↪ "102">
-<Results>
-<Result>
-<Sink Method="<de.ecspride.Activity1: void onCreate(android.os
    ↪ .Bundle)>" Statement="virtualinvoke $r3.<android.telephony
    ↪ .SmsManager: void sendTextMessage(java.lang.String,java.
    ↪ lang.String,java.lang.String,android.app.PendingIntent,
    ↪ android.app.PendingIntent)>("+49", null, $r2, null, null)"
    ↪ >
<AccessPath TaintSubFields="true" Type="android.telephony.
    ↪ SmsManager" Value="$r3"/>
</Sink>
-<Sources>
-<Source Method="<de.ecspride.Activity1: void onCreate(android.
    ↪ os.Bundle)>" Statement="$r3 = staticinvoke <android.
    ↪ telephony.SmsManager: android.telephony.SmsManager
    ↪ getDefault()>()">
<AccessPath TaintSubFields="true" Type="android.telephony.
    ↪ SmsManager" Value="$r3"/>
```

```
</Source>
-<Source Method="<de.ecspride.Activity2: void onCreate(android.
    ↪ os.Bundle)>" Statement="$r4 = virtualinvoke r3.<android.
    ↪ telephony.TelephonyManager: java.lang.String getDeviceId()
    ↪ >()">
<AccessPath TaintSubFields="true" Type="java.lang.String" Value
    ↪ ="$r4"/>
</Source>
</Sources>
</Result>
</Results>
-<PerformanceData>
<PerformanceEntry Value="1" Name="TotalRuntimeSeconds"/>
<PerformanceEntry Value="396" Name="MaxMemoryConsumption"/>
<PerformanceEntry Value="3" Name="SourceCount"/>
<PerformanceEntry Value="5" Name="SinkCount"/>
</PerformanceData>
</DataFlowResults>
```

Use in the Proof-of-Concept Solution The result printer is actually a part of FlowDroid and it is used by the proof-of-concept approach as a basis for the evaluation section. The main parts of it (i.e. concrete data leaks found, memory consumption, time taken) are used as a basis for comparison with the baseline of the evaluation, as well as the unmodified FlowDroid[15] and DroidRista[13].

5.2.6 Evaluation

Finally, the evaluation of the proof-of-concept solution is done via a script running the tool(s) for every test-case in DroidBench as well as 50 real-world applications. The description of the test cases and real-world applications is in Chapter 6.

The results are then compared with those of DroidRista, and the unmodified FlowDroid. For DroidRista there is no source code available, and no released version, so the comparisons are only made with the results which are provided in [13].

The component versions used in the proof-of-concept solution used in evaluation are:

- *Android JAR* - The Android OS JARs were retrieved from a GitHub repository[3]. For *DroidRA* and *FlowDroid* components, Android version 29 was used, while for *SuSi*, Android 17 was used. Android 29 was selected because it was the most used version of Android at the time of development (September 2020)[9]. Android 17 was used as it was the last version which could be analyzed by SuSi.

[3]https://github.com/Sable/android-platforms

- *DroidRA* - Commit *b766a32* in the GitHub repository[4]. This version is selected since it is the latest version at the time of development (September 2020).

- *SuSi* - Commit *df72e9d* in the GitHub repository[5]. As an alternative to the SuSi-provided source and sinks file, an additional source and sinks file was used. The second file was extracted from the FlowDroid JAR and it is used there as the default source and sinks file. This version is selected since it is the latest version at the time of development (September 2020).

- *Decompiler* - the decompiler provided in Soot 4.2[6]. This version is selected since it is the latest version at the time of development (September 2020).

- *FlowDroid* - Version 2.8[7]. This version is selected since it is the latest version at the time of development (September 2020).

All applications analyzed with the proof-of-concept solution were also analyzed by the unmodified FlowDroid 2.8.

Two different versions of the sources and sinks file are used to test whether different configuration files influence the results and to what degree. The two versions are the following:

- *SuSi-gen* - The sources and sinks file generated by SuSi from the Android 17 JAR. This list contains more entries, however, it is a machine-learning-generated list, may contain wrongly marked entries (which may introduce false-positives), and it is extracted from an older version of Android, so it may be missing entries too.

- *SuSi-ex* - The sources and sinks extracted from the FlowDroid release JAR, which is used by FlowDroid as the default file. This list is smaller than the list generated from SuSi so using it may miss more data leaks. The details of how this list is compiled are unknown, however, due to the size, it is possible that this is a manually generated list, containing the most commonly used sources and sinks methods in Android.

For the results, the precision, recall and F-Score are calculated based on the DroidBench results. The numbers of true and false positives, as well as true and false negatives are compared and aggregated between the proof-of-concept solution and FlowDroid. And the overall scores of precision, recall and F-Score are compared to DroidRista.

[4] `https://git.io/Jtt1b`
[5] `https://git.io/Jtt1x`
[6] `https://github.com/soot-oss/soot/releases/tag/v4.2.0`
[7] `https://github.com/secure-software-engineering/FlowDroid/releases/tag/v2.8`

The real-world application were used to compare the proof-of-concept solution to FlowDroid and to test real-world performance, since the DroidBench applications are relatively small applications which are analyzed within seconds. However, whether FlowDroid or the proof-of-concept solution detected more leaks does not imply that one solution is better than the other, since for the real-world applications, the real number of leaks is unknown.

CHAPTER 6

Evaluation of the Private Data Leak Detection Solution

This chapter describes the setup of the benchmarking tools, explain what the test baseline is and show the initial results of the tool. Aside from that, we discuss why we got these results and show the limitations within the tool.

6.1 Benchmarking the Performance

To benchmark the proof-of-concept solution, the tool was run against the DroidBench 3 test suite. Aside from DroidBench, 50 of the most popular Android applications were tested for potential data leaks to see what the risks are in the real-world applications.

The 50 Android applications were selected based on the number of downloads. The list consists of applications from the play store top chart[1] and the list of most downloaded applications of all time[2]. These lists were generally looking for applications with over 100 million downloads and excluded many pre-installed applications and redundant applications (e.g. it includes Facebook, but not Facebook Lite). Regarding the application versions, the latest versions of the APKs were downloaded as of October 30th, 2020. The full list of applications and their respective results can be seen in the benchmarking results.

6.1.1 Benchmarking Setup

The benchmarks were executed on a Windows OS system, with an AMD Ryzen 5 3600 6-core CPU and 16GB of RAM. This information is relevant due to the fact that not

[1]https://play.google.com/store/apps/top?hl=en

[2]https://en.wikipedia.org/wiki/List_of_most-downloaded_Google_Play_applications

only is the effectiveness of the solution tested, the efficiency is also tested. The efficiency was tested to see if the DroidRA APK boosting made an impact on the computational complexity of the applications for FlowDroid.

The benchmark consisted of the following steps:

1. Run DroidRA on 190 test-case applications developed within DroidBench, together with 8 test applications developed for DroidRA
2. Run Proof-of-concept solution on all 198 applications and record the execution times for the whole suite
3. Run FlowDroid on all 198 applications and record the execution times for the whole suite
4. Record all the outputs in one file for result comparison
5. Repeat steps 1-4 20 times to extract average run times

Of the 198 test cases, not all have a leak that has to be detected. Some are designed to catch a flaw within the analysis by waiting for a false positive to be reported, i.e. the analysis is valid if they resolve a data flow in a proper way.

The test applications were split according to various categories by DroidBench developers in order to determine which type of data leaks are harder to detect by which solution and which are being correctly detected. They were split into the following categories:

- **Aliasing** - Designed to test proper alias resolving.
- **Arrays and Lists** - Tests detection of leaks covered up through the use of arrays and lists.
- **Callbacks** - Tests detection of leaks hidden within callbacks, which often have to be in specific order.
- **Dynamic Code Loading** - Designed to test whether a tool can detect leaks in dynamically loaded code.
- **Field and Object Sensitivity** - Tests detection of leaks of tainted and untainted data within an object.
- **Inter-App Communication** - Tests detection of leaks involving multiple applications.
- **Inter-Component Communication** - Tests detection of leaks involving various communication between multiple components.

- **Lifecycle** - Tests detection of data leaks distributed within different lifecycle elements.
- **General Java** - Tests detection of data leaks using various Java leak methods non-specific to Android.
- **Miscellaneous Android Specific** - Tests detection of data leaks using various methods specific to Android.
- **Implicit Flows** - Tests detection of data leaks hidden in implicit flows.
- **Reflection** - Tests detection of data leaks originating from fields of a reflective class.
- **Reflection ICC** - Tests detection of data leaks caused by replacing ICC elements with reflective calls, e.g. reflected intents.
- **Self-Modification** - Tests detection of data leaks in applications which modify their execution through native code to change the source and sink targets.
- **Threading** - Tests detection of data leaks obfuscated by passing data between threads.
- **Emulator Detection** - Tests whether the application is running on an emulator or on a real device.
- **Native Code** - Tests detection of data leaks where data flow is obfuscated with native calls.
- **Unreachable Code** - Tests whether the data leak detection tool can identify that the data flow containing a leak is also not executable in real-world use.

The detailed descriptions of every test case can be seen on the DroidBench GitHub [**?**].

The APKs of the 50 selected real-world applications were obtained through APKMirror[3] and APKPure[4].

The reasoning behind having both test applications and real-world applications is that:

- Through test application we can determine the efficiency and effectiveness of the proof-of-concept solution is at detecting concrete vulnerabilities, since we know what the baseline is. It is known for every test case which leaks it contains, so we know the exact percentage of successful detection.

[3]https://www.apkmirror.com
[4]https://apkpure.com

- Through real-world applications we see how the proof-of-concept solution will behave on real cases. Developing a solution which detects 100% of leaks in controlled test cases is one challenge, while having it successfully detect leaks in application that actually affect users is another challenge. The goal of this thesis is to have a solution which can warn users that an application may leak their information. The test cases are approximately 300-400KB in size, while the real world application were on average 70MB in size (approx. 170x-230x larger). The difference is also not only in size. Applications which are not open-source, also often obfuscate their code to protect their intellectual property, which leads to difficulties of leak detection.

6.1.2 Benchmarking Results

In this subsection, first the results of the DroidBench analysis are compared between the proof-of-concept solution and FlowDroid. After that the results of the 50 real-world applications analysis is compared between the proof-of-concept solution and FlowDroid.

Test group	Total leaks	Leaks found	True positive	False positive
Aliasing	2	**4 (5)**	2 (2)	**2 (3)**
Arrays and Lists	4	12 (12)	4 (4)	8 (8)
Callbacks	18	18 (18)	16 (16)	2 (2)
Dynamic Code Loading	3	0 (0)	0 (0)	0 (0)
Field and Object Sensitivity	2	8 (8)	2 (2)	6 (6)
Inter-App Communication	11	25 (25)	10 (10)	15 (15)
Inter-Component Communication	19	**48 (47)**	18 (18)	**30 (29)**
Lifecycle	24	23 (23)	19 (19)	4 (4)
General Java	23	27 (27)	20 (20)	7 (7)
Misc. Android Specific	12	16 (16)	12 (12)	4 (4)
Implicit Flows	9	4 (4)	1 (1)	3 (3)
Reflection	17	17 (17)	12 (12)	5 (5)
Reflection ICC	11	16 (16)	9 (9)	7 (7)
Self-Modification	3	0 (0)	0 (0)	0 (0)
Threading	6	6 (6)	6 (6)	0 (0)
Emulator Detection	18	19 (19)	17 (17)	2 (2)
Native Code	5	2 (2)	2 (2)	0 (0)
Unreachable Code	0	3 (3)	0 (0)	3 (3)
Total	**187**	**248 (248)**	**150 (150)**	**98 (98)**

Table 6.1: DroidBench results of the proof of concept solution, run with *SuSi-gen* file. (FlowDroid results in parentheses)

DroidBench Results

The following tables shows the DroidBench results. The numbers in parentheses are FlowDroid numbers, and the numbers in front of the parentheses are the proof-of-concept solution numbers. Table 6.1 shows the results when running the tool with the *SuSi-gen* file, and Table 6.2 shows the results with the *SuSi-ex* file. The same goes for Tables 6.3 and 6.4 where the first one represents results with the *SuSi-gen* file, and second the results with the *SuSi-ex* file.

Test group	Total leaks	Leaks found	True positive	False positive
Aliasing	2	2 (2)	1 (1)	1 (1)
Arrays and Lists	4	9 (9)	4 (4)	5 (5)
Callbacks	18	17 (17)	13 (13)	4 (4)
Dynamic Code Loading	3	0 (0)	0 (0)	0 (0)
Field and Object Sensitivity	2	2 (2)	2 (2)	0 (0)
Inter-App Communication	11	2 (2)	1 (1)	1 (1)
Inter-Component Communication	19	6 (6)	6 (6)	0 (0)
Lifecycle	24	18 (18)	18 (18)	0 (0)
General Java	23	22 (22)	19 (19)	3 (3)
Misc. Android Specific	12	11 (11)	10 (10)	1 (1)
Implicit Flows	9	0 (0)	0 (0)	0 (0)
Reflection	17	4 (4)	4 (4)	0 (0)
Reflection ICC	11	0 (0)	0 (0)	0 (0)
Self-Modification	3	0 (0)	0 (0)	0 (0)
Threading	6	5 (5)	5 (5)	0 (0)
Emulator Detection	18	16 (16)	16 (16)	0 (0)
Native Code	5	0 (0)	0 (0)	0 (0)
Unreachable Code	0	3 (3)	0 (0)	3 (3)
Total	**187**	**117 (117)**	**99 (99)**	**18 (18)**

Table 6.2: DroidBench results of the proof of concept solution, run with the *SuSi-ex*. (FlowDroid results in parentheses)

The tables contains the following information:

- The column "Total leaks" presents the number of leaks that the test applications contain per category (the actual number of leaks).
- The column "Leaks found" presents the number of leaks that the proof-of-concept solution found in these test cases.
- The column "True positive" presents the number of leaks that the proof-of-concept solution correctly detected.

- The column "False positive" presents the number of leaks that the proof-of-concept marked as leaks, but which were incorrect guesses.

	FlowDroid	Proof-of-concept	DroidRista
Precision	60.4%	60.4%	98.4%
Recall	80.2%	80.2%	96.9%
F-Score	0.689	0.689	0.98

Table 6.3: DroidBench overall results with *SuSi-gen* file

	FlowDroid	Proof-of-concept	DroidRista
Precision	84.6%	84.6%	98.4%
Recall	52.9%	52.9%	96.9%
F-Score	0.651	0.651	0.98

Table 6.4: DroidBench overall results with *SuSi-ex*

Table 6.3 shows the overall results in precision and recall between FlowDroid and the proof-of-concept solution with both sources and sinks files. The DroidRista column is the same in both tables, as it is unknown what sources and sinks file the authors used.

Top 50 Real-World Applications Analysis Results

In the following Table 6.5, the findings of the analysis of the 50 selected Android applications are shown. The table does not contain 50 rows, due to the fact that some applications were unable to get analyzed because the DroidRA boosting step timed out during the reflective call resolving. The applications which failed to get boosted are: Google Docs, Facebook Messenger, Snapchat, Clash of Clans, Twitch and Line. Some applications got different results with different sources and sinks files. One such application is e.g. Candy Crush Saga, where 4/186 is shown in the proof-of-concept column. This means that the application was able to find 4 leaks with the *SuSi-ex* file, and 186 with the *SuSi-gen* file. The results of the remaining applications are as follows:

6.2 Analysis and Comparison of the Benchmarking Results with Existing Solutions

This section will discuss the overall results of the evaluation. First starting with the DroidBench result discussion, and a comparison with the tool closest to the proof-of-concept solution - DroidRista. After that, the results of the top 50 real-world applications analysis are discussed.

[5]FlowDroid ran with *SuSi-gen* file only since it ran into an infinite loop with the *SuSi-ex* file.

Nr.	Application	Leaks found (PoC)	Leaks found (FlowDroid)	Note
1	Adobe Acrobat Reader	0	0	
2	Amazon	20	21	
3	Among Us	0	0	
4	Audible	1	1	
5	Candy Crush Saga	4/186	186	Different source/sink file[5]
6	Chrome	8	8	
7	Discord	N/A	N/A	Infinite loop
8	Dropbox	N/A	N/A	Infinite loop
9	Excel	N/A	N/A	Infinite loop
10	Facebook	N/A	N/A	Infinite loop
11	Flibboard	9	9	
12	Gmail	N/A	N/A	Infinite loop
13	Google	0	0	
14	Google Maps	N/A	N/A	Infinite loop
15	Hill Climb Racing	1	1	
16	Instagram	N/A	N/A	Infinite loop
17	MX Player	N/A	N/A	Infinite loop
18	Microsoft Teams	0	0	
19	Netflix	6/277	277	Different source/sink file[5]
20	OneNote	N/A	N/A	Infinite loop
21	PayPal	N/A	N/A	Infinite loop
22	PicsArt Photo Studio	N/A	N/A	Infinite loop
23	Pinterest	N/A	N/A	Infinite loop
24	Pou	0	0	
25	PowerPoint	N/A	N/A	Infinite loop
26	Prime Video	1	1	
27	SHEIN	286	286	
28	ShareIt	N/A	N/A	Infinite loop
29	Shazam	N/A	N/A	Infinite loop
30	Subway Surfers	9	9	
31	Talking Tom	0/206	206	Different source/sink file[5]
32	Telegram	N/A	568	Ran out of memory with PoC
33	Temple Run 2	0	0	
34	TikTok	2	2	
35	Tinder	2	2	
36	Twitter	N/A	N/A	Infinite loop
37	UC Browser	10	10	
38	Uber	N/A	N/A	Infinite loop
39	Viber	0	0	
40	Waze	1	1	
41	Whatsapp	N/A	N/A	Infinite loop
42	Wish	5/415	415	Different source/sink file[5]
43	Youtube	4	4	
44	Zoom	N/A	N/A	Infinite loop

Table 6.5: Results of top 50 real-world application analysis

6.2.1 Evaluation of the DroidBench Results

- The proof-of-concept solution does not detect any additional leaks in comparison to the regular FlowDroid. This can imply that the tool already developed a good enough reflection detection that they do not need an extra pre-processing step, or it can imply that the test cases provided by FlowDroid and DroidRA are not complex enough that base FlowDroid cannot detect them.

- The analysis took on average 380s with the proof-of-concept solution APKs and 400s with base FlowDroid. This is an improvement of 4.8%, however, the DroidRA boosting took significantly longer than 20 seconds, so this improvement is automatically negated.

- The analysis by the proof-of-concept solution used 5% less memory in comparison to the base FlowDroid. This percentage is even higher in the test APKs designed to test reflection analysis capabilities, at 10-15%. This implies that the DroidRA APK boosting makes an impact on the computational complexity of FlowDroid, by reducing the need to analyze the reflective calls on its own.

- The size of the boosted APKs was approximately 15% higher than the base APKs. This means that every boosted application has a significant overhead in size and extra code, but that does not reflect negatively on the time it takes to process the files and the memory consumption. This means that the extra file size is insignificant since it provides other benefits such as improved memory consumption and run time.

- Running the tool with the *SuSi-gen* file produced more false positives than running it with the *SuSi-ex* file (17 false positives versus 98 false positives). Although the list from FlowDroid may be more optimized for the specific test cases developed for it, the difference in the results does ask the question which results show the true analysis of FlowDroid. It may be that using the older and more comprehensive list of sources and sinks triggers the detection of infeasible leaks coming from and going to source/sink calls that are deprecated, or it may be that the list is misconfigured in some way. The older list was also extracted by a machine learning algorithm, while the FlowDroid list was possibly hand made, so it may be due to an issue in the algorithm.

- Running the tool with the *SuSi-gen* file however, did result in more true positives found (99 versus 150). Most significant difference is in Inter-App Communication (9 more detected) and Reflection (8 more detected) leaks. Overall the 1*SuSi-ex* file resulted in higher precision (+20.2%) and running it with the *SuSi-gen* file resulted in better recall (+27.3%). This may imply that the best optimization or improvement that one could develop for FlowDroid would be to develop an optimal sources and sinks file.

- Between the results of the DroidBench analysis done by FlowDroid and the proof-of-concept solution, there were only two minor differences, and only with the *SuSi-gen* file. That is that the proof-of-concept solution reported one false-positive more in the *Inter-Component Communication* category and one false positive fewer in the *Aliasing* category than FlowDroid. This however, does not affect the overall results, but does show that there is a difference in how FlowDroid and the proof-of-concept solution analyze an application.

- In general, DroidRA made improvements to performance in FlowDroid's analysis. In comparison to the current version, the proof-of-concept solution did not make improvements in data leak capabilities, but this thesis based the limitations of FlowDroid on the 2013 version and tried improving on them. The proof-of-concept solution is able to detect reflection-based hiding techniques, something that the original version was not able to, however, the new FlowDroid version is also capable of it. The old FlowDroid and the proof-of-concept solution are not directly compared as there is no openly available version of FlowDroid 1.0, and here only the version 2.8 is used to showcase the current full capabilities of the tool.

The main idea was to boost FlowDroid's reflection resolving capabilities, but it seems that FlowDroid's developers already improved their solution before this approach. The results made in this thesis could be compared to the results from the original 2013 paper on FlowDroid, however, since the proof-of-concept solution relies on the 2.8 version of FlowDroid, comparing it with FlowDroid 1.0 would only highlight the improvements in FlowDroid itself[15].

6.2.2 Comparison of the Proof-of-Concept Solution with DroidRista

After seeing these results, and taking a closer look a the results of the similar approach in DroidRista[13], we can see that they only mentioned a comparison of their tool with the base FlowDroid with a basic table lacking details of the test cases. They show that FlowDroid could not detect ICC, implicit flows or reflection, but in all DroidBench test-cases in the case of this thesis, it could. DroidRista was published in 2019 and was most likely developed through 2018. This may imply that FlowDroid improved their ICC, implicit flow and reflection detection capabilities, or that DroidRista used FlowDroid's test data from 2013.

The comparison of the DroidBench data can be seen in Table 6.6. The results of FlowDroid and the proof-of-concept solution are the ones achieved with the *SuSi-gen* file, since it achieved a higher F-Score. One thing to note here is that the exact parameter with which they ran the tools is not known. They may have had a more fine-tuned sources and sinks file, or a different Android JAR file more suited to the test cases. DroidBench also receives a new test-case frequently, so in the past two years, there may have developed more test cases which DroidRista would fail. This is why this comparison is only superficial and not statistically significant.

If they say in this table that they detected 124 positive cases and were right in 98.4%, that means that DroidBench had 126 leaks to be detected at that time. Today, DroidBench has 187 leaks in their test cases. This means that since the test suite grew, their recall results today could be anywhere from 66% to 98.9%, since it is unknown how it would handle the 61 new leaks.

	FlowDroid	Proof-of-concept	DroidRista
True positive	150	150	124
False positive	98	98	2
False negative	37	37	4
Precision	60.4	60.4	98.4
Recall	80.2	80.2	96.9
F-Score	0.689	0.689	0.98

Table 6.6: Comparison of DroidBench results with DroidRista (DroidRista was originally tested on a different DroidBench version)

6.2.3 Evaluation of Results of the Top 50 Real-World Android Applications' Analysis

The initial results for the top 50 real-world applications benchmark show the following:

- The boosting took a relatively long time (3-10 minutes per application). During the boosting, watching the output logs, there was also a large number of call transformations happening. It seems that many of the selected applications obfuscate a large portion of their code, presumably to protect their intellectual property. If we selected only open-source applications, the percentage of obfuscated code would probably be lower.

- The long boost times even on a desktop PC, also mean that it is not very likely to run this tool on an Android device, and an analysis of a typical popular application might take up to an hour.

- The analysis of *un-boosted* applications took on average 10-300 seconds. This means again that running such a process on an Android device would take significantly more time and thus be unusable.

- The analysis of *boosted* applications took on average 10-300 seconds. There was virtually no time improvement of when running the analysis on a boosted versus an un-boosted application. The analysis of some applications ran faster and with lower memory consumption when boosted and some were analyzed faster and with lower memory consumption when un-boosted. There was no statistically significant difference.

- Running the analysis with the *SuSi-gen* file discovered more leaks, as expected, but failed to complete in many situations due to the process running out of memory. Limiting the source and sink file to key sources and sinks would potentially be a good performance improvement, but would however hinder the leak detection capabilities. As shown in Table 6.5, the using the *SuSi-gen* file found up to 80 times more leaks in some applications.

- 6 applications stalled during the DroidRA APK boosting phase. All 6 failed during transformation of obfuscated dynamically loaded code to actual method calls. The reason for this may be that these applications employ a new type of obfuscation developed after DroidRA released and it was not designed to tackle it.

- 19 applications entered infinite loops during analysis. This was mostly during call-graph construction, which is presumably due to actual loops being formed in these graphs.

- One thing to note is the difference in the number of detected leaks between some applications, with most having 0-10 and some having over 200 leaks found. The applications with the large numbers of leaks have most of the leaks coming when calling third-party SDKs, which were fully analyzed. This was one of the main concerns in the vulnerability section. Applications implement third-party analytics SDKs and this causes many leaks if improperly regulated. That said, it still may be that these are primarily false-positives.

- Since we do not have a baseline to know which of these leaks are true and which are false, we cannot infer any significant data to say that one solution is better than the other, however we still can see that there is no significant improvement brought on by the additional boosting done by DroidRA. We also cannot say that because there are an X number of leaks found in one application, that this application is insecure, since we do not know how many are true positives. The real-world applications are by orders of magnitude more complex and convoluted and may contain many more false positives. Also, since permissions are not taken into account, we do not know how many of these leaks are through granted permissions given by the user.

6.3 Limitations of the Proof-of-Concept Solution Leak Detection Capabilities

This solution comes with multiple limitations. All of which are explained in this section:

- The main limitations are the dependence on FlowDroid and DroidRA. Since both solutions are open-source, they could have been expanded and improved in other ways, and DroidRA was tweaked to be able to run correctly with FlowDroid, but the main parts of both solutions were not touched. Due to this dependence, any limitations of both approaches were inherited into this approach, with the exception of dynamic code resolution, which was improved by DroidRA.

- Dependence on the old SuSi source and sink list - Because the SuSi tool is not functional with newer Android versions and the output is incompatible with FlowDroid too, the testing had to be done on an older version of the source and sink file, which was manually formatted to fit the current FlowDroid format. This introduced a limitation that possibly makes leaks through new source and sink methods undetectable, and any misconfigured or poorly formatted line in the source and sink file may miss a leak. On the other hand, the newer, but the narrower source and sink file from FlowDroid may detect more leaks in the DroidBench test suite, but may miss leaks in real-world applications, as shown in Table 6.5.

- Influence of different versions of the Android JAR unknown - One input file of the analysis is the Android JAR, which is used both for DroidRA and FlowDroid steps of the solution was static the same across the testing phase. The JAR file of Android 29 was used, and it is unknown how different the results would be when using any other version either one or ten versions in difference. This part of the evaluation was skipped due to time constraints. Repeating everything with just one other Android version would double the effort of the evaluation.

- Analysis of real-world applications takes a long time, and in some cases does not work. Some of the 50 selected applications were unable to be analyzed because they got stuck in infinite loops during the DroidRA APK boosting phase. The long analysis time on a desktop PC mean that running such an analysis on an Android device, even with the latest and strongest mobile CPU, would take most likely 3-5 times as long. Running an analysis that takes 30-60 minutes on a phone before installing an application would not be very user friendly and would potentially have a large impact on the battery.

- Another performance issue was that in some cases the analysis ran out of memory and crashed. This happened on a PC with 16 GB of memory, and 4 GB was dedicated to the process. This further makes running the analysis on an Android device less feasible.

- Limitations inherited from FlowDroid are: all connections from sources to sinks are considered leaks. Whether the application had the permission is not the concern. To verify if the application sent data that it was not allowed to, the users would have to check the permissions for themselves. Also, the evaluation is heavily dependent on the source and sink file, as shown in Table 6.5. This means that in order to work to its full potential, FlowDroid needs an optimized source and sink file that covers all source and sink calls that exist on Android. There is unfortunately no such list for newer versions of Android and SuSi has not been updated in years.

- Limitations inherited from DroidRA are: the approach to resolve reflection is fairly simple. Replacing reflective calls with real calls works only if there is no further hiding technique involved. Any other technique mentioned in Section 3.4 in combination with reflection would be undetected by DroidRA. Encryption, obfuscation and altered flow would all be undetected due to the fact that DroidRA is programmed only for standard reflective calls. During the analysis of the 50 real-world applications, there were many cases where DroidRA was stuck while trying to resolve obfuscated code, and these applications had to be skipped. Another limitation of DroidRA is that it analyzes only the dynamically loaded code that is present in the APK, while completely missing anything downloaded from remote locations at run-time.

CHAPTER 7

Conclusion and Future Work

The goal of this thesis was to try to find and develop a new or an improved way to detect data leaks on Android. The solution is of expansive nature and it builds on existing solutions to achieve higher results.

The first part of the thesis describes the background of Android OS, security and privacy and data leaks in general. It describes actual problems that concern Android users when it comes to data leaks. It was discovered that a significant portion of the Android user base was unaware of potential leaks of their private data, and a majority was also found to be uncomfortable when being made aware when each application accessed some segment of private data.

The next part of the thesis describes how data leaks are possible in the world of software as well as how they work on Android specifically. The major vulnerabilities that exist on Android, such as ICC, third-party library leaks, inter-app leaks and a variety of proposed solutions to these vulnerabilities were presented briefly.

There are also other proposed solutions that were discovered during the research phase, but were further analyzed in the later parts of the thesis. The reasons why they were not showcased was one of the following: because they did not provide enough information about their concrete implementations and results, they did not analyze the type of vulnerability this thesis was aiming to cover, the approach was developed for a severely outdated version of the system or was too similar to one of the existing approaches.

After going over the vulnerabilities and existing solutions, ICC, cache file leaks, packed application leaks, third-party library leaks and inter-app leaks were selected as the vulnerabilities which the solution developed in this thesis will try to mitigate.

For every analyzed solution, their limitations are discussed and these limitations are then turned into selection criteria. The goal was to find a solution which would fulfill the selection criteria and serve as a basis for further improvement. The proof-of-concept solution would then improve on the inherent limitations of the selected approach. For example, some solutions only detected one vulnerability, while others detected multiple, so for the basis of the proof-of-concept solution, only existing solutions which detected multiple vulnerabilities are considered.

During the research, it could be seen that most of the showcased existing solutions used some sort of a **static** or **dynamic** analysis algorithm. A dynamic analysis solution was not developed as it was seen as time-intensive for development with regards to overall results it can provide. Some of the presented solutions used the IFDS algorithm as the basis for their static code analysis. This was used in the proposed proof-of-concept solution as well.

During the research of existing solutions, four solutions were selected as potential basis for improvement as they fulfilled the selection criteria. Of those four, upon further analysis, it was seen that two approaches could be used together: **DroidRA**[47] and **FlowDroid**[15]. FlowDroid is a static analysis approach that completely skipped analyzing reflective calls, while DroidRA was made to resolve reflective calls and translate them into normal calls.

One approach, DroidRista[13], already used the strategy of combining *FlowDroid* and *DroidRA*, however, the proof-of-concept solution uses a simplified approach without any extra steps aside from enhancing FlowDroid with DroidRA boosting and the approach produced different results. FlowDroid itself has introduced a way to resolve reflective calls on its own without external tools, since it has been in constant development since 2012, which reduced the overall impact of DroidRA on the data leak detection capabilities of FlowDroid.

The results shown in Chapter 6 show that while it is too late for this approach to make an impact on FlowDroid's data leak detection capabilities, it does introduce some benefits in performance improvements, such as a 10-15% reduction in memory consumption and 5% in run time when running DroidBench tests. However in real-world applications, these differences between the analysis run-times and resource use were smaller.

The results of this approach were only compared to base FlowDroid results and DroidRista results since these are the most similar approaches and can be used as a baseline. They have also shown findings when it comes to reported results of DroidRista. Since we do not know the exact settings of their analysis and the state of FlowDroid at the time, a direct comparison of the results achieved in this thesis and the results presented in DroidRista[13] is not optimal. One thing that differs is that DroidRista reported that FlowDroid failed to detect leaks in certain test cases, while in the evaluation in this thesis, FlowDroid detected those leaks. It is not known whether FlowDroid improved in that regard since the DroidRista article was published, or Alzaidi et al.[13] misreported the results.

Overall, the proof-of-concept solution does an effective job of mitigating all the targeted vulnerabilities and also covers some of the obfuscation techniques like dynamic code loading. However, those results are mostly due to FlowDroid itself and not the proof-of-concept solution, thus this thesis overall did not achieve the targeted improvement of static analysis approaches. The final verdict is that there is no need for a new static analysis data leak detection tool when FlowDroid is available.

The analysis of real-world applications has shown that although this solution deals with obfuscation to some degree, some applications employ it to a degree where the analysis breaks and fails to complete. This means that more effort needs to be put into expanding de-obfuscation techniques to improve real-world analysis.

The solution presented here is usable for Android users since the steps for recreating the proof-of-concept solution are described in Section 5.2. For future work, more work needs to be invested into making a readily available, open-source tool for *hybrid* (static & dynamic) application analysis. No approach presented in Section 3.5 has a public repository and a working version available for the public. An approach which analyzes APK files as they are being installed on a device would be the optimal solution for most users. That said, simply porting a solution like FlowDroid into an Android application or a plug-in would not work, as the tool would likely take 30-60 minutes to analyze an application with the computing power of a mobile CPU.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] *DroidBox: An Android Application Sandbox for Dynamic Analysis.* URL https://github.com/pjlantz/droidbox. Accessed: 21.09.2020.

[2] *ICC-bench: benchmark apps for static analyzing intercomponent data leakage problem of android apps.* URL https://github.com/fgwei/ICC-Bench/. Accessed: 26.01.2021.

[3] *Platform Architecture.* URL https://developer.android.com/guide/platform. Accessed: 19.03.2020.

[4] *CVE Details (Google - Android).* URL https://www.cvedetails.com/product/19997/Google-Android.html?vendor_id=1224. Accessed: 26.01.2021.

[5] *GATOR: Program Analysis Toolkit For Android.* URL http://web.cse.ohio-state.edu/presto/software/gator/. Accessed: 14.06.2020.

[6] *GDPR Data protection by design and by default.* URL https://gdpr-info.eu/art-25-gdpr/.

[7] *Heros IFDS/IDE Solver.* URL https://github.com/Sable/heros. Accessed: 11.09.2020.

[8] *Secure an Android Device.* URL https://source.android.com/security. Accessed: 19.03.2020.

[9] *Mobile Operating System Market Share Worldwide.* URL https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed: 26.01.2021.

[10] *XPrivacyLua.* URL https://github.com/M66B/XPrivacyLua. Accessed: 11.09.2020.

[11] Mohannad Alhanahnah, Qiben Yan, Hamid Bagheri, Hao Zhou, Yutaka Tsutano, Witawas Srisa-an, and Xiapu Luo. Detecting vulnerable android inter-app communication in dynamically loaded code. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE, apr 2019. doi: 10.1109/infocom.2019.8737637.

[12] Irwin Altman. Privacy: A conceptual analysis. *Environment and Behavior*, 8(1): 141–141, mar 1976. doi: 10.1177/001391657600800108.

[13] Areej Alzaidi, Suhair Alshehri, and Seyed M. Buhari. DroidRista: a highly precise static data flow analysis framework for android applications. *International Journal of Information Security*, oct 2019. doi: 10.1007/s10207-019-00471-w.

[14] Steven Arzt, Siegfried Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. 2013. URL `https://github.com/secure-software-engineering/SuSi`.

[15] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*. ACM Press, 2013. doi: 10.1145/2594291.2594299.

[16] Thomas Ball. The concept of dynamic analysis. In *Software Engineering — ESEC/FSE '99*, pages 216–234. Springer Berlin Heidelberg, 1999. doi: 10.1007/3-540-48166-4_14.

[17] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. *CoRR*, abs/1205.3576, 2012. URL `http://arxiv.org/abs/1205.3576`.

[18] Shikhar Bhatnagar, Yasir Malik, and Sergey Butakov. Analysing data security requirements of android mobile banking application. In *Lecture Notes in Computer Science*, pages 30–37. Springer International Publishing, 2018. doi: 10.1007/978-3-030-03712-3_3.

[19] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. Collusive data leak and more. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. ACM Press, 2017. doi: 10.1145/3052973.3053004.

[20] Ann Cavoukian, Scott Taylor, and Martin E. Abrams. Privacy by design: essential for organizational accountability and strong business practices. *Identity in the Information Society*, 3(2):405–413, jun 2010. doi: 10.1007/s12394-010-0053-z.

[21] Hongyi Chen, Ho fung Leung, Biao Han, and Jinshu Su. Automatic privacy leakage detection for massive android apps via a novel hybrid approach. In *2017 IEEE International Conference on Communications (ICC)*. IEEE, may 2017. doi: 10.1109/icc.2017.7996335.

[22] Long Cheng, Fang Liu, and Danfeng Daphne Yao. Enterprise data breach: causes, challenges, prevention, and future directions. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(5):e1211, jun 2017. doi: 10.1002/widm.1211.

[23] Hyunji Chung, Michaela Iorga, Jeffrey Voas, and Sangjin Lee. Alexa, can i trust you? *Computer*, 50(9):100–104, 2017. doi: 10.1109/mc.2017.3571053.

[24] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Krügel, and Giovanni Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *NDSS*, 2017. URL `https://www.semanticscholar.org/paper/Obfuscation-Resilient-Privacy-Leak-Detection-for-Continella-Fratantonio/482e01ba5d29de96842c3e3daebcbad29945e4c0`.

[25] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17 (2):998–1022, 2015. doi: 10.1109/comst.2014.2386139.

[26] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013. URL `http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf`.

[27] Jun Gao, Li Li, Pingfan Kong, Tegawende F. Bissyande, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, pages 1–19, 2019. doi: 10.1109/tr.2019.2956690.

[28] Yuming Ge, Bo Deng, Yi Sun, Libo Tang, Dajiang Sheng, Yantao Zhao, Gaogang Xie, and Kave Salamatian. A comprehensive investigation of user privacy leakage to android applications. *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*, August 2016. doi: 10.1109/ICCCN.2016.7568475. URL `http://ieeexplore.ieee.org/document/7568475/`.

[29] *Permissions on Android*. Google LLC. URL `https://developer.android.com/guide/topics/permissions/overview`. Accessed: 26.01.2021.

[30] Jingjing Gu, Ruicong Huang, Li Jiang, Gongzhe Qiao, Xiaojiang Du, and Mohsen Guizani. A fog computing solution for context-based privacy leakage detection for android healthcare devices. *Sensors*, 19(5):1184, mar 2019. doi: 10.3390/s19051184.

[31] Victor Guana, Fabio Rocha, Abram Hindle, and Eleni Stroulia. Do the stars align? multidimensional analysis of android's layered architecture. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, jun 2012. doi: 10.1109/msr.2012.6224269.

[32] Anisa Halimi and Erman Ayday. Profile matching across unstructured online social networks: Threats and countermeasures, 2017. URL `https://arxiv.org/abs/1711.01815`.

[33] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update, 2009.

[34] Yongzhong He, Binghui Hu, and Zhen Han. Dynamic privacy leakage analysis of android third-party libraries. In *2018 1st International Conference on Data Intelligence and Security (ICDIS)*. IEEE, April 2018. doi: 10.1109/icdis.2018.00051.

[35] Jaap-Henk Hoepman. Privacy design strategies. In *ICT Systems Security and Privacy Protection*, pages 446–459. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-642-55415-5_38.

[36] Jim Isaak and Mina J. Hanna. User data privacy: Facebook, cambridge analytica, and privacy protection. *Computer*, 51(8):56–59, aug 2018. doi: 10.1109/mc.2018.3191268.

[37] Katsutaka Ito, Hirokazu Hasegawa, Yukiko Yamaguchi, and Hajime Shimada. Detecting privacy information abuse by android apps from API call logs. In *Advances in Information and Computer Security*, pages 143–157. Springer International Publishing, 2018. doi: 10.1007/978-3-319-97916-8_10.

[38] Vineeta Jain, Shweta Bhandari, Vijay Laxmi, Manoj Singh Gaur, and Mohamed Mosbah. SniffDroid: Detection of inter-app privacy leaks in android. In *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, aug 2017. doi: 10.1109/trustcom/bigdatase/icess.2017.255.

[39] Vineeta Jain, Vijay Laxmi, Manoj Singh Gaur, and Mohamed Mosbah. APPLADroid: Automaton based inter-app privacy leak analysis for android. In *Communications in Computer and Information Science*, pages 219–233. Springer Singapore, 2019. doi: 10.1007/978-981-13-7561-3_16.

[40] Volker Klingspor. Why do we need data privacy? In *Solving Large Scale Learning Tasks. Challenges and Algorithms*, pages 85–95. Springer International Publishing, 2016. doi: 10.1007/978-3-319-41706-6_3.

[41] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011. URL `https://sable.github.io/soot/resources/lblh11soot.pdf`.

[42] Er-Rajy Latifa and El Kiram My Ahmed. Android: Deep look into dalvik VM. In *2015 5th World Congress on Information and Communication Technologies (WICT)*. IEEE, dec 2015. doi: 10.1109/wict.2015.7489641.

[43] Michael Lettner, Michael Tschernuth, and Rene Mayrhofer. Mobile platform architecture review: Android, iPhone, qt. In *Computer Aided Systems Theory – EUROCAST 2011*, pages 544–551. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-27579-1_70.

[44] Hui Li, Wenling Liu, Bin Wang, and Wen Zhang. Detection and auto-protection of cache file privacy leakage for mobile social networking applications in android. In *Human Aspects of Information Security, Privacy and Trust*, pages 703–721. Springer International Publishing, 2017. doi: 10.1007/978-3-319-58460-7_48.

[45] L. Li. Boosting static analysis of android apps through code instrumentation. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 819–822, 2016.

[46] Li Li, Alexandre Bartel, Tegawende F. Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. IccTA: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015. doi: 10.1109/icse.2015.48.

[47] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. DroidRA: taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, 2016. doi: 10.1145/2931037.2931044.

[48] Hongliang Liang, Yudong Wang, Tianqi Yang, and Yue Yu. AppLance: A lightweight approach to detect privacy leak for packed applications. In *Secure IT Systems*, pages 54–70. Springer International Publishing, 2018. doi: 10.1007/978-3-030-03638-6_4.

[49] Hongliang Liang, Tianqi Yang, Lin Jiang, Yixiu Chen, and Zhuosi Xie. Witness: Detecting vulnerabilities in android apps extensively and verifiably. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, dec 2019. doi: 10.1109/apsec48747.2019.00065.

[50] Mario Linares-Vasquez, Gabriele Bavota, and Camilo Escobar-Velasquez. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, may 2017. doi: 10.1109/msr.2017.60.

[51] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *IEEE Transactions on Mobile Computing*, pages 1–1, 2019. doi: 10.1109/tmc.2019.2903186.

[52] P. Louridas. Static code analysis. *IEEE Software*, 23(4):58–61, jul 2006. doi: 10.1109/ms.2006.114.

[53] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android OS stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24(4):2056–2101, feb 2019. doi: 10.1007/s10664-019-09689-7.

[54] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the IFDS algorithm. In *Lecture Notes in Computer Science*, pages 124–144. Springer Berlin Heidelberg, 2010. doi: 10.1007/978-3-642-11970-5_8.

[55] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015. doi: 10.1109/icse.2015.30.

[56] Miriam Quick, Ella Hollowood, Christian Miles, Dan Hampson, and Duncan Geere. World's biggest data breaches & hacks. Technical report, 2020. URL `https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/`.

[57] Joel Reardon, Álvaro Feal, Primal Wijesekera, Amit Elazari Bar On, Narseo Vallina-Rodriguez, and Serge Egelman. 50 ways to leak your data: An exploration of apps' circumvention of the android permissions system. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 603–620, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/reardon`.

[58] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 361–374, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342698. doi: 10.1145/2906388.2906392. URL `10.1145/2906388.2906392`.

[59] Thomas Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, page 5–19, Cambridge, MA, USA, 1997. MIT Press. ISBN 0262631806.

[60] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. ACM Press, 1995. doi: 10.1145/199448.199462.

[61] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. ISBN 0897912527. doi: 10.1145/73560.73562. URL `https://doi.org/10.1145/73560.73562`.

[62] Sarker T. Ahmed Rumee, Donggang Liu, and Yu Lei. MirrorDroid: A framework to detect sensitive information leakage in android by duplicate program execution. In *2017 51st Annual Conference on Information Sciences and Systems (CISS)*. IEEE, March 2017. doi: 10.1109/ciss.2017.7926086.

[63] Alan Said and Vicenç Torra, editors. *Data Science in Practice*. Springer International Publishing, 2019. doi: 10.1007/978-3-319-97556-6.

[64] Asaf Shabtai, Yuval Elovici, and Lior Rokach. *A Survey of Data Leakage Detection and Prevention Solutions*. Springer US, 2012. doi: 10.1007/978-1-4614-2053-8.

[65] Animesh Srivastava, Puneet Jain, Soteris Demetriou, Landon P. Cox, and Kyu-Han Kim. CamForensics. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems - SenSys '17*. ACM Press, 2017. doi: 10.1145/3131672.3131683.

[66] Sabine Trepte. Privatsphäre aus psychologischer sicht. In *Datenschutz - Grundlagen, Entwicklungen und Kontroversen*, pages 59–66. Bundeszentrale für politische Bildung, 2012.

[67] Omer Tripp and Julia Rubin. A bayesian approach to privacy enforcement in smartphones. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 175–190, USA, 2014. USENIX Association. ISBN 9781931971157.

[68] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-An, Gregg Rothermel, and Jackson Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, may 2017. doi: 10.1109/icse.2017.37.

[69] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999. doi: 10.5555/781995.782008.

[70] Chao Wang, Wei Duan, Jianzhang Ma, and Chenhui Wang. The research of android system architecture and application programming. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*. IEEE, dec 2011. doi: 10.1109/iccsnt.2011.6182081.

[71] Wiktoria Wilkowska and Martina Ziefle. Privacy and data security in e-health: Requirements from the user's perspective. *Health Informatics Journal*, 18(3):191–201, sep 2012. doi: 10.1177/1460458212442933.

[72] Daoyuan Wu, Debin Gao, Eric K. T. Cheng, Yichen Cao, Jintao Jiang, and Robert H. Deng. Towards understanding android system vulnerabilities. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security - Asia CCS '19*. ACM Press, 2019. doi: 10.1145/3321705.3329831.

[73] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, may 2015. doi: 10.1109/icse.2015.31.

[74] Zipeng Zhang and Xinyu Feng. AndroidLeaker: A hybrid checker for collusive leak in android applications. In *Dependable Software Engineering. Theories, Tools, and Applications*, pages 164–180. Springer International Publishing, 2017. doi: 10.1007/978-3-319-69483-2_10.