NORTHWESTERN UNIVERSITY

Towards the Privacy Leakage of Android Applications

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Zhengyang Qu

EVANSTON, ILLINOIS

September 2017

ABSTRACT

Towards the Privacy Leakage of Android Applications

Zhengyang Qu

Smartphone is becoming ubiquitous and its sales proportions have exceeded the sales of personal computer systems since 2012. The number of smartphones will increase and perhaps at an even higher rate in the coming years. The computational capacity and numerous mobile applications benefit end user's daily life. At the same time, it stores user's personal information, such as calendar event, photo, geo-location, and manages the access to the private online resource, such as bank account, email. It is thus non-trivial to resolve the security risks of smartphone privacy leakage.

The open nature allows Android to capture a dominant share of mobile operating system market. However, the open nature challenges the protection of user privacy given those platform-driven factors: (1) unregulated mobile marketplaces, (2) Android middleware with the APIs creating unpredictable runtime behavior, (3) fragmentation. Moreover, the device usage by unauthorized users produces the risk of privacy leakage driven by the human. We claim that a comprehensive solution to the privacy leakage of Android platform needs to overcome the challenges incurred by the platform-driven factors and

the human-driven factor. Four separate works are discussed: (a) AutoCog relies on a learning-based approach to deduce the semantics model and helps the user understand the in-app privacy usage by the application description; (b) DyDroid is a dynamic analysis system to fully explore the DCL usage and detect the privacy leakage; (c) AppShield allows the enforcement of arbitrary access control policy with an application rewriting design; and (d) RiskCog enforces the continuous and implicit user authentication by the manner of handling the device.

Acknowledgments

I would like to express my sincere and deepest appreciation to my academic advisor and committee chair, Professor Yan Chen. who spends numerous hours with me on the research projects, allows me to learn how to do research, and lead me through this journey of Ph.D. His guidance and support make all these happen.

I would like to thank my committee members, Professor Douglas Downey from Northwestern University, Professor Simone Campanoni from Northwestern University, and Professor Hao Chen from the University of California. Davis. Professor Hao Chen also gives me a wealth of comments for my AppShield work. I would also like to thank Professor V.N. Venkatakrishnan from the University of Illinois, Chicago and Professor Aleksandar Kuzmanovic from Northwestern University for giving valuable feedback during the Ph.D. exams.

My gratitude also goes to my collaborators, Professor Ryan Riley, Dr. Xiaoyong Zhou, and Dr. Shahid Alam, who offer valuable help to my studies DyDroid and DroidNative. Moreover, I would like to express a thank you to Dr. Michael Grace, who are my internship mentors at Samsung Research America.

The journey of Ph.D. will be very challenging without my friends and colleagues, Professor Yinzhi Cao, Hongyu Gao, Vaibhav Rastogi, Xitao Wen, Xiang Pan, Haitao Xu, Yang Hu, Libing Song, Xutong Chen, Wangjun Hong. Their help during my stay

at Northwestern University is highly appreciated. Last but not least, I would especially thank my family for the unconditional love and support.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Smartphone is becoming ubiquitous and its sales proportions have exceeded the sales of personal computer systems since 2012. The number of smartphones will increase and perhaps at an even higher rate in the coming years. The computational capacity and numerous mobile applications benefit end user's daily life. At the same time, it stores user's personal information, such as calendar event, photo, geo-location, and manages the access to the private online resource, such as bank account, email. It is thus non-trivial to resolve the security risks of smartphone privacy leakage.

The ecosystem of smartphone includes marketplaces, end users, developers, and devices. The users download mobile apps that are uploaded to marketplaces by the developers. The rich functionalities of apps depend on the invocation of Android framework APIs written in Java, which will further call the low-level native code written in C/C++. Given the various parties with different purposes of interest involved in smartphone ecosystem, it is challenging to enforce the trustworthiness among them.

The open nature allows Android to capture a dominant share of mobile operating system market. However, the open nature challenges the protection of user privacy given the following platform-driven factors.

- **Mobile marketplace.** Apart from the official Android marketplace Google Play [**73**], there are bunches of third-party mobile application markets built by

device vendors. There is no existing standard to regulate the user privacy-related application metadata provided by the developer. The end user lacks the source to understand how her/his privacy is used by the mobile application vendor.

- **Android middleware.** Android app is allowed to execute external binaries with dynamic code loading (DCL) APIs. This feature makes the application's behavior at runtime unpredictable.
- **Fragmentation.** Each Android device vendor can customize the open-sourced OS and even the low-level hardware. Although there are several security patches or toolbox depending on OS such as Samsung Knox [**137**], those solutions can hardly be deployed widely.

Smartphone is also heavily used to access user's sensitive resource online. The device usage by unauthorized users, e.g, stolen device, produces the risk of privacy leakage driven by the human. The traditional credential-based user authentication only verifies if the user knows the predefined credential, which is easy to get bypassed. The explicit input of password has the tradeoff between its usability and the continuity of protection.

The overview of this thesis is illustrated in Figure 1.1. Four separate works are discussed to comprehensively resolve the privacy leakage incurred by platform-driven and human-driven factors.

- AutoCog relies on a learning-based approach to deduce the semantics model and helps the user understand the in-app privacy usage by the application description.
- DyDroid is a dynamic analysis system to fully explore the DCL usage and detect the privacy leakage.

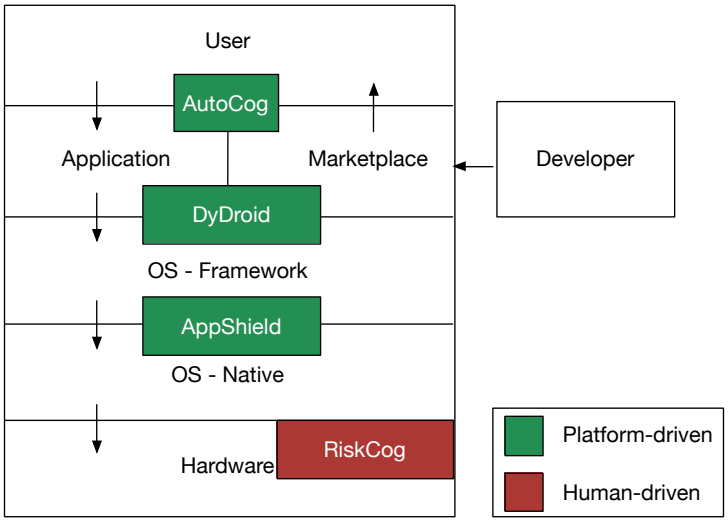


Figure 1.1. Thesis overview

- AppShield allows the enforcement of arbitrary access control policy with an application rewriting design.
- RiskCog enforces the continuous and implicit user authentication by the manner of handling the device.

Specifically, we make the following thesis statement:

A comprehensive solution to the privacy leakage of Android platform needs to overcome the challenges incurred by the platform-driven factors: marketplace, middleware, fragmentation and the human-driven factor.

1.1. AutoCog

The booming popularity of smartphones is partly a result of application markets where users can easily download wide range of third-party applications. However, due to the open nature of markets, especially on Android, there have been several privacy and security concerns with these applications. On Google Play, as with most other markets, users

have direct access to natural-language descriptions of those applications, which give an intuitive idea of the functionality including the security-related information of those applications. Google Play also provides the permissions requested by applications to access security and privacy-sensitive APIs on the devices. Users may use such a list to evaluate the risks of using these applications. To best assist the end users, the descriptions should reflect the need for permissions, which we term *description-to-permission fidelity*. We present a system AutoCog to automatically assess description-to-permission fidelity of applications. AutoCog employs state-of-the-art techniques in natural language processing and our own learning-based algorithm to relate description with permissions. In our evaluation, AutoCog outperforms other related work on both performance of detection and ability of generalization over various permissions by a large extent. On an evaluation of eleven permissions, we achieve an average precision of 92.6% and an average recall of 92.0%. Our large-scale measurements over 45,811 applications demonstrate the severity of the problem of low description-to-permission fidelity. AutoCog helps bridge the long-lasting usability gap between security techniques and average users.

1.2. DyDroid

Android has provided DCL since API level one. DCL allows an app developer to load additional code into an application at runtime. DCL raises numerous challenges with regards to security and accountability analysis of apps. While previous studies have investigated DCL on Android, we formulate and answer three critical questions that are missing from previous studies: (1) Where does the loaded code come from (remotely fetched or locally packaged), and who is the responsible entity to invoke its functionality?

(2) In what ways is DCL utilized to harden mobile apps, specifically, application obfuscation? (3) What are the security risks and implications that can be found from DCL in off-the-shelf apps?

We design and implement DyDroid, a system which uses both dynamic and static analysis to analyze dynamically loaded code. Dynamic analysis is used to automatically exercise apps, capture DCL behavior, and intercept the loaded code. Static analysis is used to investigate malicious behavior and privacy leakage in that dynamically loaded code. We have used DyDroid to analyze over 46K apps with little manual intervention, allowing us to conduct a large-scale measurement to investigate five aspects of DCL, such as source identification, malware detection, vulnerability analysis, obfuscation analysis, and privacy tracking analysis.

We have several interesting findings. (1) 27 apps are found to violate the content policy of Google Play by executing code downloaded from remote servers. (2) We determine the distribution, pros/cons, and implications of several common obfuscation methods, including DEX encryption/loading. (3) DCL's stealthiness enables it to be a channel to deploy malware, and we find 87 apps loading malicious binaries which are not detected by existing antivirus tools. (4) We found 14 apps that are vulnerable to code injection attacks due to dynamically loading code which is writable by other apps. (5) DCL is mainly used by third-party SDKs, meaning that app developers may not know what sort of sensitive functionality is injected into their apps.

1.3. AppShield

Bring-your-own-device (BYOD) is getting popular. Diverse personal devices are used to access enterprise resources, and deployment of the solutions with customized operating system (OS) dependency will thus be restricted. Moreover, device utilization for both business and personal purposes creates new threats involving leakage of sensitive data. As for functionalities, a BYOD solution should isolate an arbitrary number of entities, such as those relating to business and personal uses and provide fine-grained access control on multi-entity management. Existing BYOD solutions lack in these aspects; we propose a system, called AppShield, which supports multi-entity management and role-based access control with file-level granularity, apart from local data sharing/isolation. AppShield includes (1) application rewriting framework for Android apps, which builds Mobile Application Management (MAM) features into app automatically with complete mediation, (2) cross-platform proxy-based data access mechanism, which can enforce arbitrary access control policies. The fully functional controller with data proxy is implemented for both Android and iOS. AppShield allows for enterprise policy management without modifying device OS. The evaluation shows that AppShield is successful at policy enforcement and is reliable. It induces little impact on application's performance and size, for example, our app rewriting introduces less than 5% code size increment in over 95% apps in our evaluation.

1.4. RiskCog

Mobile payment is becoming popular. Integrating the sensitive payment functionality to the smartphone introduces new security risks, for example, the attacker pays with the

victim's account using the device stolen. By depicting the device owner with a diverse set of features, such as the face snapshot, the existing user identification is harder to get bypassed than the traditional user authentication mechanism with the simple account credential. However, it involves the heavy usage of user's personal information. Moreover, the current user identification countermeasure deploys at the application level, where each mobile payment service vendor has its channel of data collection individually. They cannot share their data, making it impossible to reuse the detection results for other apps.

We propose the system RiskCog, which solves the problem of identifying the authorized device owner by the data collected from the motion sensors with a learning-based approach. Our feature set only leverages the motion sensors, which are commonly available on smartphones and have low privacy sensitivity in the context of social impact. RiskCog is designed as a third-party service at the device level that requires no developer support. Our feature set is independent of a user's motion state and has no requirement of user movement or fixed device placement. Moreover, we resolve the issues of the imbalanced dataset with our stratified sampling and missing of ground truth with a semi-supervised learning algorithm. Along with the design of offline verification, our system can protect the user in any challenging scenario, even in the industry product. For the data collected from 1,513 users, RiskCog identifies the authorized owner in steady/moving states with the accuracy values 93.77% and 95.57%.

1.5. Organization

The thesis is organized as follows. Chapter 2 describes AutoCog. Chapter 3 presents DyDroid and the measurement results regarding DCL. Chapter 4 presents AppShield while RiskCog is introduced in Chapter 5. Finally, Chapter 6 concludes this thesis.

CHAPTER 2

AutoCog: Measuring the Description-to-permission Fidelity in Android Applications

2.1. Introduction

Modern operating systems such as Android have promoted global ecosystems centered around large repositories or marketplaces of applications. Success of these platforms may in part be attributed to these marketplaces. Besides serving applications themselves, these marketplaces also host application metadata, such as descriptions, screenshots, ratings, reviews, and, in case of Android, permissions requested by the application, to assist users in making an informed decision before installing and using the applications. From the security perspective, applications may access users' private information and perform security-sensitive operations on the devices. With the application developers having no obvious trust relationships with the user, these metadata may help the users evaluate the risks in running these applications.

It is however generally known [**63**] that few users are discreet enough or have the professional knowledge to understand the security implications that may be derived from metadata. On Google Play, users are shown both the application descriptions and the permissions[1] declared by applications. An application's description describes the functionality of an application and should give an idea about the permissions that would be

[1]In Android, security-sensitive system APIs are guarded by permissions, which applications have to declare and which have to be approved at install-time.

requested by that application. We call this *description-to-permission fidelity*. For example, an application that describes itself as a social networking application will likely need permissions related to device's address book. A number of malware and privacy-invasive applications have been known to declare more permissions than their purported functionality warrants [**59, 166**].

With this belief that descriptions and permissions should generally correspond, we present AutoCog, a system that automatically identifies if the permissions declared by an application are consistent with its description. AutoCog has multi-fold uses.

- Application developers can use this tool to receive an early, automatic feedback on the quality of descriptions so that they improve the descriptions to better reflect the security-related aspects of the applications.
- End users may use this system to understand if an application is over-privileged and risky to use.
- Application markets can deploy this tool to bolster their overall trustworthiness.

The key challenge is to gather enough semantics from descriptions in natural language to reason about the permissions declared. We apply state-of-the-art techniques from natural language processing (NLP) for sentence structure analysis and computing semantic relatedness of natural language texts. We further develop our own learning-based algorithm to automatically derive a model that can be queried against with descriptions to get the expected permissions.

AutoCog is a substantial advancement over the previous state-of-the-art technique by Pandita et al. [**118**], who have also attempted to develop solutions with the same goals. Their tool called Whyper is primarily limited by the use of a fixed vocabulary

derived from the platforms' API documents and the English synonyms of keywords there. Our investigations show that WHYPER's methodology is inherently limited regarding the following issues: (*a*) *Limited semantic information*: not all textual patterns associated with a permission can be extracted from API documents, e.g., <"*find*", "*branch atm*"> relate to location permissions and <"*scan*", "*barcode*"> relate to the permission for accessing the camera in our models but cannot conceivably be found from API documents; (*b*) *Lack of associated APIs*: certain permissions do not have associated APIs so that this methodology cannot be used; and (*c*) *Lack of automation*: it is not clear how the techniques could be automated. We have confirmed these limitations with WHYPER's authors as well.

Our methodology is radically different from WHYPER's as is evident from the following contributions.

- *Relating descriptions and permissions.* We design a novel learning-based algorithm for modeling the relatedness of descriptions to permissions. Our algorithm correlates textual semantic entities (second contribution) to the declared permissions. It is noteworthy that the model is trained entirely from application descriptions and declared permissions over a large set of applications without depending on external data such as API documents, so that we do not have the problems of limited semantic information or lack of associated APIs from the very outset. Both training and classification are completely automatic.
- *Extracting semantics from descriptions.* We utilize state-of-the-art NLP techniques to automatically extract semantic information from descriptions. The key component for semantics-extraction in our design is Explicit Semantic Analysis

(ESA), which leverages big corpuses like Wikipedia to create a large-scale semantics database, and which has been shown to be superior to dictionary-based synonyms and other methods [**69**] and is being increasingly adopted by numerous research and commercial endeavors. Such superior analysis further largely mitigates the problem of limited semantic information.

- *System prototype.* We design and implement an end-to-end tool called AutoCog to automatically extract relevant semantics from Android application descriptions and permissions to produce permission models. These models are used to measure description-to-permission fidelity: given an application description, a permission model outputs whether the permission is expected to be declared by that application. If the answer is yes, AutoCog further provides relevant parts of description that warrant the permission. This tool is published on Google Play[2] and the backend data is available on our web portal[3].

We further have the following evaluation and measurement highlights.

- *Evaluation.* Our evaluation on a set of 1,785 applications shows that AutoCog outperforms the previous work on detection performance and ability of generalization over various permissions by a large extent. AutoCog closely aligns with human readers in inferring the evaluated permissions from textual descriptions with an average precision of 92.6% and average recall of 92.0% as opposed to previous state-of-the-art precision and recall of 85.5% and 66.5% respectively.
- *Measurements.* Our findings on 45,811 applications using AutoCog show that the description-to-permissions fidelity is generally low on Google Play with only

[2] `https://play.google.com/store/apps/details?id=com.version1.autocog`
[3] `http://python-autocog.rhcloud.com`

9.1% of applications having permissions that can all be inferred from the descriptions. Moreover, we observe the negative correlation between fidelity and application popularity.

The remainder of this section is organized as follows. Section 2.2 gives further motivation of our work and presents a brief background and problem statement. Next we cover AutoCog design in detail in Section 2.3, followed by the implementation aspects in Section 2.4. Section 2.5 deals with the evaluation of AutoCog and introduces our measurement results. We have relevant discussion and related work in Sections 2.6 and 2.7. Finally, we conclude our work in Section 2.8.

2.2. Background and Problem statement

2.2.1. Background

Android is the most popular smartphone operating system with over 80% market share [**14**]. It introduces a sophisticated permission-based security model, whereby an application declares a list of permissions, which must be approved by the user at application installation. These permissions guard specific functionalities on the device, including some security and privacy-sensitive APIs such as access contacts.

Modern operating systems such as Android, iOS, and Windows 8 have brought about the advent of big, centralized application stores that host third-party applications for users to view and install. Google Play, the official application store for Android, hosts both free and paid applications together with a variety of metadata including the title and description, reviews, ratings, and so on. Additionally, it also provides the user with the ability to study the permissions requested by an application.

2.2.2. Problem Statement

The application descriptions on Google Play are a means for the developers to communicate the application functionality to the users. From the security and privacy standpoint, these descriptions should thus indicate the reasons for the permissions requested by an application, either explicitly or implicitly[4]. We call it *fidelity* of descriptions to permissions.

As stated in Section 2.1, Android applications often have little in their descriptions to indicate to the users why they need the permissions declared. Specifically, there is

[4]By implicit, we mean that the need for permission is evident from stated functionality.

frequently a gap between the access of the sensitive device APIs by the applications and their stated functionality. This may not always be out of malicious intent; however users are known to be concerned about the use of sensitive permissions [**61**]. Moreover, Felt et al. [**63**] show that few users are careful enough or able to understand the security implications derived from the metadata. In this work we thus look into the problem of *automatically assessing the fidelity of the application descriptions with respect to the permissions.*

Detection of malicious smartphone applications is possible through static/run-time analysis of binaries [**57, 83, 158**]. However, the techniques to evaluate whether application oversteps the user expectation are still lacking. Our tool can assist the users and other entities in the Android ecosystem assess whether the descriptions are faithful to the permissions requested. AutoCog may be used by users or developers individually or deployed at application markets such as Google Play. It may automatically alert the end users if an application requests more permissions than required for the stated functionalities. The tool can provide useful feedback about the shortcomings of the descriptions to the developers and further help bolster the overall trustworthiness of the mobile ecosystem by being deployed at the markets.

As for automatically measuring description-to-permission fidelity, we need to deal with two concepts: (*a*) the *description semantics*, which relates to the meaning of the description, and (*b*) the *permission semantics*, which relates to the functionality provided (or protected) by the permission. The challenges in solving our problem therefore lie in:

- *Inferring description semantics*: Same meaning may be conveyed in a vast diversity of natural language text. For example, the noun phrases "*contact list*", "*address book*", and "*friends*" share similar semantic meaning.
- *Correlating description semantics with permission semantics*: A number of functionalities described may map to the same permission. For example, the permission to access user location might be expressed with the texts "*enable navigation*", "*display map*", and "*find restaurant nearby*". The need for permission to write to external disk can be implied as "*save photo*" or "*download ringtone*".

In AutoCog, we consider the decision version of the problem stated above: *given a description and a permission, does the description warrant the declaration of the permission?* If AutoCog answers yes, it provides the sentences that warrant the permission, thus assisting users in reasoning about the requested permission. As a complete system, AutoCog solves this decision problem for each permission declared.

Whyper [**118**] is a previous work with goals similar to ours. Whyper correlates the description and permission semantics by extracting natural language keywords from an external source, Android API documents. Since APIs and permissions can be related together [**26**], the intuition is that keywords and patterns expressed in the API documentation will also be found in the application descriptions and are therefore adequate in representing the respective permissions. Based on our investigation, the methodology has the following fundamental limitations:

- *Limited semantic information*: the API documents are limited in the functionality they describe and so Whyper cannot cover a complete set of semantic patterns correlated with permissions. For example, in our findings, the pattern

<"*deposit*", "*check*"> is related to the permission CAMERA with high confidence but cannot be extracted from API documents. The mobile banking applications, such as Bank of America[5], support depositing by snapping its photo with the device's camera. Analysis on this issue in detail will be shown in Section 2.5.2.

- *Lack of associated APIs*: certain sensitive permissions such as the permission RECEIVE_BOOT_COMPLETED do not have any associated APIs [**26**]. It is thus not possible to generate the correlated textual pattern set with the API documents.
- *Lack of automation*: WHYPER's extraction of patterns from API documents involved manual selection to preserve the quality of patterns; what policies could be used to automate this process in a systematic manner is an open question.

Our learning-based approach automatically discovers a set of textual patterns correlated with permissions from the descriptions of a rich set of applications, hence enabling our description-to-permission relatedness model to achieve a complete coverage over the natural language texts with great diversity. Besides, the training process works directly on descriptions. So we easily overcome the limitations of the previous work as stated above.

[5] `https://play.google.com/store/apps/details?id=com.infonow.bofa`

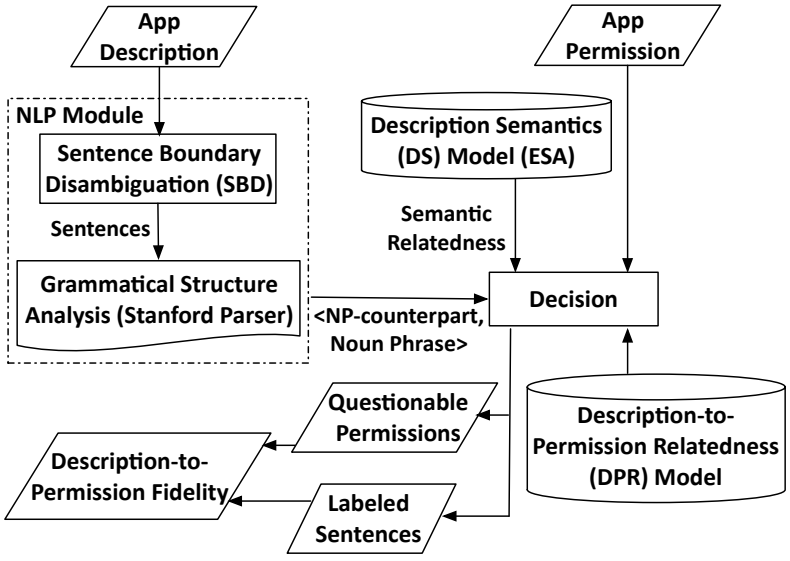


Figure 2.1. Overall architecture of AutoCog

2.3. System Design

Figure 2.1 gives an architectural overview of AutoCog. The description of the application is first processed by the NLP module, which disambiguates sentence boundaries and analyzes each sentence for grammatical structure. The output of the NLP module is then passed in together with the application permissions into the decision module, which, based on models of description semantics and description-to-permission relatedness outputs the questionable permissions that are not warranted from the description and the sentences from which the other permissions may be inferred. These outputs together provide description–to-permission fidelity. This section provides a detailed design of each of the modules and the models that constitute AutoCog.

2.3.1. NLP Module

The goal of the NLP module is to identify specific constructs in the description such as noun and verb phrases and understand relationship among them. Use of such related constructs alleviates the shortcomings of simple keyword-based analysis. The NLP module consists of two components, sentence boundary disambiguation and grammatical structure analysis.

2.3.1.1. Sentence boundary disambiguation (SBD). The whole description is split into sentences for subsequent sentence structure analysis [**92, 134**]. Characters such as ".", ":", "-", and some others like "*", "♠", "♢" that may start bullet points are treated as sentence separators. Regular expressions are used to annotate email addresses, URLs, IP addresses, Phone numbers, decimal numbers, abbreviations, and ellipses, which interfere with SBD as they contain the sentence separator characters.

2.3.1.2. Grammatical structure analysis. We leverage Stanford Parser [**143**] to identify the grammatical structure of sentences. While our design depends on constructs provided by the Stanford Parser, it is conceivable that other NLP parsers could be used as well.

We first use the Stanford Parser to output *typed dependencies*, which are semantic hierarchies of sentences, i.e., how different parts of sentences depend on each other. As illustrated in Figure 2.2, the dependencies are triplets: *name of the relation*, *governor* and *dependent*. *Part of Speech (PoS)* tagging additionally assigns a part-of-speech tag to each word; for example, a verb, a noun, or an adjective. The results are fed into *phrase parsing* provided by Stanford Parser to break sentences into phrases, which could be noun

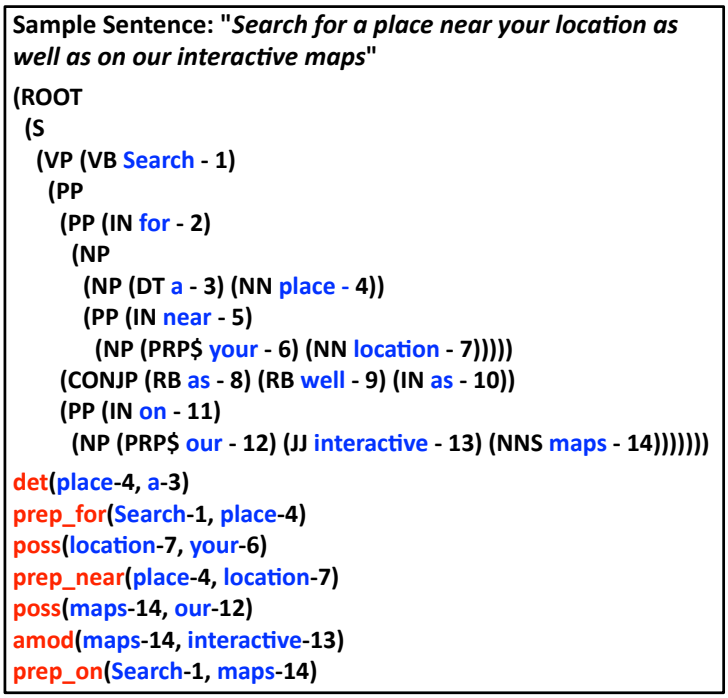


Figure 2.2. Example output of Stanford Parser

phrases, verb phrases or other kinds of phrases. We obtain a hierarchy of marked phrases and tagged words for each sentence.

The governor-dependent pair provides the knowledge of logic relationship between various parts of sentence, which provides the guideline of our ontology modeling. The concept of ontology is a description of things that exist and how they relate to each other. In our experience, we find the following ontologies, which are governor-dependent pairs based on noun phrase, to be most suitable for our purposes.

- Logical dependency between verb phrase and noun phrase potentially implies the actions of applications performing on the system resources. For example, the pairs <"scan", "barcode"> and <"record", "voice"> reveal the use of permissions camera and recording.

- Logical dependency between noun phrases is likely to show the functionalities mapped with permissions. For instance, users may interpret the pairs <"scanner", "barcode"> and <"note", "voice"> as using camera and microphone.
- Noun phrase with own relationship (possessive, such as "your", followed by resource names) is recognized as requesting permissions. For example, the CAMERA and RECORD_AUDIO permissions could be revealed by the pairs <"your", "camera"> and <"own", "voice">.

We extract all the noun phrases in the leaf nodes of the hierarchical tree output from grammatical structure analysis. For each noun phrase, we record all the verb phrases and noun phrases that are its ancestors or siblings of its ancestors. We also record the possessive, if the noun phrase itself contains the own relationship. For the sake of simplicity, we call the extracted verb phrases, noun phrases, and possessives as *np-counterpart* for the target noun phrase. The noun-phrase based governor-dependent pairs obtained signify direct or indirect dependency. The example hierarchy tree for sentence "*Search for a place near your location as well as on our interactive maps*" is shown in Figure 2.2 with the pairs extracted: <"search", "interactive map">, <"our", "interactive map">, <"search", "place">, <"search", "location">, <"place", "location">, and <"your", "location">.

We process these pairs to remove stopwords and named entities. *Stopwords* are common words that cannot provide much semantic information in our context, *e.g.*, "the", "and", "which", and so on. Named entities include names of persons, places, companies, and so on. These also do not communicate security-relevant information in our context. To filter out named entities, we employ *named entity recognition*, a well-researched NLP

topic, also implemented in Stanford Parser. The remaining words are normalized by lowercasing and *lemmatization* [**52**]. Example normalizations include "better" → "good" and "computers" → "computer".

2.3.2. Description Semantics (DS) Model

The goal here is to understand the meaning of a natural language description, i.e., how different words and phrases in a vocabulary relate to each other. Similarly meaning natural language descriptions can differ vastly; so such an analysis is necessary. Our model is constructed using *Explicit Semantic Analysis (ESA)*, the state of the art for computing semantic relatedness of texts [**69**]. The model is used directly by the decision module and also for training the description-to-permission relatedness model discussed in Section 2.3.3.

ESA is an algorithm to measure the semantic relatedness between two pieces of text. It leverages big document corpuses such as Wikipedia as its knowledge base and constructs a vector representation of text. In ESA, each (Wiki) article is called a concept, and transformed into a weighted vector of words within the article. As processing an input article, ESA computes the relatedness of the input to every concept, i.e. projects the input article into the concept space, by the common words between them. In NLP and information retrieval applications, ESA computes the relatedness of two input articles using the cosine distance between the two projected vectors.

We choose ESA because it has been shown to outperform other known algorithms for computing semantic relatedness such as WordNet and latent semantic analysis [**69**]. We offer intuitive reasons of out-performance over WordNet as this has been used in

WHYPER. First, WordNet-based methods are inherently limited to individual words, and adoption for comparing longer text requires an extra level of sophistication [**108**]. Second, considering words in context allows ESA to perform word sense disambiguation. Using WordNet cannot achieve disambiguation, since information about synsets (sets of synonyms) is limited to a few words; while in ESA, concepts are associated with huge amounts of text. Finally, even for individual words, ESA offers a much more detailed and quantitative representation of semantics. It maps the meaning of words/phrases to a weighted combination of concepts, while mapping a word in WordNet amounts to simple lookup, without any weight.

2.3.3. Description-to-Permission Relatedness (DPR) Model

Description-to-permission relatedness (DPR) model is a decisive factor in enhancing the accuracy of AUTOCOG. We design a learning-based algorithm by analyzing the descriptions and permissions of a large dataset of applications to measure how closely a noun-phrase based governor-dependent pair is related to a permission. The flowchart for building the DPR model is shown in Figure 2.3. We first leverage ESA to group the noun phrases with similar semantics. Next, for each permission, we produce a list of noun phrases whose occurrence in descriptions is positively related to the declaration of that permission. Such phrases may potentially reveal the need for the given permission. In the third stage, we further enhance the results by adding in the np-counterparts (of the noun-phrase based governor-dependent pairs) and keeping only the pairs whose occurrence statistically correlates with the declaration of the given permission.

Table 2.1. Distribution of noun phrase patterns

Pattern	#Noun Phrase (Percentage %)
Noun	1,120,850 (52.37 %)
Noun + Noun	414,614(19.37 %)
Adjective + Noun	278,785 (13.03 %)
Total	1,814,249 (84.77 %)

Pattern of noun phrase; Number/percentage of noun phrases in the pattern within 2,140,225 noun phrases extracted from 37,845 applications

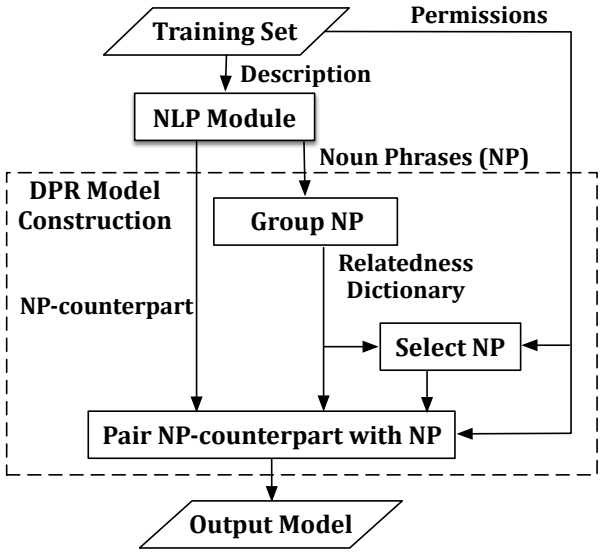


Figure 2.3. Flowchart of description-to-permission relatedness (DPR) model construction

2.3.3.1. Grouping Noun Phrases. A noun phrase contains a noun possibly together with adjectives, adverbs, etc. During the learning phase, since analyzing long phrases is not efficient, we consider phrases of only three patterns: single noun, two nouns, and noun following adjective (Table 2.1). In our dataset of 37,845 applications, these patterns account for 85% of the 302,739 distinct noun phrases. We further note that we focus on these restricted patterns only during DPR model construction; all noun phrases are considered in the decision module of AutoCog, which checks whether the description of application indicates a given permission. The DS model, which is also employed during

decision-making, can match longer patterns with similarly meaning noun phrases grouped here. Hence the negative effect of such simplification is negligible.

We construct a *semantic relatedness score matrix* leveraging DS model with ESA. Each cell in the matrix depicts the semantic relatedness score between a pair of noun phrases. Define the *frequency* of noun phrase to be the number of applications whose descriptions contain the noun phrase. As constructing the semantic relatedness score matrix has quadratic runtime, it is not scalable and efficient. We filter out noun phrases with low frequencies from this matrix, as the small number of samples cannot provide enough confidence in our frequency-based measurement. If a low-frequency phrase is similar to a high-frequency phrase, our decision process will not be affected as the decision module employs DS model. We choose a threshold; only phrases with frequency above 15 are used to construct the matrix. The number of such phrases in our dataset is 9,428 (3.11%).

Using the semantic relatedness score matrix, we create a *relatedness dictionary*, which maps a given noun phrase to a list of noun phrases, all of which have a semantic relatedness score higher than the threshold θ_g. The interpretation is that the given noun phrase may be grouped with its list of noun phrases as far as semantics is concerned. Our implementation takes θ_g to be 0.67. The lists also record the corresponding semantic relatedness scores for later use. A sample dictionary entry of the noun phrase "*map*" is:
<"map", [("map", 1.00), ("map view", 0.96), ("interactive map", 0.89),...]>

2.3.3.2. Selecting Noun Phrases Correlated With Permissions. Whether a certain noun phrase is related to a permission is learnt statistically from our dataset. If a permission *perm* and a noun phrase *np* appear together (i.e., *perm* in permission declarations and *np* in the description) in a high number of applications, it implies a close relationship

between the two. This is however not trivial; some noun phrases (*e.g.*, "game" and "application") may occur more frequently than others, biasing such calculations. Moreover, some noun phrases may actually be related to permissions but statistical techniques may not correlate them if they occur together in only a few cases in the dataset. The latter is partially resolved by leveraging the relatedness dictionary from the previous step. Based on existing data mining techniques [**116**], we design a quality evaluation method that (*a*) is not biased to frequently occurring noun phrases, and (*b*) takes into account semantic relatedness between noun phrases to improve the statics of meaningful noun phrases that occurs less than often. For the permission *perm* and the noun phrase *np*, the variables in the learning algorithm are defined as:

MP(**perm**, **np**): An application declares *perm*. Either *np* or any noun phrase with the semantic relatedness score to *np* above the threshold θ_g is found in the description. This variable will increase by 1, if *np* is in the description, or it will increase by the maximal relatedness score of the noun phrase(s) related to *np*.

MMP(**perm**, **np**): An application does NOT declare *perm*. Either *np* or any noun phrase with the semantic relatedness score to *np* above the threshold θ_g is found in the description. This variable will increase by 1, if *np* is in the description, or it will increase by the maximal relatedness score of the noun phrase(s) related to *np*.

PR(**perm**, **np**): The ratio of $MP(perm, np)$ to the sum of $MP(perm, np)$ and $MMP(perm, np)$:

$$PR(perm, np) = \frac{MP(perm, np)}{MP(perm, np) + MMP(perm, np)}.$$

AVGPR(**perm**): The percentage of all the applications in our training set that request *perm*.

INCPR(**perm**, **np**): This variable measures the increment of the probability that *perm* is requested with the presence of *np* or its related noun phrases given the unconditional probability as the baseline:

$$INCPR(perm, np) = \frac{PR(perm, np) - AVGPR(perm)}{AVGPR(perm)}.$$

MMNP(**perm**, **np**): An application declares *perm*. This variable will increase by 1, if none of *np* and noun phrases related to it in the Relatedness Dictionary are found in the description.

NPR(**perm**, **np**): The ratio of $MP(perm, np)$ to the sum of $MP(perm, np)$ and $MMNP(perm, np)$:

$$NPR(perm, np) = \frac{MP(perm, np)}{MP(perm, np) + MMNP(perm, np)}.$$

AVGNP(**np**): Expectation on the probability that one description contains *np* or related noun phrases over the training set. Assume the total number of applications is M. This variable is expressed as:

$$AVGNP(np) = \frac{\Sigma_{i=1}^{i=M} \lambda_i}{M},$$

where λ_i equals 1, if *np* is in the description of the *i-th* application. Or it equals to the maximal semantic relatedness score of its related noun phrase(s) found in description. If neither *np* nor noun phrases related to it in the Relatedness Dictionary are found, $\lambda_i = 0$.

INCNP(perm, np): This variable measures the growth on the probability that one description includes *np* or the related noun phrases with the declaration of *perm* given expectation as the baseline:

$$INCNP(perm, np) = \frac{NPR(perm, np) - AVGNP(np)}{AVGNP(np)}.$$

Semantic relatedness score is taken as weight in the calculations of variables $MP(perm, np)$ and $MMP(perm, np)$, which groups the related noun phrases and resolves the minor case issue. We should note that $INCPR(perm, np)$ and $INCNP(perm, np)$ evaluate the quality of *np* by the growth of the probabilities that *perm* is declared and *np* (or noun phrases related to *np*) is detected in description with the average level as baseline. This design largely mitigates the negative effect caused by the intrinsic frequency of noun phrase. To roundly evaluate the quality of *np* of describing *perm*, we define the $Q(perm, np)$, which is the harmonic mean of $INCPR(perm, np)$ and $INCNP(perm, np)$:

$$Q(perm, np) = \frac{2 \cdot INCPR(perm, np) \cdot INCNP(perm, np)}{INCPR(perm, np) + INCNP(perm, np)}.$$

np with negative values of $INCPR$ or $INCNP$ is discarded as it shows no relevance to *perm*. Each permission has a list of noun phrases, arranged in descending order by the quality value. The *top-k* noun phrases are selected for the permission. We set k=500 after checking the distribution of quality value for each permission. It is able to give a relatively complete semantic coverage of the permission. Increasing the threshold k excessively would enlarge the number of noun-phrase based governor-dependent pairs in the DPR model. So it would reduce the efficiency of AutoCog in matching the semantic meaning for the incoming descriptions.

2.3.3.3. Pair np-counterpart with Noun Phrase. By following the procedure presented in Section 2.3.3.2, we can find a list of noun phrases closely related to each permission. However, simply matching the permission with noun phrase alone fails to explore the context and semantic dependencies, which increases false positives. Although a noun phrase related to "*map*" is detected in the example sentence below, it does not reveal any location permission.

"*Retrieve Running Apps" permission is required because, if the user is not looking at the widget actively (for e.g. he might using another app like Google Maps)*"

To resolve this problem, we leverage Stanford Parser to get the knowledge of context and typed dependencies. For each selected noun phrase np, we denote as $G(np)$ the set of noun phrases that have semantic relatedness scores with np higher than θ_g. Given a sentence in description, our mechanism identifies any noun phrase $np' \in G(np)$ and records each np-counterpart nc (recall that np-counterpart was defined as a collective term for verb phrases, noun phrases, and possessives for the target noun phrase), which has direct/indirect relation with np'. For each noun-phrase based governor-dependent pair <nc, np>, let the total number of descriptions where the pair <nc, np'> is detected be SP. In the SP applications, let the number of application requesting the permission is tc. We keep only those pairs for which (1) $tc/SP > Pre_T$, (2) $SP > Fre_T$, where Pre_T and Fre_T are configurable thresholds. Thus we maintain the precision and the number of samples large enough to yield statistical results with confidence.

2.3.4. Decision

In DPR model, each permission has a list of related pairs of np-counterpart nc_{dpr} and noun phrase np_{dpr}, which reveal the security features of the permission. For an input application whose description has to be checked, the NLP module extracts the pairs of np-counterpart nc_{new} and noun phrase np_{new} in each sentence. We leverage the DS model to measure the semantic relatedness score $RelScore(txt_A, txt_B)$ between the two texts txt_A and txt_B. The sentence is identified as revealing the permission, if $<nc_{new}, np_{new}>$ is matched with a pair $<nc_{dpr}, np_{dpr}>$ by fulfilling the conduction:

$$RelScore(nc_{new}, nc_{dpr}) > \Upsilon,$$

$$RelScore(np_{new}, np_{dpr}) > \Theta.$$

Here, Υ and Θ are the thresholds of the semantic relatedness score for np-counterparts and noun phrases. The sentences indicating permissions will be annotated. Besides, AutoCog finds all the questionable permissions, which are not warranted in description.

2.4. Implementation

NLP Module: We use the NLTK library in Python and regular expression matching to implement the SBD. NLTK is also used for removing stopwords and normalizing words using lemmatization based on WordNet. Stanford Named Entity Recognizer is used for removing named entities.

DS and DPR Models: Noun phrases are classified by frequency. High-frequency noun phrases are grouped based on semantic relatedness score by utilizing the library `esalib`[6].

[6] `https://github.com/ticcky/esalib`

This library is the only currently maintained, open-source implementation of ESA that we could find. Our training algorithm on descriptions and permissions of large-scale applications selects the semantic patterns, which strongly correlate with the target permission by leveraging the frequency-based measurement and ESA. Our current implementation pairs np-counterpart of length one (noun, verb, and possessive) with noun phrases. The np-counterpart could be easily extended to multiple words, possibly with a few considerations about maximum phrase length, and so on.

Overall, We implement AutoCog with over 7,000 lines of code in Python and 500 lines of code in Java.

2.5. Evaluation

We first describe our dataset and methodology for collecting sensitive permissions. Then, AutoCog's accuracy is evaluated by comparing with Whyper [**118**]. Finally, we discuss our measurements, which investigate the overall trustworthiness of market and the correlation between description-to-permission fidelity and application popularity.

2.5.1. Permission Selection and Dataset

The Android APIs have over a hundred permissions. However, some permissions such as the permission VIBRATE, which enables vibrating the device, may not be as sensitive as, for example, the permission RECORD_AUDIO, which enables accessing the microphone input. It is not so useful to identify permissions that are not considered sensitive. The question to ask then is, *what permissions are the users most concerned about from the security/privacy perspective?*

Felt et al. [**61**] surveyed 3,115 smartphone users about 99 risks and asked the participants to rate how upset they would be if a given risk occurred. We infer 36 Android platform permissions from the risks with highest user concerns. Since we focus here on third-party applications, we first remove from this list the Signature/System permissions, which are granted only to applications that are signed with the device manufacturer's certificate. Seven permissions were removed as a result. The 29 remaining permissions are arranged in descending order by the percentage of applications requesting it in our dataset, which is collected randomly. We select the top 14 permissions in our evaluation, because the ground-truth of our evaluation relies on readers to identify whether sentences

Table 2.2. Permissions used in evaluation

Permission	**#App (Percentage %)**
WRITE_EXTERNAL_STORAGE	30384 (80.29 %)
ACCESS_FINE_LOCATION	16239 (42.91 %)
ACCESS_COARSE_LOCATION	15987 (42.24 %)
GET_ACCOUNTS	12271 (32.42 %)
RECEIVE_BOOT_COMPLETED	9912 (26.19 %)
CAMERA	6537 (17.27 %)
GET_TASKS	6214 (16.42 %)
READ_CONTACTS	5185 (13.70 %)
RECORD_AUDIO	4202 (11.10 %)
CALL_PHONE	3130 (8.27 %)
WRITE_SETTINGS	3056 (8.07 %)
READ_CALL_LOG	2870 (7.58 %)
WRITE_CONTACTS	2176 (5.74 %)
READ_CALENDAR	817 (2.16 %)

Permission name; Number/percentage of applications request the permission within 37,845 applications;

in application description imply sensitive permissions; the consequent human efforts make it difficult to review large number of descriptions.

We collected the declared permissions and descriptions of 37,845 Android applications from Google Play in August 2013 for the purpose of training the DPR model and evaluate AutoCog's accuracy. The permissions that constitute the subject of our study can be divided into 3 categories according to the abilities that they entail: (1) accessing user privacy, (2) costing money, and (3) other sensitive functionalities. Applications request the permissions to access privacy may leak users' personal information such as location to third parties without being awared. Permissions costing money, such as CALL_PHONE, may be exploited resulting in financial loss to the users. Other sensitive permissions may change settings, start applications on boot, thus possibly wasting phone's battery, and so on. In Table 2.2, we list the number and percentage of applications declaring each permission in our dataset.

Table 2.3. Statistics and settings for evaluation

Permission	Fre_T	Pre_T	$G_d(\%)$
WRITE_EXTERNAL_STORAGE	9	0.87	38.7
ACCESS_FINE_LOCATION	6	0.85	40.7
ACCESS_COARSE_LOCATION	5	0.8	35.3
GET_ACCOUNTS	4	0.8	26.0
RECEIVE_BOOT_COMPLETED	5	0.85	37.3
CAMERA	3	0.8	48.7
GET_TASKS	3	0.9	2.0
READ_CONTACTS*	3	0.8	56.8
RECORD_AUDIO*	3	0.8	64.0
CALL_PHONE	2	0.8	10.0
WRITE_SETTINGS	2	0.85	44.7
READ_CALL_LOG	3	0.95	6.0
WRITE_CONTACTS	2	0.9	42.0
READ_CALENDAR*	1	0.85	43.6

Hidden permissions are shadowed;
* sampled by around 200 applications, others by 150 applications

We also parsed the metadata of another 45,811 Android applications from Google Play in May 2014 for our measurements, which assess the description-to-permission fidelity of large-scale applications in Google Play and investigate the correlation between description-to-permission fidelity with application popularity. The metadata include the following features: *category of application, developer of application, number of installations, average rating, number of ratings, descriptions and declared permissions of application.*

2.5.2. Accuracy Evaluation

2.5.2.1. Methodology. WHYPER studied three permissions: READ_CALENDAR, READ_CONTACTS, and RECORD_AUDIO; Their public results are directly utilized[7] as the ground-truth. The validation set contains around 200 applications for each of the three permissions, where each sentence in the descriptions is identified if revealing the permission by human readers. Moreover, to assess AUTOCOG's ability of

[7] `https://sites.google.com/site/whypermission/`

generalization over other permissions in Table 2.2, we further randomly select 150 applications requiring each one (except the three permissions previously evaluated in public results of WHYPER) as the validation set. For each permission, the complementary set of the validation set is used as the training set to construct the DPR model, which ensures that the validation set is independent of the training set. To get the results of WHYPER on other permissions, we leverage the output of PScout [**26**] and manually extract the semantic pattern set from Android API document[8] following the method presented by Pandita et al. [**118**]. WHYPER's methodology does not work for some permissions such as RECEIVE_BOOT_COMPLETED as they do not have any associated API. To ensure the correctness of our understanding of WHYPER's methodology, we contacted WHYPER's authors and confirmed our understanding and conclusions. We also tested the system over the applications in their public results and get exactly the same output as those published, further validating the system deployment (source code is released publicly).

Regarding the ground-truth of other permissions that we extend to, we invite 3 participants to read the description and label each sentence as whether or not it suggests the target permission. The description will be classified as "good" when at least two human readers could infer the permission by one sentence in that, or it will be labeled as "bad". Column G_d" in Table 2.3 is the percentage of "good" descriptions for applications requesting each sensitive permission. The percentage values of "good" descriptions for the 3 permissions GET_TASKS, CALL_PHONE, and READ_CALL_LOG are lower than 10%. We call these permissions rarely described well in descriptions, *hidden permissions*. The scarcity of qualified descriptions leads to the lack of correlated semantic patterns. It

[8] `http://pscout.csl.toronto.edu/download.php?file=results/jellybean_publishedapimapping`

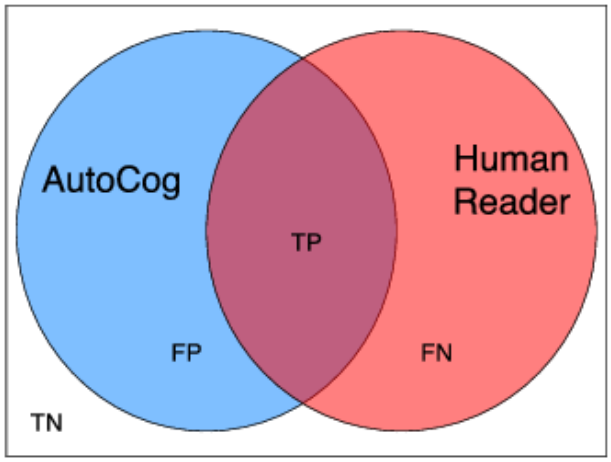


Figure 2.4. Interpretation of metrics in evaluation

would hinder the measurement of description-to-permission fidelity. After removing the 3 hidden permissions, our evaluation focuses on the other 11 permissions.

In training the DPR model, the two thresholds Pre_T and Fre_T balance the performance on precision and coverage of AutoCog. The settings in Table 2.3 depend on the percentage of applications requesting the permission in the training set. For a permission with fewer positive samples (application requires that permission), each pair of np-counterpart and noun phrase related to it tends to be less dominant in amount, we adjust Fre_T accordingly to maintain the performance on recall. We keep Pre_T high across permissions, which aims at enhancing the precision of detection.

Within the process of deciding if each application description in valuation set warrants permissions, we set the two thresholds Υ=0.8 and Θ=0.67 by empirically finding the best values for them. Low threshold reduces the performance on precision and increasing the threshold excessively causes the increment on false negatives. We set up the threshold Θ lower than Υ, because noun phrases has more diversity in patterns than np-counterparts;

phrases containing various numbers of words organized in different orders may express the similar meaning.

Our objective is to assess how closely the decision made by AutoCog on the declaration of permission approaches human recognition given a description. The number of true positives, false positives, false negatives, and true negatives are denoted as TP: the system correctly identifies a description as revealing the permission, FP: the system incorrectly identifies a description as revealing the permission, FN: the system incorrectly identifies a description as not revealing the permission, and TN: the system correctly identifies a description as not revealing the permission. Interpretation of the metrics is shown in Figure 2.4. Intersection of decisions made by AutoCog and human is true positive. Difference sets between decisions made by AutoCog and human are false positive and false negative, respectively. Complementary set of the union of decisions made by AutoCog and human is true negative. Values of *precision*, *recall*, *F-score*, and *accuracy* represent the degree to which AutoCog matches human reader's recognition in inferring permission by description.

$$\begin{aligned} Precision &= \frac{TP}{TP+FP}, \\ Recall &= \frac{TP}{TP+FN}, \\ F\text{-}score &= \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}, \\ Accuracy &= \frac{TP+TN}{TP+FP+TN+FN}. \end{aligned}$$

2.5.2.2. Results. Results of our evaluation are given in Table 2.4. AutoCog matches human in inferring 11 permissions with the average precision, recall, F-score, and accuracy

Table 2.4. Results of evaluation

System	Permission	*TP*	*FP*	*FN*	*TN*	*Prec* (%)	*Rec* (%)	*F* (%)	*Accu* (%)
AutoCog	WRITE_EXTERNAL_STORAGE	53	6	5	86	89.8	91.4	90.6	92.7
	ACCESS_FINE_LOCATION	57	3	4	86	95.0	93.4	94.2	95.3
	ACCESS_COARSE_LOCATION	49	1	4	96	98.0	92.5	95.1	96.7
	GET_ACCOUNTS	34	4	5	107	89.5	87.2	88.3	94.0
	RECEIVE_BOOT_COMPLETED	51	6	5	88	89.5	91.1	90.3	92.7
	CAMERA	67	7	6	70	90.5	91.8	91.2	91.3
	READ_CONTACTS	99	5	9	77	95.2	91.7	93.4	92.6
	RECORD_AUDIO	117	10	11	62	92.1	91.4	91.8	89.5
	WRITE_SETTINGS	65	7	2	76	90.3	97.0	93.5	94.0
	WRITE_CONTACTS	57	4	6	83	93.4	90.5	91.9	93.3
	READ_CALENDAR	79	5	6	105	94.0	92.9	93.5	94.4
	Total	728	58	63	936	92.6	92.0	92.3	93.2
Whyper	WRITE_EXTERNAL_STORAGE	11	8	47	84	57.9	19.0	28.6	63.3
	ACCESS_FINE_LOCATION	31	1	30	88	96.9	50.8	66.7	79.3
	ACCESS_COARSE_LOCATION	28	1	25	96	96.6	52.8	68.3	82.7
	GET_ACCOUNTS	9	2	30	109	81.8	23.1	36.0	78.7
	RECEIVE_BOOT_COMPLETED	*Fail to get results*							
	CAMERA	26	4	47	73	86.7	35.6	50.5	66.0
	READ_CONTACTS	89	9	19	73	90.8	82.4	86.4	85.3
	RECORD_AUDIO	105	10	23	62	91.3	82.0	86.4	83.5
	WRITE_SETTINGS	59	24	8	59	71.1	88.1	78.7	78.7
	WRITE_CONTACTS	53	9	10	78	85.5	84.1	84.8	87.3
	READ_CALENDAR	78	15	7	95	83.9	91.8	87.6	88.7
	Total	489	83	246	817	85.5	66.5	74.8	79.9

as 92.6%, 92.0%, 92.3%, and 93.2%. As discussed before, Whyper fails to get results for permission RECEIVE_BOOT_COMPLETED. For the remaining 10 permissions, Whyper achieves the average precision, recall, F-score, and accuracy as 85.5%, 66.5%, 74.8%, and 79.9%.

Across the permissions evaluated, the least precision and recall of AutoCog are 89.5% and 87.2%. Even for the cases with low percentage of "good" descriptions and low number of positive samples (permissions GET_ACCOUNT and READ_CALENDAR), our learning-based algorithm and employment of ESA could still get the DPR model aligning with user's recognition well. Whyper could only infer 5 permissions from description (last 5 in Table 2.4) with both the values of precision and recall higher than 70%. For these permissions, the API documents provide a relatively complete and accurate semantic pattern set. The example patterns such as <"*scan*","*wifi*">, <"*enable*","*bluetooth*">,

and <"*set*","*sound*"> could be extracted from the API document of the permission WRITE_SETTINGS. However, WHYPER does not perform well on the other 5 permissions. Our understanding is that the patterns extracted from API documents in these cases are very limited to cover the natural-language descriptions with great diversity. For example, the APIs mapped with the permission to write to external storage are related only to download management. Many intuitive patterns, such as <"*save*", "*sd card*">, <"*transfer*", "*file*">, <"*store*", "*photo*"> cannot be found in its API document. It is the same with <"*scan*", "*barcode*">, <"*record*", "*video*"> for camera permission, <"*integrate*", "*facebook*"> (in-app login) for permission to get user's accounts, and <"*find*", "*branch*">, <"*locate*", "*gas station*"> for location permissions. Given WHYPER's big variance of performance and our investigation on its source of textual pattern set, we find that suitability of API document to generate a complete and accurate set of patterns varies with permissions due to the limited semantic information in APIs. AUTOCOG relies on large number of descriptions in training, which would not be restricted by the limited semantic information issue and has stronger ability of generalization over permissions.

Whether or not the API documents are suitable for the evaluated permissions, we note that AUTOCOG outperforms WHYPER on both *precision* and *recall*. Next we discuss several case studies to thoroughly analyze the benefits and limitations of our design.

AutoCog *TP*/Whyper *FN*: The advantage of AUTOCOG over WHYPER on false negative rate (or recall) is caused by: (1) the difference in the fundamental method to find semantic patterns related to permissions, (2) we include the logical dependency between noun phrases as extra ontology. WHYPER is limited by the use of a fixed and limited set of

vocabularies derived from the Android API documents and their synonyms. Our correlation of permission with noun-phrase based governor-dependent pair is based on clustering results from a large application dataset, which is much richer than that extracted from API documents. Below are 3 examples:

"*Filter by contact, in/out SMS*"

"*Blow into the mic to extinguish the flame like a real candle*"

"*5 calendar views (day, week, month, year, list)*"

The first sentence describes the function of backing up SMS by selected contact. The second sentence reveals a semantic action of blowing into the microphone. The last sentence introduces one calendar application, which provides various views. In our DPR model, the noun-phrase based governor-dependent pairs <*filter, contact*>, <*blow, mic*>, and <*view, calendar*> are found to be correlated to the 3 permissions, READ_CONTACTS, RECORD_AUDIO, and READ_CALENDAR. While the semantic information for the first two sentences cannot be found by leveraging the API documents. For the last one, WHYPER could only detect it, as "*view*" and "*calendar*" are tagged with *verb* and *noun*, respectively (both of them are tagged as *noun* here).

AutoCog *TN*/Whyper *FP*: One major reason for this difference in detection is that WHYPER is not able to accurately explore the meaning of noun phrase with multiple words. Below is one example:

"*Saving event attendance status now works on Android 4.0*"

The sentence tells nothing about requiring the permission to access calendar. However, WHYPER incorrectly labels it as revealing the permission READ_CALENDAR, because it parses resource name "event" and maps it with action "save". AUTOCOG differentiates

the two phrases " event attendance status" and "event" by using ESA and effectively filters the interference in DPR model training and decision-making.

AutoCog *FN*/Whyper *TP*: This difference is caused by the fact that some semantic patterns implying permissions are not included in the DPR model. Below is one example: "*Ability to navigate to a Contact if that Contact has address*"

WHYPER detects the word "contact" as resource name and maps it with the verb "navigate". The sentence is thus identified as revealing the permission to read the address book. However, no noun-phrase based governor-dependent pair in our DPR model could be mapped to the permission sentence above, because the pair <*navigate*, *contact*> is not dominant in the training process. The DPR model might not be knowledgeable enough to completely cover the semantic patterns related to the permission. However, the coverage could be enhanced as the size of training set increases.

AutoCog *FP*/Whyper *TN*: In the training process, some semantic patterns, which do not directly describe the reason for requesting the permission in the perspective of user expectation, are selected in the frequency-based measurement. One example is given as: "*Set recordings as ringtone*"

From this sentence, user could customize her/his ringtone with recording, but it does not directly imply the functionality of recording sound. Our model assigns a high relatedness score between <*set, recording*> and RECORD_AUDIO due to quite a few training samples with related keywords and this permission together. Such cases are due to the fundamental gap between machine learning and human cognition.

AUTOCOG and WHYPER both leverage Stanford Parser [**143**] to get the tagged words and hierarchal dependency tree. The major cause of the common erroneous detection of

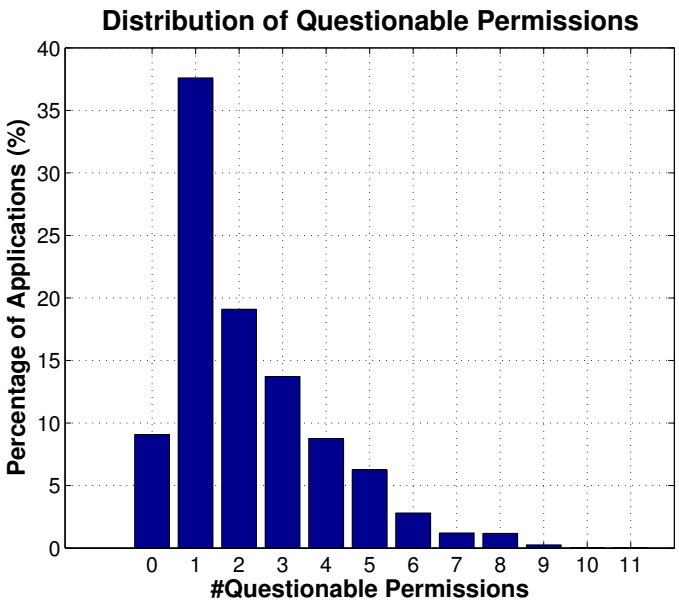


Figure 2.5. Histogram for distribution of questionable permissions

two systems (*FP*, *FN*) is the incorrect parsing of sentence by underlying NLP infrastructure, which has been well stated by Pandita et al. [**118**]. Thus, we would not discuss it in detail given the page limit. As the research in the field of NLP advances underlying NLP infrastructure, the number of such errors will be reduced.

We further list some representative semantic patterns in Table 2.5, which are found to be closely correlated by our DPR model to the permissions evaluated.

Apart from the accuracy of detection, the runtime latency is a key metric in the practical deployment of AutoCog. We select 500 applications requiring each permission and assess the runtime latency of our system in measuring the description-to-permission fidelity. AutoCog *achieves the latency less than 4.5s for all the 11 permissions.*

Table 2.5. Sample semantic patterns

Permission	Semantic Patterns
WRITE_EXTERNAL_STORAGE	<*delete*, *audio file*> <*convert*, *file format*> <*download*, *ringtone*>
ACCESS_FINE_LOCATION	<*display*, *map*> <*find*, *branch atm*> <*your*, *location*>
ACCESS_COARSE_LOCATION	<*set*, *gps navigation*> <*remember*, *location*> <*inform*, *local traffic*>
GET_ACCOUNTS	<*manage*, *account*> <*integrate*, *facebook*> <*support*, *single sign-on*>
RECEIVE_BOOT_COMPLETED	<*change*, *hd wallpaper*> <*display*, *notification*> <*allow*, *news alert*>
CAMERA	<*deposit*, *check*> <*scanner*, *barcode*> <*snap*, *photo*>
READ_CONTACTS	<*block*, *text message*> <*beat*, *facebook friend*> <*backup*, *contact*>
RECORD_AUDIO	<*send*, *voice message*> <*note*, *voice*> <*blow*, *microphone*>
WRITE_SETTINGS	<*set*, *ringtone*> <*customize*, *alarm*> <*enable*, *flight mode*>
WRITE_CONTACTS	<*wipe*, *contact list*> <*secure*, *text message*> <*merge*, *specific contact*>
READ_CALENDAR	<*optimize*, *time*> <*synchronize*, *calendar*> <*schedule*, *appointment*>

2.5.3. Measurement Results

Our measurements begin with assessing the overall trustworthiness of application market, which is depicted by the distribution of questionable permissions. We utilize AutoCog with the DPR model trained in the accuracy evaluation to analyze 45,811 applications. The training set and dataset for measurements are thus disjoint. The histogram for distribution of questionable permissions is illustrated in Figure 2.5. Only 9.1% of applications

Table 2.6. Correlation between application popularity and the number of questionable permissions and permissions requested. All values are statistically significant with p <0.001

	Correlation with application popularity		
Permission Type	$\#install$	$\#rating$	avg_rating
$\#P_q$	-0.106	-0.105	-0.110
$\#P$	0.044	0.050	0.044

are clear of questionable permissions. Moreover, we measure and observe the negative spearman correlation [**90**] between the number of questionable permissions of one application by a specific developer with the total number of applications published by that developer (with $r = -0.405$, p<0.001). A possible explanation is that developer publishing more applications are more experienced and likely to be a development team in a company, who is more standardized and better regulated at developing and deploying its mobile software. The above results reflect the severity of the permission-to-description fidelity issue: application publishers, especially the new or personal developer, generally fail to completely cover all the sensitive permissions. The deployment of AutoCog could thus assist developers produce applications with high description-to-permissions fidelity.

We further investigate the correlation between description-to-permission fidelity and application popularity. Application popularity reveals the developers' benefit and users' attitude towards the application, which thus plays a key role in the interaction between users and developers. In our measurements, application popularity is interpreted by the following features: number of installations ($\#install$), number of ratings ($\#rating$), average ratings (avg_rating). Thus, we measure the (spearman) correlation between these three features with the number of questionable permissions ($\#P_q$) and the number of permissions ($\#P$) requested by application, respectively. Table 2.6 shows that there is a

weak positive correlation between application popularity and the number of permissions requested, which is consistent with the results in [**39, 62**]. It is because that rich functionality of application which implies the need of more permissions is the main feature to drive application popularity.

However, we also find the *weak negative correlation* between the number of questionable permissions and the popularity of application. We should note that all the measured results achieve a p-value less than 0.001, which means the statistical significance. We have the following two guesses. First, for the negative correlation, there are a small part of users who are discreet enough or have the professional knowledge to fully understand the security aspects of application metadata [**63**]. They expect to get permission-related information from the description. Thus the low description-to-permission fidelity negatively affects their decisions of application installation, application assessment, and interest in applications. Secondly, such correlation is weak because most average users cannot tell the questionable permissions based on the description without a tool like AutoCog. Although we could only confirm correlation but not causation here, we expect that wide adoption of AutoCog will help average users to be more security conscious.

2.6. Discussion

AutoCog measures the description-to-permission fidelity by finding relationships between textual patterns in the descriptions and the permissions. Because of the state-of-the-art techniques used and the new modeling techniques developed, AutoCog achieves good accuracy. Still, AutoCog does have limitations because of the approach it uses and the current implementation.

The models learnt in AutoCog are examples of unsupervised learning, which has the drawback of picking relationships that may not actually exist directly. If a noun phrase appears frequently with a permission, the DPR model will learn that they are actually related. For example, if many antivirus applications use the permission GET_TASKS, the "antivirus" noun may become associated with this permission even if there is no direct relationship between the two. From another perspective though, one could argue that this is even better because AutoCog may be able to extract implicit relationships that human readers may easily miss. Anecdotally, for applications with permission GET_TASKS in our experiments, even if human readers could find only 2% of applications whose descriptions reveal that permission, AutoCog finds 18% of such applications.

For the implementation of AutoCog, we could possibly improve the accuracy by including longer noun phrases and np-counterparts. It is an efficiency-accuracy tradeoff. The evaluation of AutoCog also had some limitations. Manual reading is subjective and the results may be biased. However, given that our readers have a technical background, they may be able to discover many implicit relationships that average users ignore, thus putting up greater challenges for AutoCog. Given that whether a description implies a

permission itself is subjective and is consequently lack of ground-truth, manual labeling is the best we can do here.

Malicious developers may provide wrong descriptions to evade this approach. But it will be much easier for even average users to find such mismatch between the app's description and its functionality. And given that most apps are not malicious, such attacks will not affect the training of AutoCog.

2.7. Related Work

NLP has been widely used in the security area. Potharaju et al. [**122**] propose an approach to analyzing natural language text in network tickets to infer the problem symptoms and resolution actions. Some efforts have focused on automating mining of network failures from syslogs [**125**] and network logs [**100**]. Compared with the network tickets and logs, descriptions of applications have much more complex structures and diverse contents, which largely increases the difficulties of ontology modeling. For example, the developer could choose to use either complete sentences or enumeration lists in description; introduction and contact of company may be included for commercial purpose. There are also approaches using a mix of NLP and learning algorithm to infer specifications from API descriptions, code comments, and formal requirement documents [**119**]. The methods proposed in these papers require meta-information from source code. Our design only needs the natural language text of descriptions, which is not constrained by the availability of source code and meta-information.

The permission system in Android security framework manages the access of third-party applications to privacy- and security-relevant parts of API. Many previous studies analyze the permission system and resolve the overprivilege issue [**26, 60**], confused deputy [**40, 64, 51**] and collusion attack [**37**]. Moreover, some studies also investigate the effectiveness of permission model [**62, 91**]. Some researchers have alluded to lack of correlation between permissions and descriptions [**31**]; however, even if permissions and descriptions do not correlate, our solution can bring an improvement to the current situation. Lin et al. [**101**] utilize crowdsourcing collect users expectations of the permissions required by application and Han et al. [**78**] propose a text mining-based similarity measure method

to obtain similar security polices among Android applications, which are both complimentary to our work. While the static/run-time analysis of binaries and programming language analysis enable these approaches to detect overprivilege and confused deputy attack, the end user does not have knowledge about why the permission is requested or tools to assess whether applications overstep user expectation. Our system analyzes the descriptions of applications that the end user has direct and easy access to and labels the sentences revealing sensitive permissions, which enables users to know the reason for declaring the permission in the semantic level.

The most relevant work is WHYPER [**118**], which is the only previous work to our knowledge on bridging the gap between what user expects an application to do and what it really does. Our automatic learning-based approach works directly on large-scale descriptions to select noun-phrase based governor-dependent pairs related to each permission. Thus we would not come across the limitations of WHYPER discussed in Section 2.2.2.

2.8. Conclusion

We propose the system AutoCog that measures the description-to-permissions fidelity in Android, i.e., whether the permissions requested by Android applications match or can be inferred from the applications' descriptions. The use of a novel learning-based algorithm and advanced NLP techniques allows us to mine relationships between textual patterns and permissions. AutoCog outperforms previous work on both performance of detection and ability of generalization over permissions by a large extent. In inferring eleven permissions by description, our system achieves the average precision of 92.6% and the average recall of 92.0% as compared to previous state-of-the-art 85.5% and 66.5%. Our measurements show a generally weak description-to-permissions fidelity on the Google Play store.

CHAPTER 3

DyDroid: Measuring Dynamic Code Loading and Its Security Implications in Android Applications

3.1. Introduction

Android is the dominant smartphone OS. In Q2 2015, IDC placed the worldwide market share of Android at 82.48 percent of all active smartphones [**20**]. However, its open nature and the wide variety of app markets also make it easier to disseminate malware or otherwise untrustworthy apps. In 2016, Mirror reported that up to 10 million Android smartphones had been infected by malicious software [**1**]. After realizing the severity of the malware threat, Google developed and deployed Google Bouncer [**103**], a tool that analyzes apps submitted to Google Play [**73**] and checks them for malicious behavior before publishing them. Other security vendors, such as Bitdefender [**34**], have released products that are deployed on the client side with static malware analysis.

While most apps are distributed as standalone Android application package (APK) files, the Android platform also supports apps dynamically loading additional binaries at runtime by making use of dynamic code loading (DCL). The usage of DCL is not regulated by the OS, and as such it opens up several possible threats. For example, it can be leveraged to evade malware detection. Our research indicates that DCL is widely used in mobile marketplaces. A thorough investigation of various security-relevant aspects of DCL is thus needed.

By using DCL, a developer can change the behavior of an app at runtime in unpredictable ways. This feature can significantly ease the deployment of malicious code. Malware authors are able to evade the security check of offline analysis systems, such as Google Bouncer, by only executing the malicious code when logical conditions are met [**55**]. For example, we developed an app which downloads and dynamically loads known malware over the network. This app passed the security check of Google Bouncer, thus demonstrating the practicality of such threats. Although the similar experiment has been conducted by Poeplau et al. [**121**], our penetration proves that this issue has not been addressed within the recent two years. Moreover, our study of malware samples deployed by DCL in the wild shows instances where the malicious behavior is triggered by the status of the runtime environment, such as availability of a network connection or the system time.

Google's content policy [**74**] for apps on Google Play specifies that all application updates must go through their market. This policy is not effectively enforced, however, because apps can download and dynamically load new code at runtime without using the market. In fact, our experimental and the measurement results in Section 3.5 find numerous apps in the wild that are loading remotely fetched code and violating this policy. Android lacks the ability to track the provenance of code loaded dynamically. Thus, the malicious behaviors and privacy usage in the stealthy channel of DCL are not regulated. Moreover, benign apps that improperly implement DCL can be vulnerable to code injection attacks by other apps on the device; the OS does not enforce any sort of integrity check on dynamically loaded code, and in certain circumstances it is possible for the attacker to tamper with the code to be loaded.

While DCL can be the cause of some security problems, it can also be used to protect the intellectual property of Android developers. Some recent studies [**159, 162**] show that DCL and bytecode encryption can be leveraged to obfuscate an app, which makes it difficult to reverse-engineer with static/dynamic analysis tools. Some security providers, such as Bangcle [**29**] and Ijiami [**84**] provide such services to protect the intellectual property of developers, where the whole app's bytecode is encrypted and stored as a private resource, and an app container dynamically loads the bytecode after decryption.

In this work we perform a large-scale measurement of DCL usage in over 46K apps, investigating the following issues:

- **Provenance.** The loaded bytecode can be either packed as static files in the APK file or fetched stealthily from a remote server at runtime. The latter is capable of evading static/dynamic malware detection mechanisms. We are thus interested in the popularity of its usage, despite the fact that it is prohibited by Google Play.
- **Security risks/implications.** Are there any malicious behaviors hidden in dynamically loaded code? Does the usage of DCL in existing mobile apps have vulnerabilities? How is user privacy tracked in dynamically loaded code?
- **Application hardening.** DCL can be used by apps for the purpose of anti-reverse engineering. In the obfuscated app, the bytecode of the original app is encrypted and repacked. The modules of bytecode decryption, code loading, and app lifecycle construction are interposed in the original app's launching procedure. We investigate an app's pattern after obfuscation, popularity, and

comparison with other common obfuscation techniques, including native code, lexical obfuscation, Java reflection, and anti-decompilation.

- **Usage in the wild.** How widely is DCL adopted in apps in marketplaces? Does the DCL usage correlate with other application attributes, e.g. number of downloads, average rating, and number of ratings? We also study the source of dynamic code loading within the app itself, whether it is the main application or a third party library. For example, a developer may integrate a software development kit (SDK) related to advertising in order to generate revenue. This SDK may use DCL to load portions of its functionality at runtime. We are interested in the entity responsible for using DCL.

We summarize the following challenges. *(1) Code interception.* We need to log the DCL event and intercept the code loaded. The files containing the binaries may be temporary, which are compiled as intermediate results and will be deleted after being merged with the app triggering the DCL behavior. The app's runtime and our code interception are concurrent in the OS. We thus need to instrument the low-level IO-related APIs to enforce mutual exclusion and intercept those loaded binaries. *(2) Provenance/entity identification.* The Android OS itself does not distinguish whether or not a file in storage is downloaded from the network, meaning that it is non-trivial to determine if a file loaded using DCL was originally sourced from the network. Detecting this case will require making use of flow analysis. In addition, the code loading may be triggered by a third-party SDK or library. Our mechanism must also be able to find out whether it was the developer or the third-party library provider who performed DCL. *(3) Obfuscation identification.*

DCL is being actively used for anti-reverse engineering purposes. A proper methodology to accurately detect when an app is obfuscated in this way needs to be developed.

In this work we make the following contributions:

- We develop a framework, DyDroid, which combines both dynamic and static analysis in order to detect DCL and intercept the bytecode and/or native code loaded. The paths to the binaries to be loaded are pushed to a queue, and we instrument the IO-related calls to block file delete and rename operations during the phase of code interception. A flow analysis is implemented in the dynamic analysis, which captures the flow from a URL to a file. DyDroid tracks the call site of DCL behavior by retrieving the element of the Java stack trace. Using this stack trace we are able to differentiate the responsible entity launching DCL. After capturing the dynamically loaded code, DyDroid performs static analysis on the intercepted binaries in order to determine malicious behavior and privacy leakage. In addition, we summarize the general pattern of apps obfuscated with bytecode encryption/loading based on the samples from four mobile app security vendors. The obfuscation pattern involves how the three core components, the app bytecode decryption, DCL, and app lifecycle construction, are organized in an application subclass [**23**] as the container.
- DyDroid is capable of stable operation with little manual intervention. Various types of exceptions are automatically handled, such as device storage running out. It allows us to be the first to conduct a large-scale measurement of DCL over 46K Android apps. Our measurement tracks the provenance of DCL, including local/remote availability, and the entity. We find the 27 apps that violate the

content policy of Google Play by executing the binaries downloaded from the remote server of `Baidu` [**28**]. Generally, over 85% of DCL is initiated by third-party SDKs or libraries. And the app popularity has the positive correlation with DCL adoption. Moreover, we conduct the first large-scale measurement of various obfuscation methods to understand their distribution, pros/cons, and implications.

- Our analysis demonstrates a number of apps in the wild that use DCL to load malware. We find 87 apps which load malicious bytecode or native code at runtime, making them undetectable to existing antivirus tools such as Google Bouncer or VirusTotal [**152**]. We have conducted further analysis which reveals that the execution of the malicious code in these apps is triggered by properties of the runtime environment, such as the system time, GPS service availability, and network connectivity.
- We have identified a vulnerability in a number of DCL apps that leaves them open to code injection attacks [**121**] by other apps on the system. We explore a variant of the code injection vulnerability, where code is loaded from the internal storage of other apps, and we find 7 apps vulnerable to this attack.

The remainder of this chapter is organized as follows: Section 3.2 presents a brief background. We cover the design of DyDroid and its implementation in Sections 3.3 and 3.4. Section 3.5 presents our measurement results over large numbers of real-world apps with DCL, which is followed by the relevant discussion. We have related work in Section 3.6. Finally, we conclude our work in Section 3.7.

3.2. Background

Android apps are written in Java. The classes are compiled to Dalvik bytecode with the tool `dx` and further stored as one file `classes.dex` in the installation package. Each class is loaded and executed in the DVM[1]. Other than the internal static executable bytecode, Android also supports fetching external binaries dynamically. Developers use the class loader provided by Android to load arbitrary executable bytecode, which is stored in files with various formats, such as APK, JAR, ZIP, DEX, and ODEX (optimized DEX). The DEX code is then translated into a performance-optimized version, ODEX. There are two types of basic class loaders `DexClassLoader` [**49**], and `PathClassLoader` [**120**]. Apps can also load native code. The APIs in the Java Native Interface (JNI) [**88**] can be invoked to dynamically load native libraries in `.so` format. Android does not verify the loaded code integrity or have the ability to differentiate whether the file containing loaded binaries is originally packed in the application or downloaded from a remote server at runtime. The binaries can be accessed with diverse methods. For example, an application can even use package contexts to retrieve the classes contained in another application. However, the loading behavior will always be achieved by either using `DexClassLoader`, `PathClassLoader` for DEX code or invoking the APIs `load()`, `loadLibrary()` in the JNI for native code. All DCL goes through one of these points, which provides us with a reliable way to enforce complete mediation in intercepting the loaded code.

[1]Starting with Android 5.0 the Dalvik virtual machine was replaced by ART, an ahead of time compiler. In this work we make use of Android 4.3.1, and thus we discuss Dalvik.

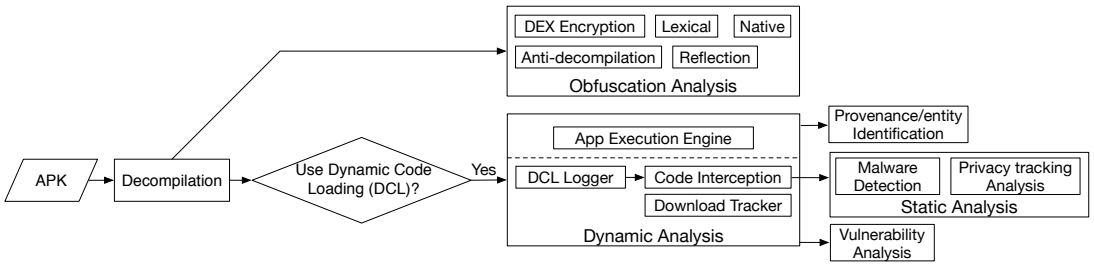


Figure 3.1. DyDroid Architecture

3.3. System Design

3.3.1. System Overview

The architecture of DyDroid is illustrated in Figure 3.1. An APK file will first be decompiled into an intermediate representation (IR). Then we check if the app creates the class loader to dynamically load DEX code or invokes the APIs related to native code loading. We do not verify the reachability of DCL-related code, only its existence within the app. This step simply serves as a filter to determine which apps to investigate further using our dynamic analysis. Apps containing DCL-related code are then executed and our App Execution Engine is used to log DCL events and track files downloaded remotely during execution. Using this information we are able to determine the provenance of the loaded code (local or remote) and whether the DCL is vulnerable to code injection attacks. The intercepted code will be passed to our static analysis to investigate the existence of malicious behavior and privacy leakage.

We also perform obfuscation analysis by checking the Android manifest file and the availability of basic components against a series of rules to identify whether bytecode loading and encryption are applied to obfuscate the app. The method is also designed to

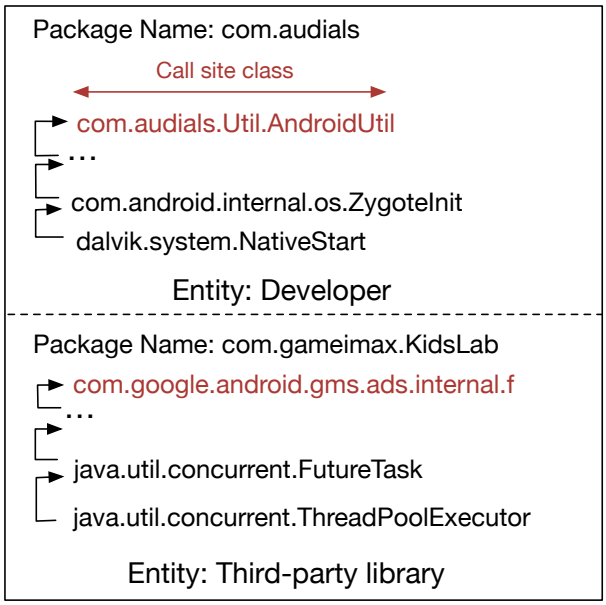


Figure 3.2. Java Stack Trace Element

recognize the usage of other anti-reverse engineering techniques, including lexical obfuscation, reflection, native code, and anti-decompilation.

3.3.2. Dynamic Analysis

To completely capture loading events, we modify the Android framework. All DCL events an app can use go through `DexClassLoader` or `PathClassLoader` in the DVM, or `load()` or `loadLibrary()` in the JNI. As such, we instrument these methods to record the following information: (1) path to the loaded file with various formats, e.g., so, APK, ZIP, JAR, DEX; (2) path to the directory storing the optimized version of the DEX code; (3) the call site class of the DCL (the class where the class loader is created). We determine the call site by analyzing the stack trace [**87**]. An example of this analysis can be found in Figure 3.2. We record the classes of the sequence of objects whose methods are called

Table 3.1. Rules of download tracker

source: URL, sink: File
URL:
URL → InputStream
InputStream:
InputStream → InputStream
InputStream → Buffer
Buffer:
Buffer → InputStream
Buffer → OutputStream
OutputStream:
OutputStream → Buffer
OutputStream → OutputStream
OutputStream → File
File:
File → File
File → InputStream

when the class loader is initialized, and the top element of the stack trace is the call site class, which is used to figure out whether the developer or a third party library provider launched the DCL. Our DCL logger skips the system binaries, such as native libraries in `/system/lib`, which are provided by security-trusted OS vendors and are thus not in our scope.

When a DCL event is captured, the path to the file being loaded is stored in a queue and logged. In some third-party libraries, such as the Google Ads library, we observed that the files loaded are temporary, meaning they are deleted after the load. As such, fully intercepting the loaded binaries requires enforcing mutual exclusion. We modify the methods related to file deleting and renaming in `java.io.File` to ensure that the delete and rename operations silently fail for files in our queue of dynamically loaded binaries. This ensures the dynamically loaded code remains available for later analysis.

To investigate if a loaded file is packed locally or fetched remotely at runtime, our dynamic analysis includes taint tracking regarding file downloads. As shown in Table 3.1, URL and File are modeled as source and sink. We first instrument the class `URL` to record all the URLs initialized, and the class `URLConnection` with its subclasses, such as `HttpURLConnection`, `HttpsURLConnection`, and `FtpURLConnection`, to track the flow from URL to InputStream. Next, we instrument the constructors, and the methods `read()` and `write()` of the classes `InputStream`, `Reader`, `OutputStream`, `Writer`, including their subclasses in the package `java.io.*`, to track the flows among InputStream, Buffer, OutputStream, and File. The copy and renaming operations are considered as the flows among Files. Each object is represented by type and hash code [**86**]. In the data flow graph, we search the paths from a URL to a File.

In order to increase the chance that our dynamic analysis engine triggers the DCL event, we employ Fuzz testing [**128, 105, 79, 148**]. Specifically, a sequence of events is generated and triggered automatically as inputs to UI elements, which invoke the callback functions and Android framework. We utilize the fuzzing tool Monkey [**148**], which runs on top of a device running the instrumented version of Android 4.3.1. We verify that the DCL-related APIs in Android 7.1 do not change significantly from Android 4.3.1. `DexClassLoader` and `PathClassLoader` remain the same and ART uses DEX to load. The class `Runtime` only adds an API (load0) to load native code. We only need to add hooking to one API to adapt to the latest version of Android. Our system modification thus works well on newer versions of Android.

Provenance/entity Information. Poeplau et al. had shown that it was feasible to evade Google Bouncer with DCL [**121**]. Our experiment indicates that the issue has not been

fixed in the recent two years. We prepared a malicious app App_M, that is derived from known malware [**106**]. We submitted this app to Google Play and it was rejected by Google Bouncer. We then implemented a new app App_L, which can dynamically load App_M from a server at runtime. The server decides whether or not to send App_L the link to the copy of App_M. The app App_L was approved and released on Google Play. We should note that we disabled the malware delivery at the server side during application review and after release. We thus make sure no end user is affected by the malware.

Google has a content policy [**74**] that apps should not using side channels other than the standard updates to modify the APK binary code. In other words, when using DCL it is only legitimate to load code already packaged into the installation package. Remote fetching new code is not allowed. However, we still found some apps fetching binaries from a remote server at runtime. This technique can ease the application updates for developers. For example, a normal application update can be packed as a DEX file and be pushed to devices instantly when it is ready, bypassing lengthy application review on the store. However, loading the code fetched remotely brings malware authors a stealthy channel to deploy malicious code after app approval by the store. Given the limitation of offline analysis systems [**115, 155**], the malware detection deployed on mobile marketplaces can be evaded easily, where the malicious code is actually fetched and executed after the application's public release. Moreover, the Android OS currently cannot tell whether the file to be loaded is fetched remotely. Thus, the existing Android ecosystem lacks a mechanism to enforce Google's policy. The DCL logger and download tracker of DyDroid records the provenance information for remotely downloaded files, meaning we can identify which DCL apps are loading code remotely and thus violating the policy.

In addition to tracking local/remote provenance, we can also determine if the DCL event was triggered by the app itself or a third party library included with it. In Java, packages organize classes into namespaces. Classes in the same package can access the package-private and protected members of each other. Android apps inherit this organizational pattern. Each app has a unique application package name that includes the classes from developers, while the third-party libraries are organized with different package names. As shown in Figure 3.2, the package name can be used to determine if the DCL event was triggered by the main app or a third party library.

Vulnerability Analysis. When studying DCL we noticed a potential vulnerability depending on where apps load their dynamic code from. If bytecode is being loaded, then the parameter `dexPath` in the constructors of `DexClassLoader` and `PathClassLoader` specifies the list of files containing bytecode to be loaded. If native code is being loaded, the parameter `libName` of API `loadLibrary()` represents the name of the library containing the loaded code. It will be passed to function `mapLibraryName()` to get the path to the library file given the runtime environment, and the API `load()` does the real job of loading code from the library file.

Under Android, the responsibility for verifying the integrity of the file being loaded is on the developer, who is generally more concerned with functionality than security. Thus, if the loaded code is located on a space writable by other parties, then other apps can replace the file with another, and cause the code to be loaded in the context of the vulnerable app.

Poeplau et al. [**121**] have previously discussed the problem of dynamic code loading from external storage. As such, part of our analysis checks for this vulnerability in our

application set. In addition, we identify another variant of this vulnerability. During vulnerability analysis, we check if the path of the file loaded falls into either of the following categories:

- **External storage.** Prior to Android 4.4, any app is able to modify the contents of external storage without declaring special permissions. This means that if an app performs DCL from a file on external storage (for example, in `/mnt/sdcard/`[2]), any other app can replace that file. After Android 4.4, apps must declare a special permission in order to write to external storage. This is a common permission, however, and it would not be unusual for an app to have it.
- **Internal storage of other apps.** Android provides each app private internal storage where only that app can create files. However, we have observed that other apps can dynamically load binaries from the private internal storage of other apps. While it unclear why an app developer would want to do this, we noticed that some do. As such, we flag this situation as a potential vulnerability in the apps that load files from the internal storage of other apps, e.g. from `/data/data/otherAppPackageName/`.

3.3.3. Static Analysis

Malware Detection. The dynamic code intercepted by our system can be bytecode or native code. Most malware detection systems for Android, however, only operate on bytecode. As such, in order to perform malware detection of our captured samples we make use of the publicly available malware analysis system DroidNative [**6, 54**], which

[2]The example paths to external storage and internal storage are based on the observation in the Android device, where we conduct our measurement.

is able to analyze both bytecode and native code binaries. DroidNative translates the binaries compiled for various platforms, such as ARM and x86, to the platform independent Malware Analysis Intermediate Language (MAIL) [**5**]. MAIL provides a high-level representation of the disassembled binary program, which includes the specific information such as control flow information, function/API calls and patterns. Given the issue of zero-day malware and malware variants, DroidNative utilizes a learning-based method, and trains a classifier based on the annotated control flow graphs (ACFG) of malware. In the evaluation with traditional malware variants, DroidNative achieves the detection rate of 99.48%. In DyDroid, we train DroidNative with 1,240 apps from 19 malware families which are collected from two sources [**165, 106**]. We then use the system to detect malware samples from among the dynamically loaded code we intercept. Specifically, DroidNative conducts a subgraph matching on the ACFG and flags a malware when the degree of match is over 90%. When a sample is flagged as malware, we manually verify it in order to reduce the possibility of false positives.

We then go further, and for each intercepted file containing malicious code, we validate whether the loading event can be reproduced under a variety of runtime environment configurations. First, we set the system time to be before the app's release date. Second, we enable airplane mode but intentionally re-enable the WiFi connection. Third, we enable airplane mode to disable all Internet connectivity. Finally, the location service is disabled.

Privacy Tracking Analysis. Previous related studies [**165, 146**] found that Android apps frequently transmit sensitive data to unknown destinations without user consent.

However, the severity of this problem remains unclear within DCL. As such, we conduct a static data-flow analysis on intercepted DEX code.

Our data-flow analysis leverages the public system FlowDroid [**25**], which achieves the high precision 86% and recall 93% in data leak detection. FlowDroid requires the application installation package as input. The manifest file and layout resource are used to locate the app entry points. While we only have the DEX binaries intercepted. Unlike a whole app that has the well-defined components interacting with the system, the loaded code interacts with the app, and an arbitrary class can be the entry point to the loaded libraries. We thus modify FlowDroid regarding the definition of program entry point and remove its dependency on the manifest file and layout resources.

Felt et al. [**61**] surveyed 3,115 smartphone users about 99 risks and asked the participants to rate how upset they would be if a given risk occurred. Specifically, their survey covered 11 data types regarding user privacy. Additionally, we combine the data types reported in other mobile privacy tracking studies [**57, 160, 167**], as listed in Table 3.10. The 18 types of privacy are classified into 5 categories:

- **Location.** Android provides the APIs that can be invoked to fetch user's real-time location.
- **Phone identity.** The smartphone identifiers (IMEI, IMSI, ICCID) can be used to recognize the device's identity.
- **User identity.** The user identifiers (phone number, device accounts) can be used to track user's identity.
- **Usage pattern.** The system's `PackageManager` APIs support fetching the apps and packages installed on device. Third parties are strongly motivated to track

this type of data. For example, advertisement providers can infer user's interests from the installed apps and selectively push customized ad content.

- **Content provider.** Content providers control the access to a structured set of data. Android has a series of default content providers to manage the private user data, e.g. bookmark in browser, address book, and call history.

For the categories location, phone identity, user identity, and usage pattern, our data-flow analysis checks the invocation of related system APIs and callback functions as the source of privacy tracking. Content provider is identified by URI [**18**] and organized as an *SQLite* database with schema and table definitions. We thus look up the URI mapped with each privacy-sensitive content provider as the source of data flow. We use the comprehensive list of sinks in the SuSi project [**127**], which was discovered by a learning approach.

3.3.4. Obfuscation Analysis

In addition to the dynamic and static analysis components of DyDroid, we also analyze obfuscation techniques applied to the apps. Based on our observation of obfuscated app samples served by various providers, e.g., Bangcle, Ijiami, 360 [**3**], and Alibaba [**7**], we found that these services share a common design based on application rewriting, where code loading and encryption are actively used with the purpose of anti-reverse engineering. An application subclass is implemented as a container. When the application process is started, this container is instantiated before any of the application's components. The class loader created in the container loads the bytecode of other components from an encrypted file packed as a local resource, and the customized code decryption runs before

the actual code load. Thus, it is impossible to reverse-engineer the bytecode through static analysis. In addition, some tricks are applied to make dynamic analysis more difficult, e.g., for one app, three distinct processes are started and attach the `ptrace` system call [**124**] in a loop to prevent the execution from being tracked and controlled externally.

When all the following rules are fulfilled, we identify the obfuscated app with DCL applied based on the decompiled IR as illustrated in Figure 3.1.

- The attribute `android:name` is defined in the `application` tag in the application's manifest file and a class loader is instantiated in this class. This is the name of the class that executes before any other components of the app. This class (container) injected via application rewriting performs as the new entry point of the whole app. It invokes the added native code to decrypt the original bytecode of the app. Moreover, the bytecode is loaded at runtime and the app lifecycle is constructed within this class.
- Not all the application components declared in the manifest file are found in the decompiled code, and a file in the format that supports bytecode storage is found locally. The decompilation tool used by us is designed for the app organized in the general pattern, where the bytecode is stored in the file `classes.dex`. Thus, the obfuscated DEX code stored as a resource (normally in the `assets` folder) cannot be found and decompiled by the reverse engineering tool. However, all the components to be invoked at runtime need to be declared in the manifest file. We thus treat this mismatch as an identification rule.
- The job of decryption is normally implemented in native code for the sake of security. The application container class that is discussed above uses the JNI to

load the local `.so` file to decrypt the bytecode. Although the code decryption may be implemented in Java within the application container class, the decryption process will be exposed to attackers, who can reverse engineer the application container class. In our dataset we did not find any examples of using Java to do the code decryption.

Our mechanism to detect obfuscation techniques includes several methods in parallel with DCL, such as lexical obfuscation, and anti-decompilation. We intend to deliver the compressive measurement results regarding the app obfuscation usage in the current mobile marketplace.

Lexical obfuscation is the process where the identifiers of classes, fields, and methods in the bytecode are replaced with meaningless ones, and thus we need to judge whether each identifier makes sense regarding semantics. We implement a parser to extract the identifiers. We compare the identifiers against a language database constructed from `DBpedia` [**47**], which dumps Wikipedia for the purpose of Natural Language Processing (NLP). If the identifiers in an application do not correspond to actual words, then we assume the app has been lexically obfuscated. ProGuard [**123**] has been integrated into Android IDE to provide the lexical obfuscation functionality. One may argue that the ProGuard identifier assignment scheme is rather repetitive and simple to identify, which is straightforward to be used to identify the usage of lexical obfuscation. However, there are several other mobile security vendors having such a functionality, such as Allatori [**9**], where the app is obfuscated by methods other than the simple identifier renaming. Our method is thus able to recognize the obfuscation usage comprehensively.

Reflection allows a running program to retrieve information about itself and the runtime environment, which can be used to instantiate arbitrary classes, invoke methods, and alter data fields. As with native code, although developers may have various purposes of using these techniques, such as performance improvement, accessing private fields and methods, they do increase the bar of reverse engineering, because they make analyzing the program statically very difficult. But dynamic analysis is still able to recover the execution of the apps obfuscated by this method. We determine if reflection is applied by checking the existence of the related APIs of the package `java.lang.reflect`. Moreover, the usage of native code can be identified by confirming with the output of our dynamic analysis.

Anti-decompilation techniques hinder the reverse engineering tools by making the code appear invalid to them. For example, the programming language pattern lacking the one-to-one mapping from DEX bytecode to the target language. When we decompile the Android apps to IR, we record the apps obfuscated with anti-decompilation techniques.

3.4. Implementation

We leverage the open source tool `baksmali` [**141**] to unpack and decompile the installation package into the IR `smali`. The log of our dynamic analysis and the dumped loaded code are stored in the external storage of the device. If the application does not declare the Android permission WRITE_EXTERNAL_STORAGE, we will rewrite and repack the decompiled version with the permission added to the manifest file. Our DCL logger and code interception rely on instrumenting the constructors of `DexClassLoader` and `PathClassLoader`, the APIs `load()` and `loadLibrary()` in the JNI, and the APIs

related to file deleting and renaming in `java.io.File`. The download tracker involves instrumenting the constructor of the class `URL` and the method `getInputStream()` of the class `URLConnection`, including its subclasses. Moreover, the flow among InputStream, Buffer, OutputStream, and File are tracked through the constructors and the methods `read()` and `write()` of the classes `InputStream`, `Reader`, `OutputStream`, `Writer`. We write a script in `Python` to parse the output of download tracker and construct the flow graph of file download.

3.5. Measurement

In this section, we will introduce our measurement data set. We then present our measurements results, which mainly answer the following questions. (1) What are the apps loading code in the remote fetch manner that is prohibited by Google Play, and who is the responsible entity? (2) How is the DCL used for app hardening, specifically, obfuscation? (3) What are the security risks/implications of DCL in the marketplaces? The dynamic analysis runs on the Samsung Galaxy Nexus device with the fuzzing tool Monkey running on top of it.

3.5.1. Data Set

We randomly collected 58,739 apps and the metadata, such as description, rating, the number of downloads, from Google Play in November 2016. The data set includes 42 application categories. 58,685 apps are successfully unpacked and decompiled into the IR. Those apps which fail in the reverse-engineering procedure are obfuscated. The decompiler crashes and does not generate the `smali` code. We find out that 46K apps have DCL operations in the decompiled IR, where 40,849 apps initialize class loaders for loading DEX code, and 25,287 apps invoke related APIs in JNI for loading native code. We note that the DCL may not be actually executed at runtime. We try to avoid blindly exercising app, given the heavy cost of dynamic analysis.

3.5.2. Results

The results of our dynamic analysis are summarized in Table 3.2. The app will be rewritten and repacked with the permission of writing to external storage added, if it is not declared,

Table 3.2. Dynamic analysis summary out of 40,849 apps for bytecode and 25,287 apps for native code

	DEX	**Native**
Failure	**495 (1.21%)**	**330 (1.31%)**
Rewriting failure	454 (1.11%)	133 (0.53%)
No activity	8 (0.02%)	13 (0.05%)
Crash	33 (0.08%)	184 (0.73%)
Exercised	**40,354 (98.79%)**	**24,957 (98.69%)**
Intercepted	16,768 (41.05%)	13,748 (54.37%)

so as to log the DCL. The anti-repackaging technique is applied to some apps, which crashes `apktool`. Moreover, the fuzzing tool cannot exercise those apps without any *Activity* component. Finally, apps may also crash at runtime due to the implementation fault by developers. We overall successfully exercise 40,354 apps for bytecode and 24,957 apps for native code, among which the DCL of 16,768 apps and 13,748 apps are actually executed and the loaded code are successfully intercepted, separately. We note that the loading of system library is not included in our scope, which is provided by security-trusted OS vendors.

By mining the log of DCL from mobile advertisement vendors, such as AdMob, we find the general pattern of the file path to the bytecode loaded by the advertisement libraries "`/data/data/AppPackageName/cache/ad*`". Within the 16,768 apps whose DCL events are captured and loaded bytecode are intercepted, we find out 15,012 apps execute the binaries related to mobile advertisement. Those files are generated intermediately and will be deleted after being merged with the apps which start the DCL behaviors.

Dynamic Code Loading in Mobile Marketplaces. The number of downloads, the number of ratings, and the average rating are used to quantify the application popularity in marketplaces. From Table 3.3, we can see that the apps with DCL are more popular than

Table 3.3. DCL v.s. Application popularity based on 58,739 applications; number of downloads; number of ratings, average ratings

	#Downloads	**#Ratings**	**Rating**
DEX	60,010	2,448	3.91
Without DEX	52,848	2,318	3.77
Native	288,995	8,668	3.82
Without Native	75,127	1,119	3.79

Table 3.4. Responsible entity of DCL out of 16,768 apps for bytecode and 13,748 apps for native code

	3rd-party (#Apps)	**Own (#Apps)**	**3rd-party & Own (#Apps)**
DEX	16,755 (99.92%)	50 (0.30%)	37 (0.22%)
Native	11,834 (86.08%)	2,280 (16.58%)	366 (2.66%)

the complementary set. There are various factors, which affect the application popularity, and we cannot assert there is any causal relation between usage of DCL and application reputation. However, given the high popularity, the security risks of DCL, such as evading malware detection, code injection vulnerability, and privacy tracking, can thus easily affect large numbers of end users.

Provenance/entity Identification. We identify if the third-party or developer is the responsible entity who launches DCL. The results are summarized in Table 3.4. For both DEX and native code, the third-party SDKs and libraries of over 85% are the actual entities to load code at runtime. Given the difficulty of reverse-engineering the code dynamically loaded, protecting the intellectual property is the possible motivation of deploying third-party libraries using DCL.

Table 3.5. Apps fetching binaries from remote servers

Package name
com.ipeaksoft.pitDadGame, com.xy.mobile.shaketoflashlight
org.madgame.Idom, com.yb.sex.cartoon5
com.jianhui.FJDazhan, com.quwenba.i9300manual
com.rhino.itruthdare, com.xiangqi.fanapp.a1521
com.huijia.moyan, org.mfactory.three.bubble
com.huijia.zuoqingwen, apps.simple.recipe
com.xiangqi.fanapp.a1284, com.ioteam.numbertest
com.avpig.acc, air.com.qqqf.xxywszzy2a
com.seven.chuanyueqinggong, com.game.knyds
air.com.qqqf.xxnjyybdc123456, com.seven.tiancantudou
com.conpany.smile.ui, com.classicalmuseumad.cnad
com.seven.chuanyuegongting, com.seven.mengrushenj
com.nexusgame.popbirds, com.XTWorks.lolsol
com.Long.ButtonsShowAndroid

With the download tracker in our dynamic analysis, we find out the 27 apps in Table 3.5, which execute the binaries downloaded from remote servers at runtime. For example, the app `com.classicalmuseumad.cnad`[3] downloads two files in the formats JAR and APK from the domain `http://mobads.baidu.com/ads/pa/`. All the DCL events of loading code in the remote fetch manner are initialized by the advertisement related third-party libraries from `Baidu` [**28**]. The update mechanism of mobile marketplace is a reasonable explanation of the measurement results. Application developer fully controls the update release. SDK vendors cannot predict whether the most up-to-date version of library will be included. In other words, the mobile market channel is not dependable. Fetching the DEX code from a remote server allows the third-party SDK providers to modify the libraries without any constraint, which is prohibited by the content policy of Google Play because it eases the deployment of malware. However, the existing Android

[3]`https://play.google.com/store/apps/details?id=com.classicalmuseumad.cnad`

Table 3.6. #Apps using obfuscation techniques out of 58,739 applications

Technique	**#Apps (%)**
Lexical	52,836 (89.95%)
Reflection	30,664 (52.20%)
Native	13,748 (23.40%)
DEX encryption	140 (0.24%)
Anti-decompilation	54 (0.09%)

OS lacks the ability to track the source of loaded code and is not effective to enforce the policy.

Obfuscation Analysis. The feature of DCL can be used to harden mobile apps. Based on our observation of application samples from mobile application security providers and the general pattern after being obfuscated, we detect the app shielded by DCL and byte-code encryption. Moreover, we also design the method to identify the usage of common obfuscation techniques. Table 3.6 lists how widely each technique is adopted in the apps within our data set.

89.95% apps use the lexical obfuscation. The high adoption rate is expected, as this functionality is included in `ProGuard` and served within Android IDE for free [**123**]. Even with the high popularity, lexical obfuscation just makes the source code not human readable. For reflection and native code, though they may be used for other purposes, such as performance improvement, accessing private fields, they do increase the difficulty of reverse engineering. 52.20% apps adopt reflection and 23.40% apps include native code.

The adoption rate of DEX encryption method is still low 0.24%. DEX encryption has the decryption functionalities inside native layer, and developers may have the compatibility concern, given the Android fragmentation issue [**19**]. It is also possible that this technique is relatively new and does not have enough market penetration. Given the 140

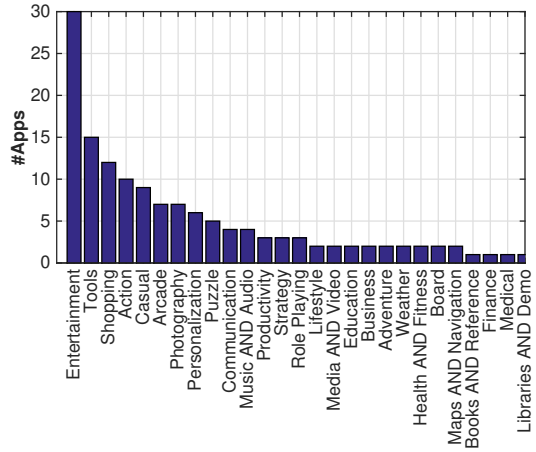


Figure 3.3. #Apps with DEX Encryption v.s. Application Category

apps using DEX encryption, we measure its distribution across application categories, which is illustrated in Figure 3.3. The categories `Entertainment`, `Tools`, and `Shopping` of apps play a dominant role. We further investigate the functionalities of apps in these categories. The apps in the category of entertainment provide the functionalities of controlling smart TV, where the TV vendors are motivated to protect the communication between smartphone and TV from being reverse engineered. The apps in the category of tools are antivirus apps and those in the category of shopping include the sensitive functionalities of payment, which are both obfuscated for the purpose of security.

Anti-decompilation disables the reverse-engineering tool `apktool` by using its implementation bug. As `apktool` keeps evolving, the percentage of apps with anti-decompilation capability remains low 0.09%. Next, we discuss the security risks and implications of DCL.

Malware Detection. Our measurement investigates the malware hidden in DCL. Overall, we find that 87 apps dynamically load malicious binaries in three malware families

Table 3.7. Malwares detected in DCL

	Family	#Apps	Sample App (#Downloads)
DEX	Swiss code monkeys	1	com.sktelecom.hoppin.mobile (10,000,000)
	Adware airpush minimob	2	com.oshare.app (10,000)
Native	Chathook ptrace	84	com.com2us.tinyfarm.normal.freefull.google.global.android.common (10,000,000)

from 91 static files [4]. All the detection results are verified by one of the authors manually with the following method so as to guarantee that there is no false positive. DroidNative outputs the ACFG match of the testing binary with the training malware sample. A testing sample will be flagged as malicious if over 90% ACFG of a malware training sample has the parallel match with its ACFG. In most cases, the identified testing samples only differ from the matched malicious samples in the memory addresses, which depend on where the app is loaded. We note that because these malware samples are loaded dynamically, existing detection systems do not detect them. All of these apps are publicly released on Google Play, which means they pass the security verification of Google Bouncer. Moreover, we submitted the malicious samples to VirusTotal [**152**] (a service that integrates various antivirus products) for scanning and it failed to detect them.

We find the apps loading malicious code in three malware families, and the results are listed in Table 3.7. For each family, one sample application package name is given for the sake of brevity. We share the full results with all malicious apps in our technical report [5]. One app loads the malicious DEX code in the `Swiss code monkeys` family. It adds the malicious code as a service, and sends IMEI, phone number, and IMSI to a remote site. A remote user is able to send and execute a command, such as app installation, website

[4]One app may have multiple malicious files to load.
[5]`http://zyqu.info/DyDroid_DSN.pdf`

Table 3.8. Malicious code loaded in various configurations over 91 files

Configuration	#Files intercepted (%)
System time	72 (79.12%)
Airplane mode/WiFi ON	56 (61.54%)
Airplane mode/WiFi OFF	53 (58.24%)
Location OFF	70 (76.92%)

navigation, adding browser bookmark, sending text message, and blocking test message response. Two apps are found to execute the malicious bytecode in the family `Adware airpush minimob`, where mobile advertisement is pushed to the device via notification. Moreover, shortcuts are placed on users' home screens and browser settings are changed to redirect homepage. There are total 84 apps loads malicious native code in the family `Chathook ptrace`, which mainly targets the two popular chatting apps `QQ` [**126**] and `WeChat` [**153**] with millions of downloads. The malicious app tries to get the root privilege first. Then, it attaches the system call `ptrace` to the two apps as the superuser, controls the two apps, and hooks the Java methods related to the chatting window. Finally, the malware leaks the chat history to a remote server.

We further investigate the malicious loading event can be reproduced with different configurations of runtime environment. The results are listed in Table 3.8. 19 files of malicious code are not loaded when the system time is set before the app's release date, which can be used to bypass the check during the app review phase. Moreover, we also observe the hide of malicious behaviors when connection or location service is not available, where those logical conditions make it more difficult to detect the malware loaded dynamically.

Vulnerability Analysis. The app that loads code from a space writable by other parties is vulnerable to code injections. We classify those apps with risky DCL into two categories:

Table 3.9. Vulnerable applications detected. Apps in the category of external storage are verified as supporting the OS versions lower than 4.4

	Category	#Apps	Package name (#Downloads)
DEX	Internal storage of other applications	0	
	External storage (< Android 4.4)	7	com.longtukorea.snmg (1,000,000) com.felink.android.launcher91 (1,000,000) com.ycgame.cf1en.gpiap (100,000) com.fitfun.cubizone.love (100,000) com.fkccy.view (100,000) com.trustlook.fakeiddetector (10,000) com.leduo.endcallsms (100)
Native	Internal storage of other applications	7	com.devicescape.usc.wifinow (1,000,000) com.renren.and02506 (100,000) air.air.com.hi4o.game.Subway_Rushers (10,000) air.com.fire.ane.test.bubblecrazy (10,000) com.renren.wan.war (10,000) air.com.fire.ane.test.ANETest (1,000) com.moeapps (100)
	External storage (< Android 4.4)	0	

(1) private storage of other apps, (2) public external storage. The results are listed in Table 3.9. We note that all the vulnerable apps are manually confirmed to make sure that even developer fails to enforce integrity verification on the loaded code. We also check the manifest files of those apps in the second category and make sure they do support the OS version lower than 4.4. 14 apps are found to have risky usage of DCL. Three vulnerable apps have over the 1M number of downloads. Both developers and OS vendors should pay attention to security regulation of DCL.

7 of them load native code from the internal storage of other apps. 6 apps load the native code from the file `libCore.so` in the internal storage of the app *com.adobe.air*. The developers of these apps are different from that of the app *com.adobe.air*, and they blindly trust the integrity of the library provided by the Adobe developer, which introduces the extra attack surface for code injection. Another app *com.devicescape.usc.wifinow* loads

the library `libdevicescape-jni.so` from the app *com.devicescape.offloader*, which share the same developer.

7 of them use public storage to cache the bytecode loaded. For example the app *com.longtukorea.snmg* stores its bytecode file `yayavoice_for_assets_2015101201.jar` in the public directory `/mnt/sdcard/im_sdk/jar/`. Malicious parties can exploit these vulnerabilities by replacing the original file with arbitrary binaries. One app with only the permission of writing to the SD card can misbehave with all the permissions declared by the vulnerable app granted.

Privacy Tracking Analysis. We investigate 18 types of privacy tracked in the loaded DEX code with our static analysis. The results are listed in Table 3.10. As we mentioned above, there are 15,012 apps loading the Google Ads library, which has strict control of user privacy and only reads the device settings. However, the remaining 1,756 apps heavily leak various types of user privacy. About 30% apps leak the user's IMEI through DCL. Some highly sensitive types of data, such as location, and installed packages are retrieved in more than 10% apps. As for the responsible entity, the majority of those privacy leakages are exclusively invoked by third-party libraries. The integrated SDK/library is a black-box for the developer, who is not clear about the security risks introduced.

Table 3.10. Privacy tracking in dynamically loaded code based on 16,768 applications (L: location, PI: phone identity, UI: user identity, UP: usage pattern, CP: content provider), Browser: read history & bookmark

Data type	**Categ**	**#Apps**	**Exclusively 3rd-party (%)**
Location	L	254	251 (98.82%)
IMEI	PI	581	576 (99.14%)
IMSI	PI	27	25 (92.59%)
ICCID	PI	8	6 (75.00%)
Phone number	UI	12	10 (83.33%)
Account	UI	23	23 (100.00%)
Installed applications	UP	32	28 (87.50%)
Installed packages	UP	235	231 (98.30%)
Contact	CP	1	1 (100.00%)
Calendar	CP	76	73 (96.05%)
CallLog	CP	32	32 (100%)
Browser	CP	1	1 (100%)
Audio	CP	5	5 (100%)
Image	CP	74	72 (97.30%)
Video	CP	31	31 (100%)
Settings	CP	16,482	16,441 (99.75%)
MMS	CP	1	1 (100%)
SMS	CP	1	1 (100%)

3.5.3. Discussion

Using a fuzzing tool in dynamic analysis may have a code coverage problem. We observe that advertisement libraries initialize most of the DCL events and the DCL events are triggered when the app is launched. Our observation matches the results in MAdScope [**111**]. Thus using monkey is enough regarding the purpose of our measurement.

Regarding the privacy tracking in DCL, users may know and accept it when installing the application. Without this differentiation, it is not possible to know if it is a violation of the promised privacy or not. Deciphering the purpose of privacy tracking is still an open question.

3.6. Related Work

Dynamic Code Loading & Measurement. Gibler et al. [**70**] design the system AndroidLeaks, which performs a static analysis to check user privacy leakage among large-scale Android applications. It does not support the analysis of dynamically loaded code. Grace et al. [**76**] conduct a measurement regarding the privacy and security risks in the advertisement libraries of Android applications, where DCL is defined as a risky flag. Other than simply focusing on advertisement libraries, we include the DCL invoked by developer her/him-self and the third-party libraries for various purposes. Zhauniarovich et al. [**164**] investigate the usage of DCL and reflection in application update, where the native code is not considered in the security model. DEX encryption together with dynamic loading has been recently found in the application of anti-reverse engineering, and some studies investigate recovering the obfuscated applications [**159, 162**]. However, there is no study to uncover its usage cases within the Android applications in current marketplaces, such as popularity, general obfuscation pattern, and pros/cons. Rastogi et al. [**130**] have the system design similar to us, which focuses on the mobile advertisement measurement on the App-Web interface, while we explore the DCL usage. Poeplau et al. [**121**] find out the vulnerabilities in the usage of loading external code with a static analysis approach, but they do not further analyze the security implications of the loaded binaries. Our code analysis is its superset, which includes five security-related aspects and allows us answer the 3 critical questions missing there. Our dynamic analysis framework allows us to precisely investigate the provenance, entity, and content of DCL. It additionally reveals that the loading of malicious code is triggered by properties of the runtime environment,

such as system time and network connectivity. Falsina et al. [**58**] propose a code verification protocol and a drop-in library to reduce the vulnerabilities in DCL, which is complementary to our study.

Program Analysis. RiskRanker [**75**] determines that DCL is taking place by static analysis. Its Dalvik code execution scheme is not able to analyze code loaded from sources other than local package, e.g. remote fetch. Zhang et al. [**161**] introduce a learning-based approach to analyzing the dependency of dynamic network requests. Crowdroid [**38**] is deployed in a crowdsourcing manner to detect Android malware using dynamic analysis. Because it applies low-level system call interposition, the analysis is not fine-grained due to the loss of context in Android middleware. Specially, it cannot differentiate the bytecode in the original application with that additionally loaded. TaintDroid [**57**] tracks the taint propagation at runtime, which aims at privacy leakage detection. Its implementation is based on DVM modification, which thus cannot handle native code. DroidBox [**53**], which combines TaintDroid and modifications of Android's code libraries, is able to log sensitive events at runtime, such as file read/write, loading class through `DexClassLoader`. It shares the same limitation with TaintDroid on analyzing native code. Some other dynamic analysis approaches can be adopted in our measurement [**35, 158, 132**], which reconstructs both low-level OS-specific and high-level Android-specific behaviors. Those methods introduce heavy latency in behavior reconstruction. Our approach intercepts the dynamically loaded code, and passes it to the cheap and efficient static analysis.

3.7. Conclusion

The unpredictability of DCL challenges the related security and accountability analysis. We build the system DyDroid, which is capable of intercepting DCL events and saving copies of the loaded bytecode and native code. We conduct a large-scale measurement of the DCL component of over 46K apps to investigate three critical questions missing in previous studies: (1) provenance, which includes the code's remote/local availability, and responsible entity; (2) app hardening, where DCL is used for the purpose of app obfuscation; (3) security risks/implications, which contains the malware detection, vulnerability analysis, and privacy tracking analysis. The apps that are found to use DCL in the remote fetch manner show that there is no existing solution to the enforcement of the related content policy. DCL is mainly used by third-party SDKs, indicated that the developer may not be aware that it is occurring. Given its stealthiness, DCL is also a channel to deploy malware, and we observe the real samples where the actual loading is controlled with logical conditions, such as system time. The security verification of DCL is needed from the app developer and OS vendors, given the apps vulnerable to code injection, which load binaries writable by other parties.

CHAPTER 4

AppShield: Enabling Multi-entity Access Control Cross Platforms for Mobile App Management

4.1. Introduction

Bring your own device (BYOD) enterprise policies have been growing in popularity. Employees use their personal devices to access an enterprise's proprietary resources. According to the survey by RCR Wireless News in 2015 [**2**], 85% of respondents indicated BYOD was incorporated into their organization's current telecom offering. The popularity of BYOD represents both an opportunity and a challenge. On the one hand, it boosts productivity and reduces the cost of dedicated devices. On the other hand, using the same device for both business and personal activities incurs new security threats, such as data exfiltration and revenue loss due to lost devices, employee job hopping, and malware. For example, considering the threat of malware alone, both Android and iOS have been reported to be affected by malware or low-reputation content [**97, 114, 140**]. Used in a BYOD setting, infected devices could threaten the confidentiality and integrity of business data. The concept of Mobile Application Management (MAM) is thus proposed to secure the BYOD utilization. Specifically, MAM solutions are the software and services that control access to enterprise resources at the mobile application level.

Android and iOS have discretionary access control to isolate data among apps. Regarding data sharing, Android provides the world read-/writable external storage, and

iOS maintains a similar directory `/Documents/Inbox/`. The system default data sharing/isolation mechanisms are insufficient for the complicated scenario of BYOD, given the numerous inter-app information flows from various entities. We also investigate existing BYOD commercial solutions (in Section 4.3.1), studies on information flow control [**157, 57, 112, 117, 110**] and application virtualization/sandboxing [**27, 96, 163**]. The following issues are not addressed.

- **Portability.** Many existing studies have been proposed to secure privileged resources in the enterprise environment [**96, 136**], but they are rarely adopted by vendors. Users have to get the customized firmware in deploying the security extension on their devices; this may not be possible because most devices have locked boot loaders and even in cases where this is technically possible, users may lack the right skills. The fragmentation issue of Android is another dominant factor that hinders the solutions with customized OS dependency from deploying in large scale. A recent report [**16**] showed 599 distinct Android brands with 11,868 distinct devices in 2013 and 18,796 distinct devices in 2014. Moreover, each of Android OS versions 2.3, 4.0, 4.1, 4.2, 4.4 has more than 10 percent of the worldwide market share. A solid MAM solution should not have any OS-specific requirement, e.g. version, firmware, to bolster the portability.
- **Multi-entity management.** Given a device, parallel data access control among application sets of various business entities is essential in the scenario of external business partner collaboration. For example, when a consulting company works closely with multiple clients simultaneously, it requires privileged data from those companies. The data sharing within each company's application set should be

orthogonal. Existing BYOD solutions cannot address this issue because they only support bisecting the apps on device into the personal set and the business set.

- **Role-based access control (RBAC).** Role-based access control (RBAC) [**135, 136, 138**] associate permissions with roles and users are made members of roles. It eases access management and is especially beneficial to large organizations like financial and medical institutions.

  While some operating systems (such as Android 5.0 and above) offer multi-account based management, the approach is not as flexible and lacks multi-entity management and RBAC support. We believe a BYOD solution should provide greater flexibility to enterprise policy administrators with respect to these aspects.

- **Fine-grained access control.** More stringent privacy laws have recently imposed new levels of confidentiality on health care and insurance companies, and financial institutions. Existing solutions do not have the policy enforcement flexible enough to secure high-credential data. In a solid solution, the data access among apps is controlled at a file level. For example, a user can share normal attachments received via email to `Dropbox`, but for a patent document with high-credential, any file sharing app's access can be blocked.

To resolve these problems in existing MAM solutions, we take the approach of application rewriting and provide it in a fully implemented prototype AppShield with the consideration of portability, which is able to enforce arbitrary access control policies with

no dependency of OS. AppShield includes two parts: (1) application rewriting framework for Android platform, which builds MAM features into an app, (2) cross platform proxy-based data access mechanism, which is able to enforce arbitrary access control policies.

The application rewriting framework automatically converts a personal app to the business version with almost no developer support. Specifically, the application using AppShield does not need to be developed in a certain way w.r.t storing/accessing documents. We hook into the `libc` [**13**] to capture all file system system-call related calls and those relevant to Android content provider [**15**]. This design enables AppShield to achieve complete mediation. AppShield protects privileged data access through the stealth channels: (1) native code, (2) dynamic code loading [**121**], and (3) Java reflection. The interposed low-level system calls can reliably intercept the privileged data request from the application level in all these scenarios. While we provide our proxy-based data access mechanism for both platforms, the application rewriting is available for Android only due to the closed-source nature of iOS. Nonetheless, with a little developer support (such as using an "AppShield" SDK), it is possible to provide iOS support.

The proxy-based data access mechanism is implemented within a controller application. Then we transparently proxy the data requests through our own controller that manages the applications' file-system-level data, content provider data and enforces access control policies. Apart from portability, the novel design of decoupling policy enforcement from OS also brings the benefit of cross platform. With the idea of data request proxy, we implement the fully functional controller application on iOS platform.

The AppShield Android app[1] has been released on both Google Play in North America, and Myapp in China. Our contributions are:

- We design a proxy-based data access mechanism that does not need OS support to enforce arbitrary access control policies, including those like MAC/SELinux [**142**] also. It is easily extended to other platforms, which is implemented on both Android and iOS.
- We investigate applying our proxy-based data access mechanism to Android MAM. The system prototype supports the configuration/enforcement of four types of security policies. *File isolation.* The privileged files of business apps are isolated from personal apps. *Multi-entity management & RBAC.* Apps can be divided into an arbitrary number of logical sets. It is further utilized in modeling RBAC, with orthogonal intra-set data access and multicast security policy update. Although we are not the first to apply RBAC to Android platform [**135, 136**], we propose a novel design without OS modification to boost portability. *Fine-grained file access control.* To provide special protection on high-credential data, the access control policy could be defined at file-level granularity. *Content provider isolation.* Other than managing the privileged structured data in system content provider, the data requests from the business apps are redirected to a private mirror content provider. For example, the business contacts are hidden from the personal apps.
- Our evaluation shows that AppShield has low overhead in memory, runtime, and package size and that it can reliably rewrite a large number of apps.

[1] `https://play.google.com/store/apps/details?id=com.webshield.appshield&hl=en`

The remainder of this section is organized as follows. Section 4.2 presents a brief background. Next, we cover the problem statement and AppShield design in detail in Section 4.3, followed by the implementation aspects in Section 4.4. Section 4.5 deals with the evaluation of AppShield. We have the relevant discussion and related work in Sections 4.6 and 4.7. Finally, we conclude our work in Section 4.8.

4.2. Background and threat model

Background. Android apps are implemented in Java, which is compiled down to Dalvik bytecode. It is also possible to use native code in apps. Android runtime environment enforces the sandbox mechanism to separate running apps. An app is assigned a unique user identifier (UID), by which the Linux kernel enforces discretionary access control (DAC) on low-level resources. Specifically, each app holds a private directory to keep the data in the internal storage, which cannot be accessed by any other app. The middleware further offers a permission system [**17**]. An app is granted permissions during installation. Apart from the pre-defined permissions guarding the system services, an app can define its customized permissions to restrict the access to their own components: *Activities*, *Services*, *Content Providers* [**15**], and *Broadcast Receivers*. Android includes content providers to control the access to a structured set of data.

3 types of MAM solutions have been proposed for BYOD.

- *Application Rewriting.* This approach inserts management hooks into existing Android apps. It has the advantages that it requires no developer collaboration and that it is independent of the OS version. However, it fails on apps that have been protected by anti-decompilation techniques.
- *Software Development Kit (SDK).* MAM vendors provide software development kits (SDK) for developers to incorporate into their apps. This approach has the disadvantage that developers must build and distribute two versions of the same app, and users' choice of business apps is limited to the markets.

- *OS Modification.* MAM features are directly built into the OS, so it neither requires developer collaboration nor can be defeated by anti-decompilation. However, since it relies on OS customization, the portability is limited.

In the case of application rewriting, third-party BYOD services are deployed with enterprise mobile marketplace. The client company selects useful general app, and BYOD vendor generates the enterprise version. Application rewriting requests reverse-engineering the personal app. With developer's cooperation in an enterprise setting, the developers can be asked not to apply anti-decompilation techniques, and either the developer's certificate or the unique certificate generated by BYOD vendor can be used to sign the business app under the agreement. Thus, app update can be easily managed in a timely manner.

Permissions are associated with roles, and users are made members of appropriate roles. Compared with the traditional group-based access control that only involves a set of users, using the role concept to bridge the user set and the permission set largely simplifies management of permissions and brings extra semantics in access control, which is valuable in the scenario of MAM.

Threat Model. On the device, both personal apps and business apps are installed. The personal apps may contain malware, which is able to access and leak the privileged data to untrusted servers. Moreover, for the data owned by an enterprise, other companies are motivated to track it.

OS level protection sacrifices the portability. Considering Android fragmentation, a solution without portability cannot fulfill the needs of BYOD, where employees utilize their diverse personal smartphones for business usage also. We agree that our defenses can be compromised if a device is rooted. Root is however too strong a threat model. Only

hardware or hypervisor-based solutions can ensure defense against superuser attacks. OS-level defenses remain vulnerable. Furthermore, a lot of modern devices are not rootable by any known means, meaning our defenses can offer complete protection.

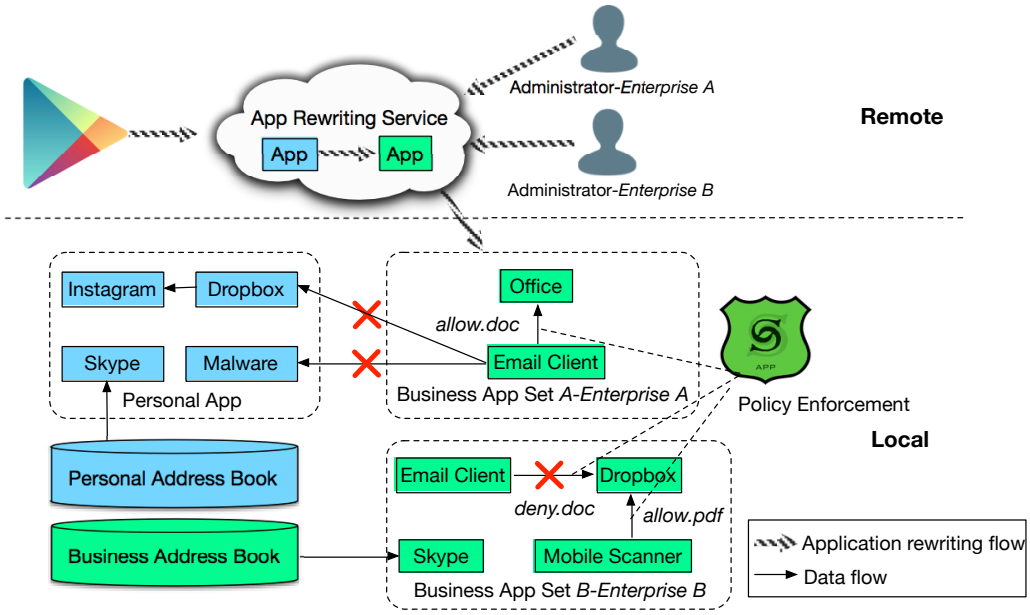


Figure 4.1. Security model

4.3. System Design

4.3.1. Problem Statement

Security Model. The security model of AppShield is depicted in Figure 4.1. An employee may install both personal and business apps on her device. A personal app may be any app that the user wishes to install, including possibly malicious apps. The business app, however, is issued by the IT administrator, who grants business apps as follows. First, he selects any off-the-shelf app from a mobile marketplace that is useful for his organization and submits the request to the BYOD vendor. Then, BYOD provider vets it using existing malware detection systems, such as [**24, 75, 129**]. Finally, the app is converted into business version and deployed in the enterprise mobile application marketplace after getting the agreement from the application developer.

Personal apps share data by existing mechanisms, such as content provider and public external storage, on Android. For example, `Instagram` posts the photos managed by `Dropbox`. Business apps share corporate data using the mechanisms provided by AppShield. AppShield manages a secure space where all the business data are maintained and security policies can be dynamically configured and enforced at file-level granularity as the tuple:

$$Policy = (\texttt{App\_S}, \texttt{Obj}, \texttt{App\_R}, \texttt{D}), \tag{4.1}$$

where `App_S` and `App_R` are the apps to share and receive the data, `Obj` is the object to be shared, and `D` is the decision made. When the `Office` app, for example, opens a document "`allow.doc`" from the business `Email Client`, AppShield validates the identity of the `Office` app, verifies against the security policy, opens the attachment file, and provides the business version of `Office` with the file descriptor of the opened file, whereas the app `Dropbox` could not access the file "`deny.doc`" owned by `Email Client` due to the policy violation.

As for multi-entity management, business apps from different companies installed on a device can be classified into various logic sets by the IT administrator. Given the flexibility and simplicity of management, RBAC is introduced to model the capabilities assigned to the user through the user-role review phase. Specifically, in Figure 4.1, the business app set A represents that a user is assigned the role holding the permissions to check the email and edit attached enterprise document belonging to enterprise A. The business app set B grants higher privilege to the user and allows the access to the address book and scanned document shared via the cloud service of enterprise B.

Table 4.1. Comparison with existing MAM solutions

Method	System	Isolation	Multi-entity management	RBAC	Granularity	Sharing	Portability
Rewriting	**AppShield**	**Sandbox**	**Yes**	**Yes**	**File-level dynamic**	**Local**	High
	AirWatch [**4**]	Sandbox	No	Yes	Static	Online	
	Mocana [**109**]	Sandbox	No	No	Static	Online	
SDK	Good [**72**]	Sandbox	No	No	Coarse dynamic	Online	High
	Citrix [**41**]	Sandbox & Encryption	No	Yes	Static	Local	
	AirWatch	Sandbox	No	Yes	Static	Online	
OS modification	Android L	DAC	No	No	Coarse dynamic	Local	Low

System Overview. Our system is organized into two parts: (1) an application rewriting framework for Android platform as the back-end that converts a personal app from mobile markets to a hardened business version by injecting MAM functionalities; (2) a front-end mobile app for both Android and iOS platforms that enforces the security policies with our proxy-based data access mechanism.

Table 4.1 lists existing MAM solutions on corporate data isolation/sharing and access control. The leading MAM vendors, except Citrix [**41**], fail to support local privileged data sharing, which requires the network connection and reduces the usability. Given the lack of fine-grained access control, these solutions are not able to provide special care of data with high-credential. All of the existing MAM solutions listed in Table 4.1 only bisect apps into the business set and the personal set. AppShield supports classifying the installed apps into an arbitrary number of groups, which enables multi-entity management. Some current BYOD systems provide RBAC support, but they deploy the access control module on the server side handled by their own administrators, which is not feasible in managing

the data from multiple companies on the same device due to the lack of communication channel among IT administrators. Our solution jointly considers role modeling and multi-entity management.

To our best knowledge, Bring Android to work [**36**] deployed on Android 5.0 and above is closest to our framework but it still fails to satisfy all the requirements listed in Section 4.1. This system is implemented at the operating system level. It divides the external storage into two directories: `/storage/emulated/0/` for personal apps and `/storage/emulated/10/` for business apps. The two versions of an app run with different UIDs. The data in one directory is only publicly accessible and shareable by apps from the corresponding set.

On Android L, we found that enterprise data could be shared among them without proper regulation. Because Android L only enforces DAC at the root directories of the two application sets, the fundamental data sharing mechanism of authorized apps remains the same with general personal apps. When a privileged file is shared via file system, it goes through the public storage that is readable by other business apps, and the only difference is that data exchange is in the business root directory. It is not capable of setting up multiple business application sets, and thus neither the multi-entity management nor the fine-grained access control is supported.

Given our radically different design and methodology from existing studies, we summarize the following challenges:

- **Lack of OS support.** The existing Android storage mechanism can only support either data isolation by private internal storage or data sharing by the system-wide read-/writable external storage or by content providers. Previous

work, such as TrustDroid [**163, 93**], Maxoid [**157**], Aquifer[**110**], and DR BACA [**136**], need to modify Android middleware to achieve the domain-level data isolation or permission regulation, which strongly reduces the portability. Thus, it is non-trivial to enable allocating a selective set of apps privileged data access permission without OS modification and root privilege.

- **Diversity of data access behavior.** Developers could utilize a diverse set of methods to access privileged data. We need to abstract the data access behavior to completely enforce the data isolation/sharing policies.
- **Performance penalty.** Some previous studies employ virtualization-based approaches to provide isolation between private and corporate domains [**30**]. Such methods do not scale well on the resource-constrained mobile device. Moreover, deep virtualization reduces the battery lifetime given the duplication of complete OS.

4.3.2. Application Rewriting Framework

The developer can either call the OS API based on the framework interface written in Java or directly invoke the native libraries. All the OS-level API invocations go through libc, which then makes system calls into the kernel. The libc layer provides us with a reliable point that abstracts all the complex high-level data access requests. Overwriting the entries in the global offset table (GOT) during the dynamic linking procedure allows us to inject our hooks to monitor the app's data access behavior and enforce our security policies. Details of this application rewriting method were discussed in Aurasium [**156**].

We do not claim the application rewriting design as our contribution, but rather our investigation on its usage in data access control.

Android apps are distributed in APK, which is a JAR archive including compiled Java source files in Dalvik bytecode, compiled manifest file, resources such as layout, images, and native libraries. We first unpack the APK file and decompile the dex bytecode to an intermediate representation (IR) `smali` [**141**] to enable our modification on bytecode. Our rewriting modifies 3 parts of application:

- **Native code.** We implement our customized system call hooks in C/C++ to monitor the privacy-sensitive behavior, such as `open()` and `rename()` for file access and `ioctl()` for data exchange via the content provider. Java code cannot modify process memory space, so we include the native code to overwrite the GOT with the address of our detour hooks whenever any ELF file is loaded. Moreover, business apps have frequent communication with AppShield, which includes information such as the identifier of business app to enforce security policies, and we thus implement the communication via the socket in the native layer for the latency performance.
- **Manifest file.** Android OS has the process `zygote` to initialize all the apps. When an app is running, its runtime environment is established. To enable GOT overwriting in ELF file, we modify the Manifest file to wrap the target app with our preprocess procedure. Specifically, we inject a service into the app that invokes the native code to modify the GOTs of all the loaded ELFs, and the preprocess procedure is configured in the parent class of the whole target app to guarantee it is running in the middle of `zygote` initialization and the

start of the app. Moreover, AppShield front-end app manages the security policy repository set by the IT administrator and enforces the security policies that grant the app the access to privileged data. Thus, we need to declare the *Activities* in the manifest file, which are injected into the target app's bytecode to popup UI message about the violation of secure policies. Regarding the data sharing/isolation of content provider, we create a mirror content provider in the private internal storage of AppShield and guard it with a special permission. Therefore, if a business app needs access to this content provider, it must declare this permission in the Manifest file.

- **Bytecode.** We modify the bytecode to configure the preprocess procedure in the parent class of the app. For example, `class A` is the child class of `class B` whose parent class is `android.app.Application` [**11**]. Then we replace the parent class of `class B` with our injected service. The *Activities* showing UI message are written in Java, compiled and converted to Dalvik bytecode.

We then compile the IR into the rewritten version of bytecode and repack the app into an APK file. An app needs to be signed, but rewriting invalidates its original signature, and AppShield cannot sign the rewritten app using its original private key. The signature is mainly used for identifying the developer. Moreover, app updates require the new version of each app to be signed with the same private key as the old version. AppShield can achieve these functions by signing apps originally signed with same keys with same (but new) keys.

AppShield is deployed as a remote service and generates a random private key to sign each business app. When the app is installed, the client side AppShield keeps the

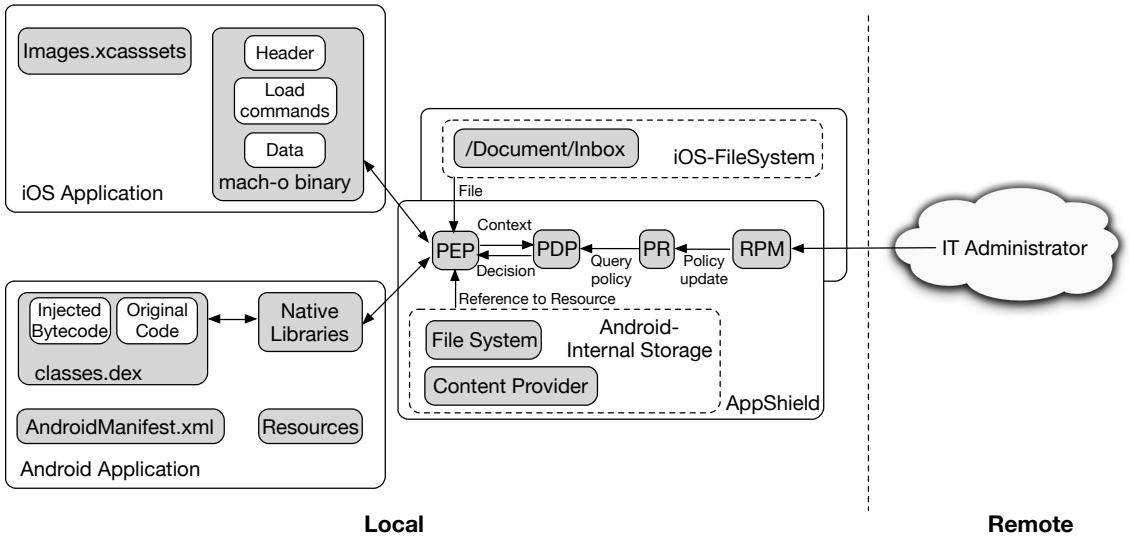


Figure 4.2. Proxy-based data access mechanism

mapping from the package name to its signature, which is used to differentiate business apps and personal apps. Due to the physical isolation of signature generation and the one-to-one mapping of original keys to new keys, it is difficult for an attacker to create a malicious app with the same signature as that of a legitimate business app to launch the privilege escalation attack. Our remote service can manage app update in the same way as mobile markets.

4.3.3. Proxy-based Data Access Mechanism

Figure 4.2 illustrates our proxy-based data access mechanism. In Android, any operation on privileged data via file system and content provider goes through our customized low-level system calls. The injected bytecode collects the context of the operation, such as the package name, signature, and data properties. The context is then sent to the **Policy Enforcement Point (PEP)**, which is implemented as a *Service* in AppShield and can be accessed by other apps through the socket in the native layer. On iOS platform,

the request of file operation carrying the app's identity and target file object is sent to PEP, which is implemented as a handler. The **Policy Decision Point (PDP)** decides whether the operation is allowed based on the context from PEP and the query results from the **Policy Repository (PR)** that could be remotely updated by IT administrator via **Remote Policy Manager (RPM)**.

4.3.3.1. Android. AppShield virtually maintains a file system and content providers in its internal storage. If data sharing is allowed, AppShield generates a reference to the data, which is granted to the business app. The business app indirectly operates on privileged data based on the reference to avoid creating duplicated data for the sake of performance, security, and synchronization. Data isolation is achieved, because the file system and the content provider are privately stored in the internal storage, and PDP validates whether the app requesting data operation is a business one; if so, application identity is further verified against security policy set.

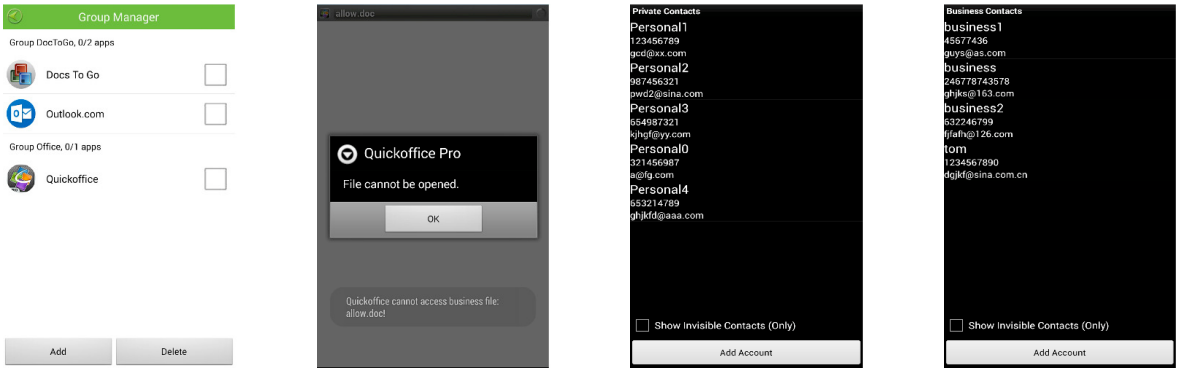
File-system. Wherever the original app stores the data, such as public external storage and privately accessible space, AppShield redirects the file operations from business apps to its own internal storage. We need to hook the following system calls:

- **open(), creat().** As an app invokes these two system calls, AppShield invokes the original system calls with a modified file path in the internal storage of AppShield and passes the flags and modes with a returned file descriptor.
- **rename(), mkdir(), remove().** The file paths in the parameters of these system calls are replaced with the business file paths in its internal storage.
- **stat(), lstat().** AppShield first gets a file descriptor to the business file in its internal storage and then invokes the `fstat()` to fetch the file status.

Content Provider. Content providers manage the access to a structured set of data, which is identified by `URI` [**18**]. Our proxy-based data access mechanism on content provider goes as follows:

- **Mirror content provider.** The core of content provider is the *SQLite* database. AppShield duplicates the target content provider with the same schema and table definition in its private internal storage. AppShield guards the mirror content provider with a special permission.
- **System call ioctl().** This is the main system call through which all binder IPCs are sent. By interposing on this system call, AppShield replaces the URIs to the original content provider with the URIs to the mirror content provider to redirect the data operation. Using context in this system call, AppShield validates who initiates the operations on the content provider, and the PDP module decides whether to allow the access. The malicious app thus cannot operate on the mirror content provider by the overwriting URI and permission declaration.

4.3.3.2. iOS. Given the closed source iOS, it is difficult to have the rewriting framework inject the MAM features into general iOS apps without developer support. However, we easily extend our proxy-based data access mechanism on iOS platform and implement the AppShield iOS client in `Swift`, which manages the virtual file system in its private space. The business app, which owns the privileged file, could *create* and *update* privileged file by sending it to AppShield's directory `Documents/Inbox/`. At the same time, AppShield records the mapping between the app's identity and the file object, which is expressed as `App_S` and `Obj` in Equation 4.1. The "Open-in management" feature, introduced from iOS 7 [**85**], allows AppShield to control which app the device uses to open a file. Thus,



(a) Application sets (b) Parallel access among application sets (c) OS contact provider access (d) Business contact provider access

Figure 4.3. Multi-entity management, RBAC & Content provider isolation

when an app `App_R` attempts to operate on the privileged file, AppShield validates the request against the policies in PR.

4.3.4. Security Policy

4.3.4.1. File isolation. The file-related operations from personal apps to business apps are strictly prohibited. All the files owned by business apps are kept in the internal storage of AppShield client app, which is invisible to all the other apps. When an app initializes the file operation request, the package name bound with its signature are sent to AppShield, which verifies whether it is a business app against the record in a database. It is extremely challenging to evade this security check because it requires the attacker to get the mapping relation between package name to app signature, which is constructed on the remote server side and securely stored in the private space of AppShield client side.

4.3.4.2. Multi-entity management & RBAC. Given the business apps from different companies, IT administrators can set up multiple app sets, where the union of the apps set's functionalities represents the permissions granted to this role (set). After a business

app is pushed and installed on the device, it is assigned to a business app set following the configuration made by IT administrators, which can be dynamically adjusted on-the-fly. Once the business identity of the app requesting file access `App_R` is verified, AppShield would further check whether there is an app set including both the owner of the target file `App_S` and `App_R`. If the two apps are not grouped into the same set, the file operation will be denied, which thus guarantees the orthogonal data access among roles. The example is illustrated in Figure 4.3a and Figure 4.3b, where one app set includes email client `Outlook`, document editor `Docs to Go`, and another set consists of the app `Quickoffice`. When `Quickoffice` tries to open the file *allow.doc* as an attachment in `Outlook`, the request is denied because the policy maintains the parallel access among different roles.

4.3.4.3. Fine-grained file access control. Android Lollipop allows all the requests across the business apps. In contrast, AppShield's file sharing is managed at file-level granularity for the apps in the same set. Given the sender app `App_S`, the receiver app `App_R`, and the file object `Obj`, AppShield checks the corresponding security policy in its repository, whose default value is `Allow`. This mechanism enables more flexible access control in protecting the high confidential file.

4.3.4.4. Content provider isolation. Business app conducts operations on the mirror content provider. If the app's identity is verified, the cursor of the mirror content provider will be returned, or AppShield will assign the app with the reference to the system content provider. This design guarantees the isolated operation on data in system default content provider and business privileged content provider. Note the example app in Figure 4.3c and Figure 4.3d, with the behavior of accessing the system's address book, the enterprise app fetches the business contacts in mirror content provider.

4.4. Implementation

We leverage the existing open source tools `apktool` [**10**] to unpack, decompile, and repack the app. We implement our customized system calls in C/C++. The open source tool AXML [**12**] allows us to modify the Manifest file at ease. The activities used to popup warning message are implemented in Java and those `.class` files are converted to bytecode using `dx` included in Android build tools. We also implement a script in `Python` to rewrite the bytecode in IR.

Android has 3 system content providers: contact provider, SMS provider, and calendar provider. The proxy-based data access mechanism is currently implemented on the contact provider. The calendar provider and SMS provider could be extended easily with small engineering efforts. For the content providers of third-party apps, our solution interposes on the system call `ioctl()` and blocks the operation when the app managing the content provider and the app accessing the data are from different sets.

Table 4.2. 35 file-related applications

Package name	**Isolation**	**Multi-entity management & RBAC**	**File-level granularity**
com.pixatel.apps.filemgr	√	√	√
cn.wps.moffice_eng	√	√	√
com.aor.droidedit	√	√	√
com.dataviz.docstogo	√	√	√
net.appositedesigns.fileexplorer	√	√	√
com.ImaginationUnlimited.instaframe	√	√	√
com.joodioapps.DocToPdf	√	√	√
com.lyrebirdstudio.mirror	√	√	√
com.mail.emails	√	√	√
com.majedev.superbeam	√	√	√
com.microsoft.skydrive	√	√	√
com.outlook.Z7	√	√	√
com.outthinking.textonpic	√	√	√
org.devgiant.project.zipfileextracter	√	√	√
com.sketchpicture.pictutreeffect	√	√	√
com.taxaly.noteme	√	√	√
com.thomasgravina.pdfscanner	√	√	√
com.ToDoReminder.gen	√	√	√
com.youthhr.phonto	√	√	√
cz.awk.android.docconv	√	√	√
joa.zipper.editor	√	√	√
jp.ne.shira.csv.viewer	√	√	√
net.daum.android.solmail	√	√	√
com.acr.sdfilemanager	√	√	√
com.sapparray.docmgr	√	√	√
com.jellydog.freereader	√	√	√
com.olivephone.office	√	√	√
vn.esse.WordToText	√	×	×
couchDev.tools.DocxParser	√	×	×
com.qo.android.am3	√	√	√
com.probcomp.filexplorer	√	√	√
com.seeke.pdfreader	Crash		
com.topnet999.android.filemanager	√	√	√
com.nimblesoft.filemanager	√	√	√
com.infraware.office.link	Cannot rewrite		
Succeed	33/35	31/35	31/35

4.5. Evaluation

We evaluated AppShield on a Samsung Galaxy Nexus with 4.3 Jelly Bean and an iPhone 5s with iOS 8.1.1.

4.5.1. Security Policy Enforcement

We selected 50 apps from Google Play to evaluate the effectiveness of our proxy-based data access mechanism. These apps have common business functions, such as email, file-sharing, document editing/viewing, and contact management, which were classified into two sets by the type of sensitive data operation: (1) 35 file-related apps, and (2) 15 contact provider-related apps.

We first used AppShield to convert these 50 apps to business versions. Then we manually interacted with these apps. Only one app can not be rewritten due to its obfuscation, which crashed the reverse engineering toolchain during unpacking, decoding, and repacking. One app crashed after rewriting. Even if we just decompiled and repacked the app without any code modification or injection, this app still crashed, which is probably attributed to the usage of repackage-detection techniques, e.g. integrity verification.

We then tested each file-related app against three security policies. Specifically, whether the file owned by the business app was isolated from personal apps and business apps from another group; whether the request from other business apps in the same group can be allowed and blocked according to the configuration. The results are listed in Table 4.2. Two apps cannot enforce the security policies regarding multi-entity management and fine-grained access control. After investigating the reason through application reverse-engineering, we found that these two apps looked up files with the path starting with "/./sdcard", which was not considered when being converted to paths in the private space of AppShield and thus the business files cannot be located.

The 15 contact provider-related apps were evaluated on content provider isolation. We checked whether each app loaded data from the system contact provider before rewriting

Table 4.3. 15 contact provider-related applications

Package name	Isolation
com.appyown.contactsbackuprestore	√
com.globile.mycontactbackup	√
com.idea.backup.smscontacts	√
com.ijinshan.kbackup	√
com.mofinity.ui	√
com.payneservices.LifeReminders	×
com.tos.contact	√
net.IntouchApp	√
com.actimust.simplecontacts	√
com.netqin.contactbackup	√
no.uia.android.backupcontacts	√
com.xuecs.ContactHelper	√
digiteria.backup	√
nexg.contactbackup	√
com.brainworks.contacts.cuteblue	√
Succeed	14/15

Table 4.4. Large-scale evaluation on 1000 applications

Total Apps	Succeed	Cannot be rewritten	Crashed
1000	953(95.3%)	12(1.2%)	35(3.5%)

and from the mirror contact provider as the business version. The results are abstracted in Table 4.3. One app failed in policy enforcement. Unlike the normal case where app loaded the address book data from contact provider, this app indirectly used `Intent` to start the system contact manager app. Our solution does not have the control over system apps.

Across the 120 times of policy enforcement (3 for each file-related app, 1 for each contact provider-related app), our mechanism achieves the success rate 109/120 (90.8%). The general reason for the failure is that our implementation does not consider developer's specific pattern of API invocation. e.g., the path of the privileged file.

4.5.2. Reliability

For the test on the reliability of AppShield, we picked top 250 apps by popularity on Google Play in September 2015 within the following categories: `Business`, `Finance`, `Medical`, and `Productivity`. We used AppShield to convert these 1000 apps to their business versions, and then automatically ran the apps using the UI/Application Exerciser Monkey [**148**]. The results are shown in Table 4.4.

12 apps failed during rewriting because their obfuscation crashed the reverse engineering tools `apktool` in unpacking, decoding, and repacking. While we acknowledge that AppShield cannot reliably rewrite apps with anti-reverse engineering techniques, our large-scale test shows that the percentage of these apps is still low. Also, developers are actively improving the reverse engineering tools that AppShield relies on. For the 35 rewritten apps that crashed during execution, we ran their original versions and found 29 of them also crashed, which clearly were not caused by AppShield. To investigate the reasons why the remaining 6 rewritten apps crashed while their original versions did not, we just unpacked and repacked them without modifying their code or data, and found all of them still crashed after repacking. We hypothesize that they might use anti-repacking techniques, such as signature validation. We performed these tests on real-world apps without developer support. In an enterprise MAM situation, however, it is reasonable to assume that the MAM provider can work with the developers so as to enable successful rewriting of their apps. Developers have strong incentive to work with MAM providers as this allows their apps to be used across entire enterprises.

4.5.3. Impact of Application Rewriting

4.5.3.1. Latency. We evaluated AppShield's performance by both *micro-benchmark* and *macro-benchmark*. We implemented a test app that opens files and loads data from contact provider. Moreover, we developed an iOS app that can delegate the permission of accessing its private files to a selective set of apps. Given the closed nature of iOS, we could not modify the invocation of low-level system calls and hence cannot build an application rewriting framework. For evaluation, we implemented the proxy-based data access mechanism inside the app. Even though our rewriting framework is not cross platform, our proxy-based data access mechanism is. We expect that with reasonable developer support, our solution is still feasible on iOS platform.

- **Micro-benchmark.** We conducted a stress test with 1000 data access operations to investigate the latency introduced by AppShield. First, we recorded the accumulated time spent on getting the file descriptor on Android and getting the file contents from the iOS AppShield client with and without our security policies enforced. Because we cannot dig into low-level system calls of closed-source iOS, we measured the time of loading file contents on that platform. We also measured the total time of fetching the cursor, which is a reference to the content provider. Only the operations that we benchmarked contain the latency introduced by AppShield for policy enforcement, and the further operations on data remained the same with the unmodified app.

  The results are listed in Table 4.5. In the worst case, AppShield introduced an overall latency of 0.202s on Android file system during 1000 operations, because acquiring each file descriptor involves one round of IPC with AppShield. For

Table 4.5. Runtime latency introduced by AppShield

	File System				Content Provider	
	Android		iOS		Android	
	Original	AppShield	Original	AppShield	Original	AppShield
Micro-benchmark×1000 (s)	0.180	0.382	0.171	0.347	7.303	9.014
Macro-benchmark (s)	1.472	1.524	1.643	1.753	1.068	1.194

the performance on iOS, AppShield introduced a latency of 0.176s. AppShield introduced a latency of 1.711s when getting the cursor of a content provider. Since IPC is the dominant factor in the latency and has a fixed cost, the relative latency decreases, as the original operation takes longer.

- **Macro-benchmark.** We asked one user to manually load data via the file system and contact provider on the smartphone. We recorded the time from when the user started to access the data until when she closed the app after the data was fully rendered on screen. The user performed a series of data access operations for 5 times with and without AppShield. Table 4.5 shows the average of time. AppShield introduced a latency of 52ms, 110ms, and 126ms in data operations on Android file system, iOS file system, and Android content provider, respectively. Such latency is barely perceptible. Although user experience on application response might not be accurate to the order of millisecond and there is a slight difference in each round of manual operation, we try our best to simulate user's daily usage manner.

4.5.3.2. Memory consumption & Code size. Figure 4.4 shows the cumulative distribution function (CDF) of the overhead in memory usage and code size caused by rewriting. To eliminate the side effect of Android garbage collection when calculating memory usage, we used the tool `dumpsys` in Android Debug Bridge (adb) to get the maximal memory

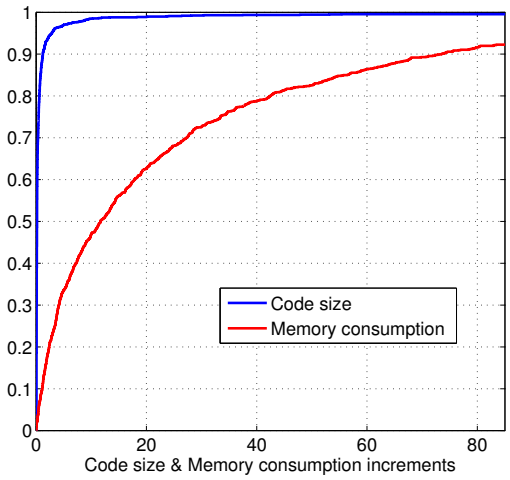


Figure 4.4. Code size & memory usage overhead (CDF)

usage during the execution of an app. To eliminate the side effect of compression during app packing, when calculating code size, we sum up the customized native libraries, Manifest file, and bytecode.

AppShield's rewriting introduced less than 5% code size increment in over 95% apps, and more than 85% apps incurred the memory usage overhead less than 60%. The average overhead was 28840.3KiB in memory usage and 33.7KiB in code size. Our system hooks into the low-level system calls, and the dynamic linking naturally supports the efficient memory utilization by avoiding code duplication. Moreover, we add our customized system calls, and the classes for UI notification just once rather than inlining them at every point where the original app accesses privileged data.

4.6. Discussion

AppShield does have some limitations because of its current implementation. Our rewriting mechanism involves unpacking the APK file and decompiling the dex bytecode to IR. App developers sometimes use anti-reverse engineering techniques to crash decompilation tools to protect their intellectual property. Moreover, when the IT administrators conduct the security verification on the apps to be selected as business ones, the obfuscated app may challenge the correctness of the verification. However, our large-scale evaluation shows that the percentage of these apps is low. Moreover, the app developer could be asked not to apply such tools, where tiny developer support is needed. Developers are often willing to work with enterprises as this offers them a large high-payoff user base.

Another limitation is that it depends on hooking on the dynamically-linked `libc`. Any system call invoked not via the system `libc`, such as by using a statically-linked libc, will bypass our hooking mechanism. The chance of this happening is very low, and can be detected statically. Regarding the iOS platform, it is extremely hard to automatically rewrite apps and hook those system calls, given its closed-source nature. However, the proxy-based data access mechanism is cross-platform, which is implemented as a client iOS app leveraging the "Open-in management" feature.

4.7. Related Work

Virtualization & Sandboxing. L4Android [**96**] combines the L4Linux and Google modifications of Linux kernel to enable executing Android OS on top of a microkernel. Running multiple Android OS instances in parallel on the same device enables the complete isolation but has high performance penalty. TrustDroid [**163**] addresses the performance issues. It introduces the logical domain isolation approach, where two single domains are considered and isolation is enforced as a data flow property between the logical domains without running each domain as a single virtual machine. Boxify [**27**] constructs virtual sandboxes to secure Android apps, but the decision on which app to be isolated relies on manual identification. We model the data access control problem in the scenario of MAM, and app identity is classified by its business/personal purpose. These approaches fail to consider the data-sharing problem to give a fine-granulated control that grants a selective set of apps the access to privileged data.

Rewriting. Davis et al. [**46**] rewrite the Dalvik bytecode to allow interposing on security sensitive APIs. Retroskeleton [**45**] supports the retrofit of app's behaviors by static and dynamic method interposition. These approaches are based on the high-level API interposition, and thus, they cannot completely enforce the security policies across all layers of Android framework. Aurasium [**156**] adopts the design most similar to us that provides reference monitor capabilities by repackaging Android apps to use a customized version of libc. AppShield extends the usage of this application rewriting technique with the proxy-based data access mechanism to achieve data access control, and multi-entity management. Similarly, ASM [**80**] provides a programmable interface for API hooking, which can also be leveraged to implement user-level access control.

RBAC. Vaidya et al. [**150**] propose RoleMiner to assist automatic role construction following a learning approach. Previous studies mostly focus on the general modeling of RBAC. Rohrer et al. [**135, 136**] further investigate the specific RBAC problem when using Android device in sensitive environment, such as finance and health, but the mechanism involves the modification of system middleware and lacks a system prototype to be evaluated.

4.8. Conclusion

We present the proxy-based data access mechanism, which can enforce arbitrary access control policies. Given the critical issues of MAM, our prototype system AppShield achieves multi-entity management and RBAC at file-level granularity, apart from privileged data isolation from personal apps and corporate data sharing across business apps. We implement it on both Android and iOS platforms to demonstrate its cross platform property. Our design has neither dependency on OS nor the root privilege, which thus has good portability. AppShield is successful at policy enforcement with low latency and is reliable.

CHAPTER 5

RiskCog: Unobtrusive User Identification on Smartphones in the Wild

5.1. Introduction

Smartphones provide users with various functionalities, among which the popularity of mobile payment is growing exponentially. Based on the report by eMarketer [**56**], 37.5 million users are expected to use mobile payments in the year 2016. Although mobile payments have benefitted users immensely, they also introduce several security threats. Among them, human-driven risks, arise when parties other than the owner have access to the smartphone and utilize the payment functions for their own benefit. According to the report by LexisNexis [**68**], for the 15% of merchants accepting mCommerce payments in 2014, mobile transactions accounted for 14% of the total transaction volume, and 21% of the volume of fraudulent transactions.

Human-driven risks still lack effective countermeasures. The traditional user authentication mechanism only verifies whether the user knows the account credential set up previously at the start of using the service. Moreover, the login credential based authentication requires the explicit user action, for example, account/password input. Learning-based user identification approaches are proposed. They construct the model to describe the usage pattern of the authorized phone owner, such as the locations he/she frequently

Table 5.1. Comparison with related studies

Study	**Require user movement**	**Fixed/dynamic device placement**	**Scale (#Users)**	**Require label**	**Offline real-time verification**
RiskCog	No	Dynamic	>1,500	No	Low latency: 3237.7 ms
Lu et al. [**104**]	Yes	Dynamic	<50 (walking detector training), <50 (supervised training), <10 (unsupervised training)	No	High latency: 27863.586 ms
Derawi et al. [**48**]	Yes	Fixed	<100	Yes	No
Ho et al. [**81**]	Yes	Fixed	<50	Yes	No
Kwapisz et al. [**95**]	Yes	Fixed	<50	Yes	No
Ren et al. [**133**]	Yes	Fixed	<50	Yes	Not implemented
Related challenge	Lack of feature	Data availability & dynamic device placement	Imbalanced data set	Unlabeled data	Constrained mobile environment

visits and face snapshot. Comparing with the simple login credential, the user identification mechanism utilizes a diverse set of features to verify user's identity, and it is thus much harder to get bypassed. The learning-based approach can be applied to the scenario of payment, while it fundamentally is a new solution to the general services with the requirement of user identification. For example, Alice and Bob are close friends. Alice leaves her phone at Bob's home. Bob can check Alice's Facebook private activity without her consent, if the Facebook app's automatic login option is enabled. However, the learning-based approaches will detect the unauthorized user implicitly. The app provider or the end-user can customize the follow-up behavior, such as notifying the phone owner or rendering an empty page.

We aim at enforcing use identification at the device level. This generic service allows the detection results to be reused and removes the redundancy in terms of the data collected from each individual app. In our threat model, each smartphone has a unique owner, and attackers attempt to operate the phone. We only assume the acceleration sensor, gyroscope sensor, and gravity sensor are available on the device. In Table 5.1, we list the problems in previous learning based approaches and summarize the following challenges:

(1) Lack of feature. Although Android supports numerous sensors, which can be potentially used to fingerprint users, the fragmentation issue [**19**] hinders it from being deployed widely. Specifically, many sophisticated sensors are not available on some low-end devices. Moreover, a portion of sensors has to be integrated into an app to work, e.g., the pressure sensor needs to be bound to a view element within an app. A device level protection cannot have dependency on those sensors. Additionally, any feature involving user's privacy in the context of social impact is also not useable for the sake of privacy.

(2) Data availability & dynamic device placement. Only those data collected during daily usage are usable because fingerprinting user fundamentally depends on the user's specific pattern of handling the phone. Some users produce less training samples. For example, she/he might use the phone rarely, or prefer to keep the phone on a stationary plane, in which no motion event can be used to represent the authorized owner.

Previous studies [**48**] [**81**] [**95**] [**133**] also have the strong assumption of fixed device placement, e.g., in the pocket of trousers. To offer a full verification, we should not have any requirement of device placement and user's motion state. Moreover, the app-specific pattern will challenge the classification accuracy. The user's pattern dramatically varies

with different types of apps, e.g., the frequent typing gestures in a chatting app compared with the rotation in a race car game.

(3) Imbalanced data set. When identifying the authorized phone owner, we labeled the data of the authorized user as 1 and that of other users as 0. The resulting binary classification task is imbalanced as the number of positive examples is much less than that of negative examples. The imbalance ratio in our project increases when the system is applied to a larger scale of users.

(4) Unlabeled data. The proposed prototype systems [**48**] [**81**] [**95**] [**133**] introduce supervised learning algorithms for the well-labeled training set, for example, whether each data sample is generated when the authorized user is using the phone. However, the well-labeled data is not available in the practical scenario. For example, the device owner may give the phone to her/his family member during data collection.

(5) Constrained mobile environment. Leveraging the remote sever is limited by the availability caused by numerous factors, such as disconnected/weakly connected environment. The client-server model is not feasible given our intention to deliver the real-time user authentication. A complicated classification model with high prediction accuracy requires heavy computation resource. It is not suitable to the constrained mobile devices. To our knowledge, only the work by Lu et al. [**104**] has the offline verification. The complex gait analysis based on Universal Background Model (UBM) has high latency 13993.379 ms.

We design a system, called RiskCog, to provide the offline user identity verification service. It is based on our semi-supervised learning algorithm to identify the phone owner. Each data sample is collected, preprocessed, and further classified by whether the user

is in a steady or moving state. Two parallel classifiers are trained for these two states, which are used to predict whether the authorized device owner is using the phone. We made the following contributions:

- We find 56 features to generally identify smartphone device owner with motion sensors. Moreover, The feature set does not involve any in-app invocation or user privacy tracking. It is independent of user's motion state and device placement.
- We design a semi-supervised online learning algorithm, where the classifier is trained incrementally with the data collected in chunks. It eliminates the high time latency when handling unlabeled data with unsupervised methods. By checking the consistency among the data sent to server incrementally, we can filter out the part that is not coherent with the authorized owner's fingerprint.
- We develop an unobtrusive user identification mechanism with cross platform capability. We decouple the verification from the server side and resolve the issue of availability. Our optimization of learning algorithm produces a light-weighted identity verification service on resource-constrained mobile devices. It only takes 3237.7 ms to finish the verification.

We achieve high accuracy for unobtrusive user identification with wild data collected from an industry mobile payment app by one of the biggest mobile service vendors in the world. In our evaluation on 1,513 users, RiskCog achieves the classification accuracy values of 93.77% and 95.57% for the steady state and moving state, respectively. We also release an Android app[1] and an iOS app[2] including our user identification mechanism.

[1]RiskCog Android client app. `http://139.224.207.24/SensorDemo-release.apk`
[2]RiskCog iOS client app. `http://139.224.207.24/riskcog-ios.zip`

The remainder of this chapter is organized as follows. Section 5.2 presents a brief background of our work. In Section 5.3, we cover RiskCog design in detail, which is then followed by an explanation of the implementation procedures in Section 5.4. Section 5.5 deals with the overall evaluation of our system. Section 5.6 and 5.7 include discussion about relevant work and Section 5.8 includes our concluding remarks.

5.2. Background

5.2.1. Authentication, and user identification

Authentication is used to prevent the unauthorized parties from using the sensitive services. Currently, the credential is the predominant form of an authentication system. It is known to have many security problems. First, it is only able to verify if the user knows the credential rather than recognize whether she/he is the owner of the device. The credential-based authentication is thus vulnerable to dictionary attacks. A recent report about data breaches [**42**] shows that 4.1% of users choose "123456" as their passwords, and 79.9% of apps still accept weak (lower-case only letters) passwords. Moreover, the credential-based authentication cannot enforce the full protection and achieve usability at the same time. A fully on-demand verification requires a user's explicit input action, every time the user accesses the sensitive services. While if the user uses the functionalities, such as automatic login, out of the concern of usability, the identity will only be verified once.

Considering the security issues of the traditional login credential based authentication, user identification introduces learning-based approaches. It is able to exactly describe the authorized device owner by a diverse set of features. At the time of verification, the system will predict the probability that the user who attempts to access the service is the owner of the device by checking the alignment between the collected test samples with the trained model. It is much harder for the attackers to bypass the verification compared with the traditional authentication mechanisms, given the difficulty of mimicking a legitimate

user's patterns. Moreover, it requires no explicit actions from the user, which can enforce the on-demand protection without sacrificing the usability.

5.2.2. Privacy

The privacy preserving property of RiskCog is defined in the context of social impact. Previous studies found the feasibility of fingerprinting mobile devices with motion sensors [**50**] [**44**]. However, device tracking does not imply the identity of its owner in social life. We only know the mapping between a trained model and an authorized device owner. However, it is unable to further figure out who the device owner is. Compare with the motion sensor data, other types of features involved are more sensitive. It is straightforward to know who the user is when face recognition is utilized for the purpose of authentication [**151**] [**8**]. As for geo-location, it is able to identity the owner in the physical world as stated in previous studies [**33**] [**71**] [**32**] [**82**] [**94**].

5.2.3. Platform porting & sensor availability

Our user identification only requires the acceleration sensor, gyroscope sensor, and gravity sensor. Thus, RiskCog can be easily migrated to various mobile platforms (e.g., Android and iOS with leading marketshare). Moreover, we survey the required sensors' availability on 11 types of devices with high market penetration by 6 major smartphone vendors. All the three motion sensors are available on the popular devices surveyed, except Samsung Galaxy CORE Prime that was released in 2014.

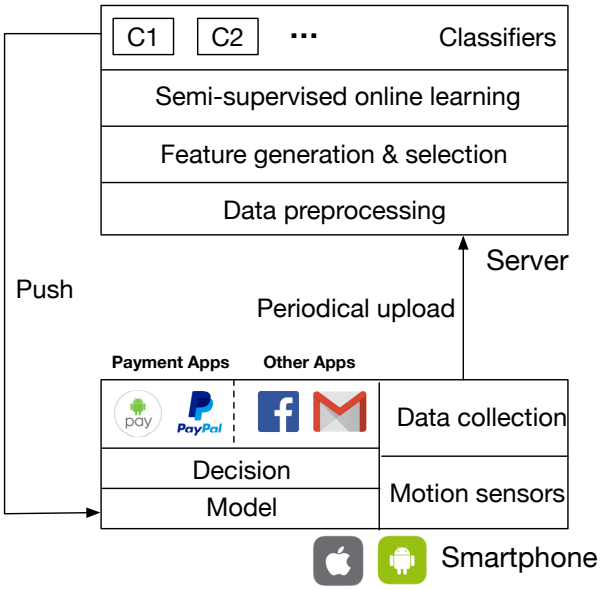


Figure 5.1. System architecture; training phase starts at data collection from motion sensors and ends at the classification model is pushed to the device followed by the local identification verification.

5.3. System Design

The architecture of our system is illustrated in Figure 5.1. The client app deployed on the device periodically collects data from the acceleration sensor, gyroscope sensor, and gravity sensor. Those data are incrementally uploaded to the server. After preprocessing, the training set is constructed, which includes both the data from the target device labeled as 1 and the data from other devices labeled as 0. Our semi-supervised online learning algorithm is based on the assumption that most of the data uploaded from one device originate from the authorized owner. We thus have a well labeled training set for the negative instances collected from other devices. But it is infeasible to get reliable labels for positive instances from the device owner because the device may be used by the owner's friends during data collection. However, our semi-supervised online learning mechanism

does not require the explicit label. It cognitively detects and filters the data not aligning with the user's pattern by checking its statistical consistency with historical data. The specific usage manner of each phone owner will be modeled as the classifier. When the classifier is fully trained, the trained model will be pushed to the device. As the mobile payment vendors receive the transaction requests from the smartphone, they just need to locally query whether the phone is being used by the authorized owner. Compared with the deployment model of existing learning-based user identification, the unified solution of RiskCog eliminates the cost where each mobile payment vendor maintains its user fraud detection mechanism in the backend. Moreover, those data used for identification are uploaded only once rather than be sent to each vendor's server separately.

5.3.1. Data collection and preprocessing

RiskCog collects data from acceleration sensor, gyroscope sensor, and gravity sensor. Each sensor reading includes values corresponding to the x, y, and z axes:

$$\{X_a(k), Y_a(k), Z_a(k)\}, \{X_{gy}(k), Y_{gy}(k), Z_{gy}(k)\}, \{X_{gr}(k), Y_{gr}(k), Z_{gr}(k)\}.$$

Here, the parameter k represents the k-th acceleration, gyroscope and gravity reading in the time dimension.

We develop a mobile app for data collection. It needs to detect the duration when the device is being actively used. Moreover, we observe that the sensor readings largely vary with different types of apps even for the same user, which will affect the classification accuracy of the trained model. We demonstrate the findings in our experiment with three types of apps. We note that those three apps are purely for the experimental purpose to

indicate the difference among user gestures in playing with various apps. A user keeps rotating his/her phone when playing a race car game driven by the acceleration sensor; a lot of typing gestures are generated when using a chatting app; the device is stable when a news app is used. However, the sensor readings are relatively consistent during the start of an app, in other words, the loading phase.

We have a `BroadcastReceiver` to capture the system event where the screen of a device is turned on, and then it starts a `Service` [**21**] that periodically queries the current app in the foreground. When the currently active app is different from the one in the last query, we will recognize that a new app is started. The data collection will keep running for 3 seconds if both of the two conditions are satisfied: (1) the screen of the phone is on, (2) a new app is running in the foreground. The data are thus collected during the active daily usage and the application-specific pattern is also filtered. Because our data are only collected from motion sensors, the effect of other interference factors, such as voice is negligible.

Our preprocess includes the data calibration and motion state recognition. A user may not actually hold the phone during daily usage. We thus observe that a portion of data is ineffective to reflect the difference among various users' patterns, even if we apply the two conditions above in the data collection stage. Our data calibration phase has the following condition regarding the gravity sensor values in three dimensions, and it allows RiskCog to remove the data in those situations, such as keeping the device flat on a desk. We have a participant to handle the phone and put a phone on a stationary plane. Then we get the boundaries of the gravity sensor readings on three dimensions by

minimizing the errors of device placement prediction.

$$\{-1.5 < X_{gr}(k) < 1.5\} \cap \{-1.5 < Y_{gr}(k) < 1.5\} \cap \{9 < |Z_{gr}(k)| < 10\}$$

After removing the data samples that are ineffective to represent the pattern of device owner, we project those sensor readings to our global coordinate system, which allows RiskCog to be insensitive to device orientation. We first identify the gravity direction based on the values read from the gravity sensor on three dimensions, whose opposite direction will be set as the z-axis in the global coordinate system. It is thus straightforward to decide the remaining x-axis and y-axis by Fleming's right-hand rule.

The usage pattern differs to a large extent when the user is moving as opposed to when the user is steady. If we use one classifier for all the motion states, there will be huge inconsistency within the data samples of one user that will affect the classification accuracy. A classifier is thus trained for each state of a user.

We observe that the difference between the values of acceleration sensor and those of gravity sensor (D-value) in the moving state has a higher amplitude compared to the D-value in the steady condition. We define the k-th D-values on three dimensions as:

$$X_d(k) = |X_a(k) - X_{gr}(k)|, Y_d(k) = |Y_a(k) - Y_{gr}(k)|, Z_d(k) = |Z_a(k) - Z_{gr}(k)|.$$

We then get the median values $\tilde{X_d}(k)$, $\tilde{Y_d}(k)$, and $\tilde{Z_d}(k)$ on three dimensions in the data collection duration. With a predefined threshold, the user's motion state is thus classified as steady given the condition below:

$$\sqrt{\tilde{X_d}(k)^2 + \tilde{Y_d}(k)^2 + \tilde{Z_d}(k)^2} < \delta.$$

5.3.2. Feature generation and selection

For the classification, we only utilize data collected from acceleration and gyroscope sensors. Since standard classification methods cannot be directly applied to time-series data, we first extract the feature vectors from the raw time series data. To fulfill this, we divide the raw time series data into 0.2-second segments and extract features based on the 10 sensor readings within each segment. We denote the ith value of the feature vector as $\mathcal{F}_i = \{F_{1i}, F_{2i}, \cdots, F_{pi}\}$, which includes p features. In order to maintain the consistency among feature vectors in the temporal domain, we utilize sliding window design with 50% overlap between each pair of neighbor segments, i.e.

$$\{X_a(k), Y_a(k), Z_a(k), X_{gy}(k), Y_{gy}(k), Z_{gy}(k)\}_{k=1}^{10} \quad \Rightarrow \quad \mathcal{F}_1$$

$$\{X_a(k), Y_a(k), Z_a(k), X_{gy}(k), Y_{gy}(k), Z_{gy}(k)\}_{k=5}^{15} \quad \Rightarrow \quad \mathcal{F}_2 \cdots$$

According to our experiment, if the length of sliding window is too long, it will result in a loss of accuracy. Meanwhile, if the length is too short, It's time-consuming to process the raw data.

We generate a total of 56 features, which covers multiple moments and other commonly used statistical properties of the distribution. We separately utilize each potential feature to establish a classification model and evaluate the accuracy of these 56 models given the ground truth in the laboratory settings. It verifies the effectiveness of each single feature to depict the device owner's pattern. We also rank features based on their independent classification capabilities, which is further utilized in our stratified sampling.
(1) Mean: $\sum_{k=1}^{K} x(k)/K$, (2) Standard Deviation: $\sqrt{\sum_{k=1}^{K}[x(k) - \bar{x}]^2/(K-1)}$

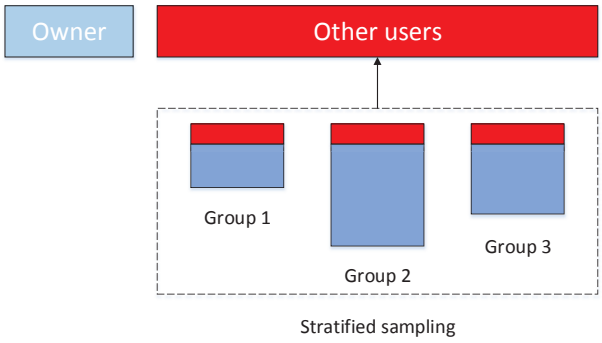


Figure 5.2. Training set construction

(3) Average Deviation: $\sum_{k=1}^{K} |x(k) - \bar{x}|/K$, (4) Skewness: $\sum_{k=1}^{K} [(x(k) - \bar{x})/\sigma]^3/K$

(5) Kurtosis: $\sum_{k=1}^{K} [(x(k) - \bar{x})/\sigma]^4/K - 3$

(6) Lowest value: $\min\{x(k)\}$, (7) Highest Value: $\max\{x(k)\}$

(8) Cross zero rate: $\sum_{k=0}^{K-1} ||sgn[x(k+1)] - sgn[x(k)]||/K$

(9) RMS Amplitude: $\sqrt{\sum_{k=1}^{K} [x(k)]^2/K}$

(10) Average root sum square: $\sum_{k=1}^{K} \sqrt{x^2(k) + y^2(k) + z^2(k)}/K$

Here, K equals 10 for our case. Replace x in the first 9 formulas with $X_a(k)$, $Y_a(k)$, $Z_a(k)$, $X_{gy}(k)$, $Y_{gy}(k)$, $Z_{gy}(k)$ respectively and they will render us 54 features. The last formula provide us with 2 features from 2 sensors. In total, 56 features are extracted and used for the classification purpose.

5.3.3. Semi-supervised online learning algorithm

Training set. During the training phase, data are collected to generate the feature vectors. Each user has n feature vectors, denoted by $\mathcal{F}_i, i = 1, \cdots, n$. For p phone users, $n \times p$ vectors can be used to train the classifier all together. The sample size refers to the

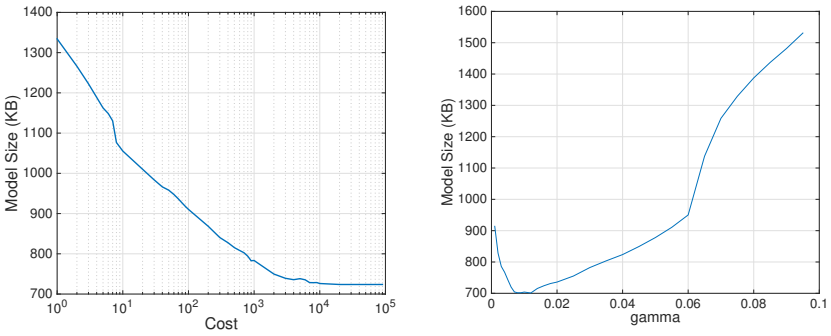
number of feature vectors n. Treating the data set of the authorized user as Class 1 and that of all the other users as Class 0, we has a highly imbalanced data set as p is large.

We employ the stratified sampling to handle this problem [**147**], which groups the vectors by one feature value. According to the feature selection ranking result, the average root sum square of acceleration (ARSSA) readings is the most important feature for classification. Therefore, we carry out stratified sampling based on these feature vectors of all the other users. We also observe that the temporal continuity of sensor reading is actually helpful to depict the authorized owner's pattern of handling the device. Our sampling method thus needs to keep this property. To be specific, we select the 1st, 100th, 200th,... ARSSA values for each user, sort all $n \times (p-1)/100$ values and divide them into 5 equal size strata. Then, an equal amount of samples is randomly drawn from each strata. To preserve the time consistency, each chosen sample along with 99 samples after it are all selected to form the negative sample set. By doing so, negative samples including in the training set has better representativeness of the $p-1$ users. And the final model becomes more robust than simple random sampling. Regarding the number of the stratum, a larger number brings us a fine-grained sampling, in other words, a stronger capability of representing other users. However, it introduces higher latency.

Our training set is constructed as the Figure 5.2. The ratio of the number of samples by the owner to that of other users is 1:5, which can be properly handled by normal learning algorithms. The optimal point is chosen by experiment.

Classification method. We choose Support Vector Machine (SVM) with radial basis function (RBF) kernel as our classification method for those reasons:

(1) Nonlinear classification boundary. Our problem is not linearly separable. SVM



(a) C (logarithmic scale, $\epsilon = 0.01$, $\gamma = 0.01$)

(b) γ ($\epsilon = 0.01$, $C = 100$)

Figure 5.3. Model size v.s. C and γ (Accuracy $\geq 90\%$)

with a nonlinear kernel helps us build a proper classification boundary.

(2) Comparatively high dimensional space. We have a comparatively high dimensional feature space. SVMs have been reported in many studies to work better with our situation [**154**] [**89**].

(3) Dependent/correlated data. Our features are extracted from motion sensors and the readings on three dimensions are inevitably correlated, given the nature of human activity. SVM does not explicitly assume feature independence.

Optimization. The size of model is essential when verifying the user identity offline. Considering the limited computational resources of mobile devices, a smaller model indicates the lower CPU and memory consumption in identity verification and lower traffic when pushing the model to the smartphone. Moreover, the size of model is related to the number of support vectors in SVM learning algorithms. Thus, the phase of optimization also avoids overfitting. We conduct the grid search to find the optimal configuration of the parameters C and γ in the SVM with RBF kernel. We choose 80 users, who generate most number of data samples in daily usage, from our data set with 1,513 users that will

be explained in detail in Section 5.5. Given the fixed parameter $\epsilon = 0.01$, we change C from 1 to 90,000 and γ from 0 to 0.1. As shown in Figure 5.3, we find the model size decreases with the value of C. As the value of C exceeds 100, the system will get tiny decrement in the model size (cost C is depicted in the logarithmic scale). The model size will reach the minimal value when the value of γ equals to 0.01.

Our semi-supervised online learning algorithm is illustrated in Figure 5.4. The data samples are uploaded to our server in chunks. One chunk is split into two parts for both training and testing purposes. The new training data and other users data are used to construct a training set. The online learning module takes the old classifier and training set as inputs and produces a new classifier. The new classifier does a validation on the test samples from the data chunk and other users data. The old classification accuracy and the new one are represented as α_{old} and α_{new}, respectively. The condition to commit the new classifier is expressed as:

$$\lambda\alpha_{new} + (1 - \lambda)\alpha_{old} > \alpha_{old} - \beta.$$

The parameter λ ranging from 0 to 1 is the factor to quantify the weight of new classification accuracy. The value β is the threshold to represent the normal performance variation rather than that caused by data inconsistency.

The authorized device owner may share her/his phone to others, such as friends and family members. We have no idea of the label ground truth. Incorporating the noisy data in the classification model would affect the prediction accuracy. Our design of commit/rollback allows RiskCog to detect the misalignment and filter the noisy data.

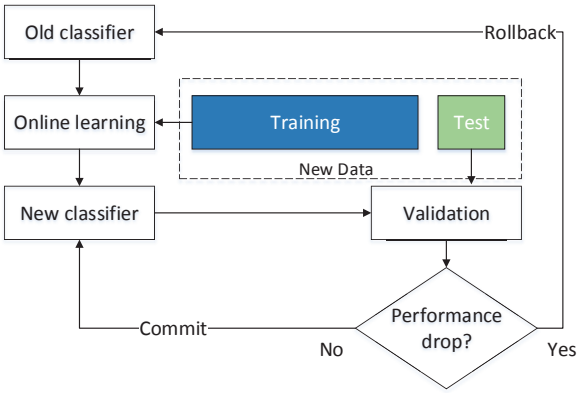


Figure 5.4. Semi-supervised online learning; each chunk of data is committed if there is no classification accuracy drop and the training finishes when the classification accuracy values are stable across chunks.

Another problem is to identify whether the classifier is fully trained. We mark the model as ready for identification with the condition:

$$\alpha_{new} > A \text{ and } Var(\alpha) < V.$$

In our commit/rollback design, we have the classification accuracy for each chunk of data. When the latest prediction accuracy is higher than the threshold A and the variance of all the accuracy values of those chunks which are accepted previously is smaller than V, we will identify the classifier training is finished. This implies that the performance converges to a stable state.

Decision. Given a vector for one sliding window in duration $w = 0.2$ second, the classifier outputs the probability if the owner is using the phone p. If it is larger than the decision threshold θ, we will identify the user as the owner.

5.4. Implementation

Our data collection scheme involves verifying the active device screen and the presence of applications in the foreground. We implement the `BroadcastReceiver` [**131**] to capture the system event where the device screen is turned on. Android provides the two APIs `getRunningAppProcesses()` and `getRunningTasks()` to retrieve the current application running in the foreground. However, starting from Android 5.0, those APIs are deprecated and cannot return the information of other applications. The list of running apps can be alternatively fetched by using `UsageStatsManager` [**149**]. However, this requires users to grant application the permission in system settings. To preserve the portability of RiskCog on the fragmented Android devices, we invoke the system command line tool `ps` and implement a parser to map the running `pid` to the application name. Our implementation allows us to intercept the active applications properly on all the existing versions of Android without any permission.

We implement the data preprocess module in C++. It is intended to filter the data which are ineffective to fingerprint user's pattern and distinguish the two motion states. We use `LibXtract` [**99**] to extract the 56 features. We train our model with `LibSvm` [**98**] on the server side and enforce the offline identity verification with `AndroidLibSvm` [**22**] on the Android platform and `LibSvm` on the iOS platform. Overall, We implement RiskCog with over 5,000 lines of code in C++ and 2,000 lines of code in Java.

5.5. Evaluation

Our data set includes two parts. We define the ground truth as the knowledge whether the device owner is used the smartphone. For the experimental data with ground truth, we have 10 participants to use the same phone for one day. Each participant generates 9240 samples by handling the phone in both steady and moving states. For each user, we split the data samples into the training set and the test set. The ratio of the number of samples in the training set to that in the test set is 4:1. In the training set, the ratio of the number of samples from the owner to that of other users is 1:5. The test set follows the same distribution. Moreover, we also have the labelled dataset provided by the Internet company `Ant Financial` [**107**] for the accuracy benchmarking test, which is generated by 30 participants in the steady state from the iOS platform.

For the raw data without ground truth, they are directly collected from the product by an Internet company `Tencent` [**145**] with millions of users. In our collection scheme, the collection frequency is 50Hz. We collect data from 1,513 different users for 10 days. Each data collection phase lasts 3 seconds. For the sake of traffic usage and battery consumption in the production environment, there are at most 20 data collection phases (60 seconds) in one hour. The IMEI is the user identifier. We note that a portion of the data collected will be filtered, which are ineffective to fingerprint the user (e.g., phone is put on a stationary plane). The distributions of training and test sets are identical to the experimental data mentioned above. The following metrics are used in our evaluation.

True positive (TP). The authorized owner is correctly identified.

False positive (FP). Other users are incorrectly identified as the owner.

False negative (FN). The authorized owner is incorrectly identified as other users.

True negative (TN). Other users are correctly identified.

Performance & Overhead. We first evaluate the time latency of training on the server side. Then we check the battery consumption, CPU, and memory usage of the phone when our client app collects data and verifies the user's identity.

The classification performance of RiskCog is depicted with the following values: precision for phone owner P_{owner}, recall for phone owner R_{owner}, precision for others P_{other}, recall for others R_{other}, and classification accuracy.

$$\begin{aligned} P_{owner} &= TP/(TP+FP), R_{owner} = TPR = TP/(TP+FN) \\ P_{other} &= TN/(TN+FN), R_{other} = TN/(TN+FP) \\ FPR &= FP/(FP+TN), Accuracy = \frac{TP+TN}{TP+FP+FN+TN} \end{aligned}$$

The ROC curve reflects the overall performance of RiskCog. It shows the true positive rate against false positive rate with various classification threshold θ.

The training module is deployed on the server side with `LibSVM`, where both the positive samples and negative samples are available. As discussed above, our evaluation of accuracy also involves both positive samples and negative samples and is conducted on the server. In the architecture of RiskCog, the user identity verification on the Android platform is enforced offline with `AndroidLibSvm`, where only the data generated from the device (positive sample) are available. We verify that both `LibSVM` and `AndroidLibSvm` produce the same prediction result given the identical prediction model and data sample as input. It is thus valid to utilize our evaluation of accuracy to depict the effectiveness of RiskCog to enforce user identity verification locally.

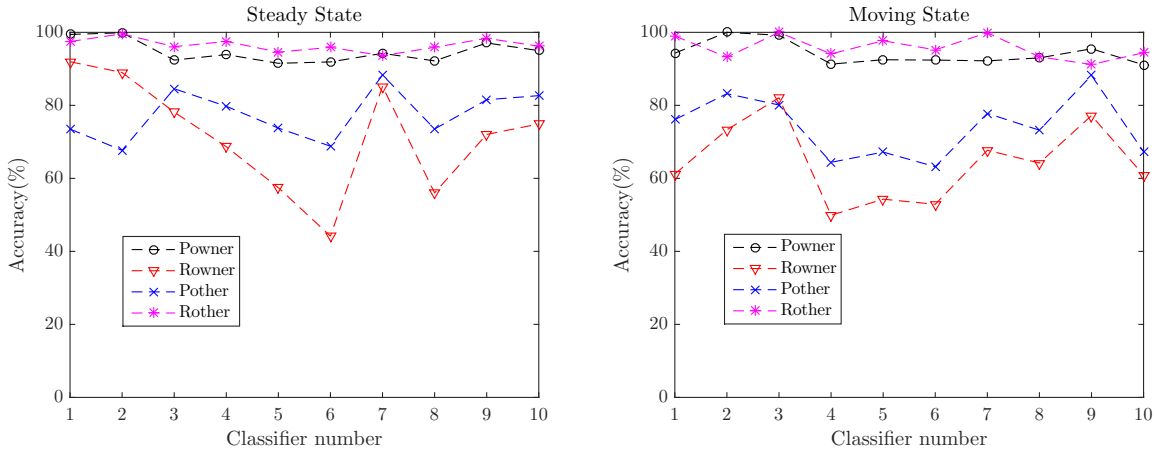


Figure 5.5. Accuracy on experimental data with ground truth for 10 participants

5.5.1. Accuracy

Batch learning - experimental data with ground truth. Because the experimental data are labeled, our online learning algorithm is not needed in this scenario. We simply train the classifier with the whole training set, where the classification threshold θ is set to 0.5. Regarding the configuration of SVM, we set the cost value as 100 and γ as 0.01. Figure 5.5 shows the classification performance of our system for users in the steady state and the moving state, respectively. All the trained classifiers for the 10 participants have the values P_{owner} and R_{other} higher than 90% in both of the two motion states. In particular, the average values of P_{owner}, R_{owner}, P_{other} and R_{other} are 94.76%, 71.76%, 77.41%, and 96.53% for the steady state. As for the moving state, the values are 94.15%, 64.37%, 74.07%, and 95.80%. The average accuracy for the steady state is 84.15%, and that for the moving state is 80.09%. The results indicate that RiskCog achieves the low false positives, while the number of false negatives is relatively high. In our training set organization, we set the ratio of the number of positive samples (owner) to that of

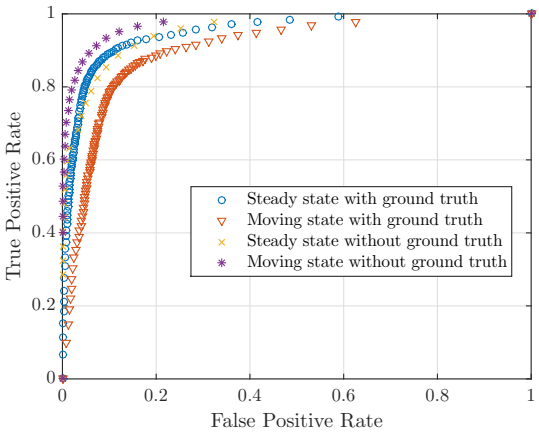


Figure 5.6. ROC curve for 10 participants with ground truth and 1,513 users without ground truth; decision threshold θ varies from 0 to 1 with step growth 0.01

negative samples (other users) as 1:5. It means the classifier can accurately recognize the unauthorized users' patterns, i.e. the illegal access, while some gestures of the authorized owner will be missed. In user identification, a false positive, i.e. the illegal access to the user's account is more critical than a false negative (false alarm). Thus, we pay more attention to restricting the false positives when configuring our system.

In Figure 5.6, we use the ROC curve to depict the true positive rate against the false positive rate at various threshold θ. It starts from 0 to 1 with step growth 0.01. Given the value of θ, we calculate the average values of TPR and FPR for all the participants. The areas of the two curves for moving/steady states are 0.9513 and 0.9043. RiskCog has enough space for tuning given various requirements of sensitivity and specificity.

For the 30 participants in the steady state from the iOS platform, we have the average values of P_{owner}, R_{owner}, P_{other} and R_{other} as 88.07%, 75.08%, 97.28%, and 98.87%.

Online learning - raw data without ground truth. In the training set, each user on average has 20,648 samples for the steady state and 9,280 samples for the moving state.

The training set will be divided into 10 chunks. Regarding the conditions of accepting a chunk of data and training termination (Section 5.3.3), we set the parameters λ, β, A, and V as 0.5, 0.1, 0.8, and 0.05, where we observe the average number of chunks taken to finish the online learning is 5.8.

In Figure 5.6, the areas of the two curves for the moving state and the steady state are 0.9719 and 0.9506 for all the 1,513 users without ground truth. The performance is slightly better than that in the laboratory setting. It is attributed to the bigger size of the training set and the stratified sampling applied, which allows the classifier to well differentiate the authorized device owner from others.

For all the 1,513 users in the wild, our system achieves the average values of P_{owner}, R_{owner}, P_{other} and R_{other} as 87.39%, 73.28%, 96.07%, and 98.43% for the steady state, and 89.35%, 81.41%, 97.81%, and 98.89% for the moving state. The average accuracy values for the two states are 93.77% and 95.57%. Even with those challenges in the practical deployment, such as imbalanced data set and unlabeled data, our design including the stratified sampling and semi-supervised online learning algorithm allows RiskCog to have the performance that is similar to that in the laboratory setting. The prediction accuracy for the steady state is slightly lower than that for the moving state, from which we can see that our feature set is nearly independent of user's motion state and RiskCog is able to provide the full protection on the user's account. All the previous studies [**104**] [**48**] [**81**] [**95**] [**133**] rely on the features, which are only available when the user is moving, such as step cycle.

Table 5.2. The overhead results on three different smartphones; the measurement of battery consumption lasts three hours.

		Data Collection		**Identity Verification**	
Phone Type	Battery Consumption (mAh)	CPU (%)	Memory (MB)	CPU (%)	Memory (MB)
Samsung N9100	132.5/3000	10.34	14.4	8.80	21.4
Sony Xperia Z2	113.8/3200	1.82	18.0	9.00	26.0
MI 4	128.7/3080	1.30	14.0	12.00	24.3

5.5.2. Overhead

We measure the runtime latencies of the training phase on the server side with an Intel Xeon E5 CPU and 64G of physical memory running on Ubuntu 14.04. On average, RiskCog is able to analyze the user's data for 10 days and train the classifiers for the steady and moving states within 148.36s and 21.21s.

On the client side, we utilize the tool Emmagee[3] to assess the impact on battery consumption, CPU, and memory usage. Emmagee can sample the hardware resource usage of an app on the device. Table 5.2 shows the results. For the battery consumption, we let one participant use the client app for three hours, which includes both data collection and offline identity verification. Only one percent of the battery is required by our app in one hour. The CPU usage is over 10% on the device Samsung N9100 during data collection. The case does not happen on other two phones. It is possible that the higher CPU utilization is related to the low-level system implementation. Our optimization of SVM setup reduces the resource consumption of the CPU-intensive offline verification.

We also check the latency of offline user identity verification. We execute the procedures: data collection, data preprocessing, feature extraction, and decision for 1000 times on the device Samsung N9100 and record the average time for each step. The results are

[3]Emmagee. `https://github.com/NetEase/Emmagee`

Table 5.3. Latency of offline user identity verification

Procedure	Time (ms)
Data collection	3211.6
Data preprocessing	0.5
Feature extraction	12.3
Decision	13.3
Overall	3237.7

listed in Table 5.3. We can see the whole process can be finished within 3237.7 ms. The latencies introduced by steps other than data collection are negligible.

5.5.3. Resistance to brute-force attacks

RiskCog verifies the user identity by a set of features collected from motion sensors, and the brute-force attacks in our scenario are thus based on a large set of sensor data generated by users other than the authorized user. We assess the robustness of RiskCog with two types of brute-force attacks.

Automatic attack. We implement a sensor data generator. It first randomly selects one dimension (x, y, z axes) that aligns with the gravity direction. The acceleration on that dimension is set randomly within a range around positive/negative gravity acceleration value. For the nine values collected from motion sensors, the starting points are generated around baseline values within predefined ranges, which we called initial deviation range. We define this point as the initial point. We observe that those data generated by human have the property of temporal continuity. Our random data generator also follows this rule, where the current slot differs from the previous slot by the small step deviation range. Moreover, the generated data is bounded to guarantee that they confirm to the laws of physics. In daily usage, users are possible to change their ways of handling the phone, which would break the continuity. We thus define a continuous interval, in which

the consecutive samples are continuous, and entering a new continuous interval involves the generation of a new initial point.

Given the trained classifiers of the 80 users who are randomly selected from the overall 1,513 users, we generate the fake data including 600K samples and check the percentage of samples which are correctly labeled as unauthorized. The average percentage of samples successfully blocked by our system is 90.44%.

Manual attack. We also have humans to launch the brute-force attack. One classifier is trained for the authorized device owner by fingerprinting the usage manner. 10 participants handle the same phone with various gestures for 40 times, where a participant generates 49 samples each time. Those samples are checked against the classifier, and we identify the percentage of samples which are correctly labeled as other users. RiskCog blocks the attacks by human with probability over 99.85%.

5.6. Discussion

We discuss two scenarios which may allow attackers to evade our detection. First, the attackers can hook those APIs related to motion sensors and manipulate the return values. RiskCog may always read the same values from sensors and the verification will be bypassed. However, hooking the system APIs requires the root privilege, which is a too strong assumption. We can also enforce root detection and deploy our service on those devices without being rooted.

Second, the data from motion sensors are publicly readable. Any app can fetch the data, reproduce our learning process, and deduce the usage manner of the authorized owner. We can manipulate the training set, e.g., interchanging features in the vectors: $< F_1, F_2, ... > \Rightarrow < F_2, F_1, ... >$, which is invisible to the attackers. When RiskCog attempts to identify the user, the verification phase will not be affected if the same rule of manipulating vector is applied.

5.7. Related Work

Gait recognition. The vision-based approaches [**113**] [**77**] [**102**] [**139**] were initially proposed to recognize human's gait. Acceleration sensors were found to be useful for gait identification [**66**] [**144**]. Coskun et al. [**43**] explored how much the device placement increased the accuracy of recognizing activities. These studies focus on classifying a user's gait, rather than verifying the identity.

Sensor-based user authentication. Kwapisz et al. [**95**] and Derawi et al. [**48**] made use of phone-based acceleration sensor to authenticate cell phone users. Ren et al. [**133**] proposed a user verification system leveraging the unique gait pattern derived from acceleration sensors to detect possible user spoofing in the mobile healthcare system. These approaches require that the sensors are placed in specified body locations and the samples in the training set are well labeled. Lu et al. [**104**] overcame these limitations by projecting the data samples onto a global coordinate system for the resilience to device orientation and handling the unlabeled data with an unsupervised learning algorithm, and achieved the offline user identity verification. However, it relies on user inputs to update the model and reduce false negatives. While our semi-supervised online learning algorithm not only requires no user action but also has a lower latency of handling unlabeled data in training compared with an unsupervised one. Moreover, with a radically different design without involving complex UBM and the extra step of feature extraction, we reduce the latency of offline verification by 90%. All the proposed solutions require the user's movement because the model depends on the features, such as step cycle. RiskCog can verify the identity, when the user is steady, simply by the manner of handling the device.

Feng et al. [**65**] investigated authenticating users with touchscreen gestures, where they built a sensor glove to collect data. Frank et al. [**67**] used the touchscreen input as features for user identity verification, and they implemented an Android app to capture the touch screen events because those data are only available at the application level. RiskCog does not depend on any external sensor and all those features used are on the device level.

5.8. Conclusion

We present the system RiskCog to provide the on-demand and offline user identity verification with a learning-based approach. The trained classifier depicts the owner's specific manner of handling his/her smartphone based on the data collected from motion sensors. Unlike previous related studies, we have no requirement on the user's motion state or the device placement. Plus the offline real-time identity verification that allows our system to be usable in the disconnected environment, RiskCog can protect user anywhere and anytime. As deploying RiskCog in the production environment on a large scale, we resolve several new issues, such as the imbalanced data set and training set without ground truth with our stratified sampling method and a semi-supervised online learning method. We achieve the classification accuracy values 93.77% and 95.57% among the 1,513 users for the steady state and the moving state.

CHAPTER 6

Conclusion

The privacy leakage has been a dominant security issue of the Android platform in recent years. I decompose the problem by the top-down architecture. In the layer of user interaction, I propose the system AutoCog, which relies on a learning-based approach to deduce the semantics model and helps the user understand the in-app privacy usage by the application description. The design has been extended to other usages, such as analyzing the application privacy policy and the texts in UI elements. In the layer of middleware, the trigger of privacy tracking behaviors can be controlled by the developer given the unpredictability of DCL, which bypassed the security check of mobile marketplaces at ease, such as Google Bouncer. I build the dynamic analysis system DyDroid to fully explore the DCL usage and detect the privacy leakage. Regarding the prevention of privacy leakage, all the solutions based on OS modification suffer the limited portability because of the fragmentation issue. AppShield hooks the calls in the native layer of Android with an application rewriting design. The proxy-based data access mechanism allows the enforcement of arbitrary access control policy without dependency on the OS. The system can be used to produce the enterprise version of mobile apps in the scenario of BYOD.

The smartphone stores user's privacy and also is used to access user's sensitive resource online, such as bank account. Those risks of privacy leakage are driven by the human. The traditional credential-based user authentication only verifies if the user knows the

predefined credential, which is easy to get bypassed. The explicit input of password has the tradeoff between its usability and the continuity of protection. With the proposed learning algorithm, the system RiskCog recognizes the authorized user by the manner of handling the device. To enable the protection in various challenging environments, the verification is successfully deployed on the resource-constrained mobile devices.

References

[1] Millions of Android smartphones are infected with malware - here's how to keep yours virus-free. http://www.mirror.co.uk/tech/millions-android-smartphones-infected-malware-8382366.

[2] 2016 Predictions: The year of BYOD management. http://www.rcrwireless.com/20160129/opinion/2016-predictions-the-year-of-byod-management-tag10.

[3] 360 jiagu. http://jiagu.360.cn/.

[4] AirWatch: Enterprise Mobility Management. http://www.air-watch.com/.

[5] S. Alam, R. N. Horspool, and I. Traore. Mail: Malware analysis intermediate language: a step towards automating and optimizing malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 233–240. ACM, 2013.

[6] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi. Droidnative: Semantic-based detection of android native code malware, 2016.

[7] Ali jaq. http://jaq.alibaba.com/safety/.

[8] Alibaba uses facial recognition tech for online payments. http://www.computerworld.com/article/2897117/alibaba-uses-facial-recognition-tech-for-online-payments.html.

[9] Allatori Java string renaming document. http://www.allatori.com/doc.html#properties-renaming.

[10] Android-apktool: A tool for reengineering Android apk files. http://code.google.com/p/android-apktool/.

[11] Android application class. http://developer.android.com/reference/android/app/Application.html.

[12] Android binary XML file parser. https://github.com/xgouchet/AXML.

[13] Android bionic. https://android.googlesource.com/platform/bionic/.

[14] Android Captures Record 81 Percent Share of Global Smartphone Shipments in Q3 2013. http://blogs.strategyanalytics.com/WSS/post/2013/10/31/Android-Captures-Record-81-Percent-Share-of-Global-Smartphone-Shipments-in-Q3-2013.aspx.

[15] Android content provider. http://developer.android.com/guide/topics/providers/content-providers.html.

[16] Android fragmentation report august 2014 - opensignal. http://opensignal.com/reports/2014/android-fragmentation/.

[17] Android manifest permission. http://developer.android.com/reference/android/Manifest.permission.html.

[18] Android URI. http://developer.android.com/reference/android/net/Uri.html.

[19] Android's biggest problem is operating system fragmentation. http://o.canada.com/technology/personal-tech/androids-biggest-problem-is-operating-system-segmentation.

[20] Smartphone os market share. http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

[21] Android service. https://developer.android.com/guide/components/services.html.

[22] AndroidLibSvm. https://github.com/yctung/AndroidLibSvm.

[23] Android Application Class. https://developer.android.com/reference/android/app/Application.html.

[24] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of NDSS*, 2014.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[26] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012.

[27] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. Von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *Proc. USENIX Security*, 2015.

[28] Baidu. `http://www.baidu.com/`.

[29] Bangcle. `http://www.bangcle.com/`.

[30] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, 2010.

[31] K. Benton, L. J. Camp, and V. Garg. Studying the effectiveness of android application permissions requests. In *IEEE PERCOM Workshops*, 2013.

[32] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive computing*, 2(1):46–55, 2003.

[33] C. Bettini, X. S. Wang, and S. Jajodia. Protecting privacy against location-based personal identification. In *Workshop on Secure Data Management*, pages 185–199. Springer, 2005.

[34] Bitdefender Antivirus Software. `http://www.bitdefender.com/`.

[35] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE, 2010.

[36] Bring Android to Work. `http://www.android.com/it/preview/`.

[37] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS Symposium*, 2012.

[38] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[39] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *ACM WWW*, 2012.

[40] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *ACM MobiSys*, 2011.

[41] Citrix. `https://www.citrix.com/`.

[42] You won't believe the 20 most popular cloud service passwords. `https://www.skyhighnetworks.com/cloud-security-blog/you-wont-believe-the-20-most-popular-cloud-service-passwords/`.

[43] D. Coskun, O. D. Incel, and A. Ozgovde. Phone position/placement detection using accelerometer: Impact on activity recognition. In *ISSNIP*, pages 1–6, 2015.

[44] A. Das, N. Borisov, and M. Caesar. Tracking mobile web users through motion sensors: Attacks and defenses. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[45] B. Davis and H. Chen. Retroskeleton: Retrofitting android apps. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 181–192. ACM, 2013.

[46] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *IEEE MoST, San Francisco, CA*, 2012.

[47] DBpedia - A Wikipedia based Natural Language Processing Database. `http://dbpedia.org/datasets`.

[48] M. O. Derawi, C. Nickel, P. Bours, and C. Busch. Unobtrusive user-authentication on mobile phones using biometric gait recognition. In *Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP), 2010 Sixth International Conference on*, pages 306–311. IEEE, 2010.

[49] Android DexClassLoader. `http://developer.android.com/reference/dalvik/system/DexClassLoader.html`.

[50] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and S. Nelakuditi. Accelprint: Imperfections of accelerometers make smartphones trackable. In *NDSS*. Citeseer, 2014.

[51] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011.

[52] Q. Do, D. Roth, M. Sammons, Y. Tu, and V. Vydiswaran. Robust, light-weight approaches to compute lexical similarity. *Computer Science Research and Technical Reports, University of Illinois*, 2009.

[53] Droidbox - android application sandbox. `https://github.com/pjlantz/droidbox`.

[54] DroidNative. `https://bitbucket.org/shahid_alam/droidnative`.

[55] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49:255–273, 2015.

[56] Mobile payments will triple in the US in 2016. `http://www.emarketer.com/Article/Mobile-Payments-Will-Triple-US-2016/1013147`.

[57] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010.

[58] L. Falsina, Y. Fratantonio, S. Zanero, C. Kruegel, G. Vigna, and F. Maggi. Grab'n run: Secure and practical dynamic code loading for android applications. In *ACSAC*, pages 201–210. ACM, 2015.

[59] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *ACM SPSM*, 2011.

[60] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, 2011.

[61] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *ACM SPSM*, 2012.

[62] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *USENIX WebApps*, 2011.

[63] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *ACM SOUPS*, 2012.

[64] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission redelegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[65] T. Feng, Z. Liu, K.-A. Kwon, W. Shi, B. Carbunar, Y. Jiang, and N. Nguyen. Continuous mobile authentication using touchscreen gestures. In *Homeland Security (HST), 2012 IEEE Conference on Technologies for*, pages 451–456. IEEE, 2012.

[66] J. Frank, S. Mannor, and D. Precup. Activity and gait recognition with time-delay embeddings. In *AAAI*. Citeseer, 2010.

[67] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *IEEE transactions on information forensics and security*, 8(1):136–148, 2013.

[68] LexisNexis True Cost of Fraud mCommerce. `http://lexisnexis.com/risk/downloads/whitepaper/true-cost-fraud-mobile-2014.pdf`.

[69] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *IJCAI*, 2007.

[70] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing*, pages 291–307, 2012.

[71] P. Golle and K. Partridge. On the anonymity of home/work location pairs. In *International Conference on Pervasive Computing*, pages 390–397. Springer, 2009.

[72] Good Technology. `https://www1.good.com/`.

[73] Google Play. `https://play.google.com/store?hl=en`.

[74] Google Play Developer policy. `https://play.google.com/about/developer-content-policy.html`.

[75] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys*. ACM, 2012.

[76] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 101–112. ACM, 2012.

[77] J. Han and B. Bhanu. Individual recognition using gait energy image. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(2):316–322, 2006.

[78] W. Han, Z. Fang, L. T. Yang, G. Pan, and Z. Wu. Collaborative policy administration. *IEEE TPDS*, 25(2):498–507, 2014.

[79] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the*

12th annual international conference on Mobile systems, applications, and services, pages 204–217. ACM, 2014.

[80] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *Proc. 23rd USENIX Security Symposium (SEC14)*, 2014.

[81] C. C. Ho, C. Eswaran, K.-W. Ng, and J.-Y. Leow. An unobtrusive android person verification using accelerometer based gait. In *Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia*, pages 271–274. ACM, 2012.

[82] B. Hoh, M. Gruteser, H. Xiong, and A. Alrabady. Enhancing security and privacy in traffic-monitoring systems. *IEEE Pervasive Computing*, 5(4):38–46, 2006.

[83] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *ACM CCS*, 2011.

[84] Ijiami. `http://www.ijiami.cn/`.

[85] iOS Open-in management. `http://searchmobilecomputing.techtarget.com/tip/Open-in-management-helps-secure-iOS-data`.

[86] Java Object HashCode. `https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()`.

[87] Java StackTraceElement. `https://docs.oracle.com/javase/7/docs/api/java/lang/StackTraceElement.html`.

[88] Java Native Interface. `http://developer.android.com/ndk/samples/sample_hellojni.html`.

[89] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

[90] M. G. Kendall. Rank correlation methods. 1948.

[91] K. Kennedy, E. Gustafson, and H. Chen. Quantifying the effects of removing permissions from android applications. In *IEEE MoST*, 2013.

[92] T. Kiss and J. Strunk. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525, 2006.

[93] P. Kodeswaran, V. Nandakumar, S. Kapoor, P. Kamaraju, A. Joshi, and S. Mukherjea. Securing enterprise data on smartphones using run time information flow control. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*, pages 300–305. IEEE, 2012.

[94] J. Krumm. Inference attacks on location tracks. In *International Conference on Pervasive Computing*, pages 127–143. Springer, 2007.

[95] J. R. Kwapisz, G. M. Weiss, and S. A. Moore. Cell phone-based biometric identification. In *BTAS*, pages 1–7, 2010.

[96] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4android: a generic operating system framework for secure smartphones. In *ACM SPSM*, pages 39–50. ACM, 2011.

[97] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee. The core of the matter: Analyzing malicious traffic in cellular carriers. In *NDSS*, 2013.

[98] LibSvm. `https://www.csie.ntu.edu.tw/~cjlin/libsvm/`.

[99] Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store. `https://github.com/jamiebullock/LibXtract`.

[100] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *IEEE DSN*, 2008.

[101] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *ACM Ubicomp*, 2012.

[102] Z. Liu and S. Sarkar. Improved gait recognition by gait dynamics normalization. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (6):863–876, 2006.

[103] H. Lockheimer. Android and security, February 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[104] H. Lu, J. Huang, T. Saha, and L. Nachman. Unobtrusive gait verification for mobile phones. In *Proceedings of the 2014 ACM international symposium on wearable computers*, pages 91–98. ACM, 2014.

[105] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[106] Mobile Malware Dump. `http://contagiominidump.blogspot.com`.

[107] Ant Financial. `https://www.antfin.com/`.

[108] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, 2006.

[109] Mocana - Strong and Usable Security. `https://www.mocana.com/`.

[110] A. Nadkarni and W. Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1029–1042. ACM, 2013.

[111] S. Nath. Madscope: Characterizing mobile in-app targeted ads. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 59–73. ACM, 2015.

[112] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[113] M. S. Nixon, J. N. Carter, D. Cunado, P. S. Huang, and S. Stevenage. Automatic gait recognition. In *Biometrics*, pages 231–249. Springer, 1996.

[114] Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store. `http://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/`.

[115] J. Oberheide. Dissecting android's bouncer, June 2012. `https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/`.

[116] D. L. Olson and D. Delen. *Advanced data mining techniques*. Springer, 2008.

[117] M. Ongtang, K. Butler, and P. McDaniel. Porscha: Policy oriented secure content handling in android. In *ACSAC*, pages 221–230. ACM, 2010.

[118] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*, 2013.

[119] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *IEEE ICSE*, 2012.

[120] Android PathClassLoader. `http://developer.android.com/reference/dalvik/system/PathClassLoader.html`.

[121] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, 2014.

[122] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *USENIX NSDI*, 2013.

[123] ProGuard. `http://developer.android.com/tools/help/proguard.html`.

[124] ptrace. `http://man7.org/linux/man-pages/man2/ptrace.2.html`.

[125] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *ACM SIGCOMM*, 2010.

[126] QQ. `https://play.google.com/store/apps/details?id=com.tencent.mobileqq&hl=en`.

[127] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[128] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 209–220. ACM, 2013.

[129] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[130] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley. Are these ads safe: Detecting hidden attacks through the mobile app-web interfaces. 2016.

[131] Android BroadcastReceiver. `https://developer.android.com/reference/android/content/BroadcastReceiver.html`.

[132] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[133] Y. Ren, Y. Chen, M. C. Chuah, and J. Yang. User verification leveraging gait recognition for smartphone enabled mobile healthcare systems. *Mobile Computing, IEEE Transactions on*, 14(9):1961–1974, 2015.

[134] J. C. Reynar and A. Ratnaparkhi. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the fifth conference on Applied natural language processing*, 1997.

[135] F. Rohrer, N. Feleke, Y. Zhang, K. Nimley, L. Chitkushev, and T. Zlateva. Android security analysis and protection in finance and healthcare. *Boston University MET*.

[136] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva. Dr baca: Dynamic role based access control for android. In *ACSAC*, pages 299–308. ACM, 2013.

[137] White Paper : An Overview of Samsung KNOX. `http://www.samsung.com/es/business-images/resource/white-paper/2014/02/Samsung_KNOX_whitepaper-0.pdf`.

[138] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[139] S. Sarkar, P. J. Phillips, Z. Liu, I. R. Vega, P. Grother, and K. W. Bowyer. The humanid gait challenge problem: Data sets, performance, and analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(2):162–177, 2005.

[140] Significant iPhone and iPad malware threats will emerge in 2015. `http://www.ibtimes.co.uk/significant-iphone-ipad-malware-threats-will-emerge-2015-1490577`.

[141] Smali: An assembler/disassembler for Android's dex format. `http://code.google.com/p/smali/`.

[142] S. Smalley and R. Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[143] R. Socher, J. Bauer, C. D. Manning, and A. Y. Ng. Parsing with compositional vector grammars. In *Proceedings of the ACL*, 2013.

[144] S. Sprager and D. Zazula. A cumulant-based method for gait identification using accelerometer data with principal component analysis and support vector machine. *WSEAS Transactions on Signal Processing*, 5(11):369–378, 2009.

[145] Tencent. https://www.tencent.com/en-us/index.html.

[146] K. Tian, D. Yao, B. G. Ryder, and G. Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 262–271. IEEE, 2016.

[147] J. E. Trost. Statistically nonrepresentative stratified sampling: A sampling technique for qualitative studies. *Qualitative sociology*, 9(1):54–57, 1986.

[148] UI/Application exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[149] Android UsageStatsManager. https://developer.android.com/reference/android/app/usage/UsageStatsManager.html.

[150] J. Vaidya, V. Atluri, and J. Warner. Roleminer: mining roles using subset enumeration. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 144–153. ACM, 2006.

[151] E. Vazquez-Fernandez and D. Gonzalez-Jimenez. Face recognition for authentication on mobile devices. *Image and Vision Computing*, 2016.

[152] VirusTotal. https://www.virustotal.com/.

[153] WeChat. https://play.google.com/store/apps/details?id=com.tencent.mm&hl=en.

[154] J. Weston, S. Mukherjee, O. Chapelle, M. Pontil, T. Poggio, and V. Vapnik. Feature selection for svms. 2000.

[155] R. Whitwam. Circumventing Google's Bouncer, Android's anti-malware system, June 2012. http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system.

[156] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, pages 539–552, 2012.

[157] Y. Xu and E. Witchel. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems*, page 26. ACM, 2015.

[158] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium*, 2012.

[159] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *Research in Attacks, Intrusions, and Defenses*, pages 359–381. Springer, 2015.

[160] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054. ACM, 2013.

[161] H. Zhang, D. D. Yao, and N. Ramakrishnan. Causality-based sensemaking of network traffic for android application security. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, pages 47–58. ACM, 2016.

[162] Y. Zhang, X. Luo, and H. Yin. Dexhunter: toward extracting hidden code from packed android applications. In *Computer Security–ESORICS 2015*, pages 293–311. Springer, 2015.

[163] Z. Zhao and F. C. C. Osono. trustdroid: Preventing the use of smartphones for information leaking in corporate networks through the used of static analysis taint tracking. In *MALWARE*, pages 135–143. IEEE, 2012.

[164] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications. In *CODASPY*, pages 37–48. ACM, 2015.

[165] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *Security and Privacy, IEEE Symposium on*, 2012.

[166] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[167] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.