การเรียนรู้การส่งผ่านข้อมูลส่วนตัวจากแอปพลิเคชันบนระบบปฏิบัติการแอนดรอยด์จากข้อมูลการวิเคราะห์ซอร์สโค้ดอย่างรวดเร็ว

นายณัฐนนท์ วงศ์วิวัฒน์ชัย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2563

LEARNING PERSONALLY IDENTIFIABLE INFORMATION TRANSMISSION IN ANDROID APPLICATIONS BY USING DATA FROM FAST STATIC CODE ANALYSIS

Mr. Nattanon Wongwiwatchai

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2020

Thesis Title	LEARNING PERSONALLY IDENTIFIABLE INFORMATION TRANSMISSION IN ANDROID APPLICATIONS BY USING DATA FROM FAST STATIC CODE ANALYSIS
By	Mr. Nattanon Wongwiwatchai
Field of Study	Computer Engineering
Thesis Advisor	Dr. Kunwadee Sripanidkulchai, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Master's Degree

.... Dean of the Faculty of Engineering

(Professor Supot Teachavorasinskun, Ph.D.)

THESIS COMMITTEE

.... Chairman

(Associate Professor Kultida Rojviboonchai, Ph.D.)

.... Thesis Advisor

(Dr. Kunwadee Sripanidkulchai, Ph.D.)

.... Examiner

(Dr. Ekapol Chuangsuwanich, Ph.D.)

.... External Examiner

(Assistant Professor Sukumal Kitisin, Ph.D.)

ณัฐนนท์ วงศ์วิวัฒน์ชัย: การเรียนรู้การส่งผ่านข้อมูลส่วนตัวจากแอปพลิเคชันบนระบบปฏิบัติการแอนดรอยด์จากข้อมูลการวิเคราะห์ซอร์สโค้ดอย่างรวดเร็ว. (LEARNING PERSONALLY IDENTIFIABLE INFORMATION TRANSMISSION IN ANDROID APPLICATIONS BY USING DATA FROM FAST STATIC CODE ANALYSIS) อ.ที่ปรึกษาวิทยานิพนธ์หลัก : ดร. กุลวดี ศรีพานิชกุลชัย, 58 หน้า.

ความสะดวกสบายในการใช้เครื่องมือสื่อสารส่งผลให้มีการใช้งานเครื่องมือสื่อสารในชีวิตประจำวันและเก็บข้อมูลส่วนตัวลงในเครื่องมือเหล่านั้นมากขึ้น ซึ่งปัจจุบันผู้ใช้งานเครื่องมือสื่อสารก็เริ่มตระหนักถึงการเข้าถึงข้อมูลส่วนตัวโดยแอปพลิเคชัน รวมถึงความเสี่ยงที่แอปพลิเคชันเหล่านี้จะส่งข้อมูลส่วนตัวของผู้ใช้งานไปยังเซิร์ฟเวอร์ภายนอก โดยที่บางครั้งผู้ใช้งานก็ไม่รู้ตัว การที่จะบอกว่าแอปพลิเคชันมีการส่งออกข้อมูลส่วนตัวออกไปหรือไม่นั้นไม่ใช่เรื่องง่าย ถ้าเกิดมีข้อมูลเหล่านี้เปิดเผยให้ผู้ใช้งานรับรู้ก่อนทำการติดตั้งแอปพลิเคชัน ผู้ใช้งานก็ทราบถึงข้อดีและข้อเสียของแอปพลิเคชัน และสามารถตัดสินใจได้ว่าจะยอมรับความเสี่ยงที่ข้อมูลส่วนตัวจะถูกเปิดเผยได้หรือไม่ ในอดีตวิธีการตรวจจับการส่งออกข้อมูลส่วนตัวอาทิเช่น การวิเคราะห์ซอร์สโค้ดและการวิเคราะห์พฤติกรรมเชิงพลวัตนั้นเป็นวิธีที่ใช้เวลาในการวิเคราะห์นาน ใช้เวลาตั้งแต่หลายนาทีถึงหลายชั่วโมง ในขณะที่วิทยานิพนธ์นี้ได้นำวิธีการวิเคราะห์ซอร์สโค้ดอย่างรวดเร็วไปใช้ในการสร้างโมเดลจำแนกแอปพลิเคชันที่มีการส่งออกข้อมูลส่วนตัวภายในระยะเวลาต่ำกว่าหนึ่งนาที โดยมีประสิทธิภาพที่เทียบได้กับวิธีการในอดีต คณะผู้จัดทำได้ทำการทดสอบประสิทธิภาพของโมเดลกับแอปพลิเคชันยอดนิยมบนระบบปฏิบัติการแอนดรอยด์มากกว่า 19,000 แอปพลิเคชัน วิธีการตรวจจับใหม่นี้มีประสิทธิภาพและรวดเร็ว ทำให้เหมาะสมกับการนำไปใช้วิเคราะห์และตรวจจับการส่งออกข้อมูลส่วนตัวของแอปพลิเคชันได้ทันทีทันใด

ภาควิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อนิสิต	....
สาขาวิชา	วิศวกรรมคอมพิวเตอร์	ลายมือชื่อ อ.ที่ปรึกษาหลัก	....
ปีการศึกษา	2563		

6270081721: MAJOR COMPUTER ENGINEERING

KEYWORDS: DATA ANALYTICS, MACHINE LEARNING, PRIVACY, PERSONALLY IDENTIFIABLE INFORMATION (PII), MOBILE APPLICATIONS

NATTANON WONGWIWATCHAI : LEARNING PERSONALLY IDENTIFIABLE INFORMATION TRANSMISSION IN ANDROID APPLICATIONS BY USING DATA FROM FAST STATIC CODE ANALYSIS. ADVISOR : Dr. Kunwadee Sripanidkulchai, Ph.D., 58 pp.

The ease of use of mobile devices has resulted in a significant increase in the everyday use of mobile applications as well as the amount of personal information stored on devices. Users are becoming more aware of applications' access to their personal information, as well as the risk that these applications may unwittingly transmit Personally Identifiable Information (PII) to third-party servers. There is no simple way to determine whether or not an application transmits PII. If this information could be made available to users before installing new applications, they could weigh the pros and cons of having the risk of their PII exposed. To detect PII transmission, heavy-weight methods like static code analysis and dynamic behavior analysis are used. They take anywhere from a few minutes to several hours of testing and analysis per application. On the other hand, in this thesis, we use fast static code analysis to extract features that we then use to build a classification model to detect PII transmission in under a minute with performance comparable to heavy-weight methods. We evaluate our model against a large number of top-ranked Android applications, totaling over 19,000. Our method is both fast and effective, making it ideal for detecting and analyzing PII transmission in mobile applications in real-time.

Department:	Computer Engineering	Student's Signature	....
Field of Study:	Computer Engineering	Advisor's Signature	....
Academic Year:	2020		

Acknowledgements

I want to thank my advisor, Dr. Kunwadee Sripanidkulchai, for everything from the first time I met her during junior year to the end of this master's degree journey. She gave both academic-related and personal-related advice. She guided me to think more systematically and constructively.

I especially thank Phannawhat Pongkham, Papon Chaisrisookumporn, Nisaruj Rattanaaram, Prawsang Chayakulkeeree, and Methus Narksenee for supporting various processes in this thesis.

I am very honored to have Asst. Prof. Kultida Rojviboonchai, Dr. Ekapol Chuangsuwanich, and Asst. Prof. Sukumal Kitisin as my thesis committee. I also would like to express my gratitude for taking the time to read and critique my manuscript.

I also want to thank my family, friends, and colleagues for both physical and mental support.

I would like to thank CP Alumni from the department of computer engineering, Faculty of Engineering, Chulalongkorn University for the Chula Computer Engineering Graduate Scholarship.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter I

INTRODUCTION

Smartphones and tablets have become an essential part of our daily lives. The usage of mobile devices has grown beyond the global population [1]. With more than 75% market share, Android is the most popular mobile operating system [2]. The Google Play Store has approximately 3 million applications available for download [3]. Mobile devices store Personally Identifiable Information (PII) such as contact information, email addresses, geolocation information, and unique device identifiers for a variety of reasons. In concept, PII is sensitive personal information that should not be shared with anyone without the user's permission. Legislation such as the EU General Data Protection Regulation [4] is designed to protect users' privacy. Applications must fully disclose their data collection activities and obtain user consent before accessing user data. In practice, however, users rarely read the privacy policies of applications or pay attention to permission requests. They often agree to all requests in order to use applications without fully understanding the risks involved [5]. As a result, users may have unknowingly agreed to allow their PII to be transmitted to remote servers such as application servers, ad servers, trackers, and malicious sites directly or through the use of third-party libraries by applications and sometimes applications intentionally fail to disclose that they collect PII [6]. This is recognized as the PII transmission problem [7].

Before downloading and installing an application, users should be able to quickly check and verify if it transmits PII. Because downloading is interactive, these checks should only take a few seconds. When users receive the verification result, they can choose to install the application or decline to install it in order to protect their personal information from being transmitted by the application. As a result, fast, reliable, and accurate real-time detection is required.

To detect PII transmissions made by applications, a variety of techniques have been proposed. They can be divided into three categories. The first approach is based on static analysis of the application source code by decompiling and analyzing the information in the source code without running it. The analysis is heavy-weight in that it creates a complete control flow graph by tracing all method calls in the source code to find potential PII leakage paths. [8; 9; 10; 11]. The second approach is dynamic analysis of application behavior by analyzing the network traffic generated while running the application [11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21; 22; 23; 24]. The last approach is a combination of the first two approaches [11]. While all of these approaches are effective at detecting PII tranmissions, they are all heavy-weight in that they take a long time to run, from many minutes to hours. These approaches are ineffective for real-time detection.

In contrast, we use fast static code analysis to detect PII transmissions. It is a light-weight approach that takes less than a minute to run. We take a similar approach to static analysis in that we analyze the source code without running the application. However, we take a step back and do not generate the control flow call graph. We are the first to argue that detecting PII transmissions using simple pattern matching to analyze strings in the source code is already highly effective. In this thesis, we present a real-time approach for detecting PII transmissions in mobile applications using fast static code analysis.

The main contributions of this thesis are:

- An approach for detecting PII transmissions in Android applications, using learning algorithms on features generated by fast static code analysis for over 19,000 applications. Our work is suitable for application users who want a real-time analysis on an application before deciding to install it, all in under a minute.
- An evaluation that compares the performance of classification models with different features and learning algorithms. Our model is capable of achieving an F1 score of greater than 0.70.

- As part of our experimental evaluation, we found that 51.8% of the top mobile applications in today's Google Play Store are identified as applications that transmit PII. This is a much larger-scale analysis with a much higher number of PII-vulnerable applications than previously reported.

Chapter II

BACKGROUND

In this section, we cover background knowledge that relates to our work. The first section is about defining Personally Identifiable Information (PII). The next section is about learning algorithms that we select from related work to apply to our work. The last section is about metrics that we use to evaluate the performance of the classification model.

2.1 Personally Identifiable Information (PII) Definition

Even though there is no universally accepted definition of Personally Identifiable Information (PII), many countries have enacted legislation that defines PII as “any information or opinion that directly or indirectly relates to an identifiable live natural person (excluding data about the deceased or legal persons) [11]”.

There is no standardized list of data elements defined as PII, despite the fact that there is a vast body of related work in mobile application privacy [8; 9; 10; 16; 17; 18; 19; 20; 21; 22; 23] as shown in Table A.1 in the Appendix. Because each study defines and detects different sets of data elements, we utilize a comprehensive list compiled from previous work, as shown in Table 2.1 [11]. There are 32 PII data items in total that can be divided into 4 types based on the source of PII: device, SIM card, user, and location information.

Table 2.1: Comprehensive PII List.

PII Type	**PII List**
Device Information	Advertiser ID, Android ID, Device Serial Number, Google Services Framework ID, IMEI, MAC Address
SIM Card Information	GSM Cell ID, ICCID, IMSI, Location Area Code, Phone Number
User Information	Age, Audio, Calendar Event, Contract Book, Country, Credit Card Number and CCV, Date Of Birth, Email, Gender, Name, Password, Photo, Physical Address, Relationship Status, SMS Message, SSN, Time Zone, Username, Video, Web Browsing Log
Location Information	GPS (Latitude and Longitude)

2.2 Learning Algorithms

We select the following learning algorithms used by previous work related to fast static code analysis to create the classification model for detecting PII transmission.

1. Neural Network (NN): A neural network is a computational system based on the human brain's neural structure. A Neural Network has multi-layer networks of neurons. A neuron is a mathematical function that takes inputs and applies a function on them to get the output. To transfer its knowledge, each neuron in the same layer connects to neurons in other layers.
2. Logistic Regression (LR): Logistic Regression is an algorithm that finds a logistic curve that best fits the dataset. It requires the creation of a logit variable containing the natural log of the probability of the class occurring or not occurring. The probabilities are then estimated using the maximum likelihood estimation algorithm.
3. Support Vector Machine (SVM): Support Vector Machine is an algorithm for determining the best hyperplane for separating two classes of data. The best hyperplane has the minimum error, maximizing the margin of separation between two classes. For data with high dimensions, one needs to use a mapping function to make the data linearly separable. The mapping function can be represented in SVM training and prediction by the dot product of the mapping or the kernel function, which can be efficiently computed and is usually easier than the mapping function. The Gaussian function and RBF (Radial-Basis function) are the most commonly used kernel functions.
4. Naive Bayes (NB): Naive Bayes is a probabilistic classifier based on Bayes theorem for conditional probabilities. It is based on the assumption that all attributes in a dataset are independent of one another. As a result, it assumes that the presence or absence of a characteristic representing a specific class has no bearing on the presence or absence of any other characteristic, which

is not true for the majority of classification tasks. The maximum likelihood algorithm is commonly used in Naive Bayes training.

5. k-Nearest Neighbor (kNN): k-Nearest Neighbor algorithm saves all training data and classifies new data points based on the majority class of their k-nearest neighbors in the given dataset. The Euclidean distance is used by kNN to calculate the distance between pairs of data in order to find the nearest neighbors for each data.

6. Random Forest (RF): Random Forest is an ensemble algorithm that chooses the majority of decision tree results. Each tree is built using a unique set of training dataset.

2.3 Performance Metrics

We use the following metrics to evaluate the performance of each classification model.

True Positives, False Positives, True Negatives, False Negatives

True Positives (TP) is the number of applications that transmit PII that are correctly classified, while False Positives (FP) is the number of applications do not transmit PII that are incorrectly classified.

True Negatives (TN) is the number of applications that do not transmit PII that are correctly classified, while False Negatives (FN) is the number of applications that transmit PII that are incorrectly classified.

Accuracy, Precision, Recall, F1 Score

Accuracy is the ratio between the number of correct predictions and the number of total predictions.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2.1}$$

Precision is the ratio between the number of correct positive predictions and the number of positive predictions.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

Recall is the ratio between the number of correct positive predictions and the number of positive labels.

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

F1 score is the weighted average of precision and recall defined below. It is different from accuracy in that it does not take true negatives into consideration.

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{2.4}$$

Chapter III

RELATED RESEARCH

In this section, we discuss related research based on three approaches to detecting PII transmission: dynamic analysis, static analysis, and a combination of both dynamic and static analysis.

3.1 Dynamic Analysis for PII Transmission Detection

The goal of dynamic analysis is to keep track of network traffic flows while the application is running. Interaction and input for applications can be created manually or automatically with the help of a tool like Monkey [25], Appium [26], and Sapienz [27]. Setting up a proxy server as an intermediate server between the mobile device and the Internet is required to capture network traffic while applications are running. Various techniques, such as string matching [12; 13; 14; 15; 16; 19; 20; 21; 22; 23], rules for identifier detection [17], and time-series flow construction [18] are then used to extract the captured traffic.

Dynamic analysis is a popular approach because if a leak is detected, it is definitive. However, there is one major flaw: there is no comprehensive and scalable method for generating input for the application. Although manual testing has the potential to provide comprehensive coverage, it is not scalable when dealing with a large number of applications. On the other hand, automated input generation tools that are currently available are scalable but not comprehensive, as they can only achieve around 50% code coverage [28]. Furthermore, dynamic testing takes much longer than static testing because it requires many time-consuming steps, such as starting a mobile device, installing the app, and running the app long enough to capture traffic for analysis. An application is typically tested in 10 minutes. However, it is sometimes necessary to spend more than 90 minutes [19].

3.2 Static Analysis for PII Transmission Detection

The focus of static analysis is to examine application source code rather than run the application. Using decompiler tools like Apktool [29], jadx [30],or baksmali [31], the application package must be decompiled to acquire the original source code. After that, information is extracted and analyzed from the source code. There are two types of static analysis: heavy-weight and light-weight.

Source code is analyzed in the heavy-weight approach by reconstructing control flow graphs to find source-sink paths using tools like LeakMiner [8], AndroidLeaks [9], and FlowDroid [10]. A source is a method for reading or writing non-constant data to a shared memory resource, while a sink is a method for sending non-constant data from shared memory to a remote connection. If PII-related methods are defined as sources, this approach can be used to find PII source-sink paths in the code. Reconstructing call graphs ensures that all of the application's possible paths are explored. As a result, heavy-weight static analysis provides more comprehensive code coverage than dynamic analysis.

However, when leaks are found on paths that are not reachable during execution, such as dead code branches [10], heavy-weight static analysis may produce false positives. False negatives may also occur if the source-sink list is missing, the control flow graph is incorrectly constructed, or the application source code is obfuscated to avoid detection.

FlowDroid is the most advanced heavy-weight static approach, with high precision (0.86) and very high recall (0.93) when compared to LeakMiner's precision of 0.475 and AndroidLeaks' precision of 0.65.

In the light-weight approach, application manifest files (application metadata) and source code are analyzed using fast static code analysis techniques such as pattern matching to extract strings and tokens. Light-weight static analysis using tools such as MobSF [32] have been used as features for malware and virus detection machine learning models [33; 34; 35; 36; 37; 38], but there is no previous work exploring the relationship between data extracted from light-weight approaches and privacy. This thesis is the first study to use light-weight static analysis for detecting PII transmissions.

When comparing state-of-the-art heavy-weight vs. light-weight static analysis, specifically FlowDroid vs. MobSF, FlowDroid's call graphs can be difficult to fully reconstruct, depending on the size and complexity of the application. It is possible that some paths will be completely overlooked. Furthermore, FlowDroid takes longer to analyze, taking an average of 2-3 minutes vs. 45 seconds for MobSF.

While light-weight approaches are faster, it is questionable whether they are effective or not. In this thesis, we argue and demonstrate that light-weight static analysis can be used to detect PII transmissions as an alternative to heavy-weight static and dynamic approaches.

3.3 Combined Dynamic and Static Analysis for PII Transmission Detection

Because both dynamic analysis and static analysis have flaws, relying solely on one of them for detection may not be sufficient. As a result, we combine their results to compensate for their individual weaknesses, leading to better detection in by VULPIX (**VUL**nerable **P**ersonal **I**nformation Lea**ks**) [11]. VULPIX uses both static (VULPIX$_S$) and dynamic analysis (VULPIX$_D$) to detect PII transmissions in Android applications. When a PII transmission is detected by any approach, VULPIX will flag the application as having a PII transmission.

VULPIX$_S$ improving FlowDroid's call graph construction and source-sink matching by updating the PII-related sources, i.e., methods listed in the official Android API references [39], into the source-sink file. As a result, this improves the detection of PII.

Instead of using Monkey [25] to generate user interaction during dynamic testing, VULPIX$_D$ uses *Man*key, a new tool that addresses Monkey's shortcomings: unable to perform human-like tasks, such as entering values into text fields. This is a common interaction in modern applications (i.e., logging in with a username and password), and it restricts the application coverage achievable by Monkey. Mankey, which is specifically designed to handle PII input, was built using Appium's testing framework [26]. Mankey outperforms Monkey in the following ways:

- Rather than randomly selecting any position on the screen, Mankey selectively performs actions on interactive elements, such as clicking on buttons, clicking on clickable elements, filling in text fields, and pressing the back button, as if it were a real human user.
- Mankey's text input is designed to look like a human user filling in the correct information for data elements like credit card numbers, usernames, and passwords, rather than randomly filling in text fields. Mankey achieves this by detecting all visible elements on the screen and determining whether or not any of them are text fields. It then loops through each of these fields, looking for a clue as to what data that field is attempting to collect. For example, if the field has a resource-id of the string "pass", "pwd", or "pword", Mankey will enter the password in this field.
- If there are no interactive elements on the current screen, Mankey will revert to Monkey's default behavior. However, Mankey enacts a few limitations in order to facilitate data collection. Mankey will not perform any system-level actions other than the back button because these other actions may interfere with the device's wifi (i.e., turn it off) and disable the network traffic capturing

process.

- To enable apps that need login using third-party OpenID providers like Google or Facebook, Mankey permits a maximum of 5 actions to be performed outside the target application.

VULPIX$_D$ performs automated testing on the application using Mankey while capturing network traffic through a proxy, mitmproxy [40].

VULPIX$_D$ uses three PII detection algorithms on the network traffic: exact-matches, regular expression matches, and field extraction from HTTP request message. When the PII value is known and unique, such as the device IMEI, exact-matches are used [11]. When the PII is not unique but has a well-defined format, such as a credit card number, regular expression matches are used [11]. When field values are obfuscated or encrypted, field extraction from HTTP request message are used. HTTP request messages consist of 3 parts; request lines (method, URL, and version), header lines (pairs of header field name and value), and body (post data for POST method only). Values of URL, header field name, and post data are extracted by focusing on field names (e.g. username, password, lat, lon) to bypass the obfuscated or encrypted field values. There are two methods to extract these field names; exact-match with unique string for the specific field name and regular expression matching for the common field name.

VULPIX can detect all 32 PII elements listed in Table 2.1 by combining both static and dynamic approaches, whereas VULPIX$_S$ can only detect 24. VULPIX$_D$ can theoretically detect all 32 PII elements, but in our initial evaluation of 19,608 applications, 20 PII elements were detected as shown in the “This Thesis Dataset” column of Table A.1 in the Appendix due to (i) Mankey’s limited application testing capabilities, which may miss certain application interactions, or (ii) the tested applications did not send those PII.

VULPIX’s detection may be comprehensive, but it is heavy-weight because it takes around 15 minutes to test, making it unsuitable for real-time detection. The detection results from VULPIX are used as labels in this thesis to train features extracted from MobSF's light-weight static analysis. Because light-weight static analysis is fast, we argue that it can be used in place of heavy-weight approaches to reduce the detection time to under a minute.

Chapter IV

METHODOLOGY

In this section, we describe the methodology we use to identify mobile applications that transmit PII. We envision our work being used by mobile application users to quickly check and verify whether an application transmits PII before downloading and installing it. We would like to be able to do this in real-time to support users who are browsing the application market for new applications to install.

Our methodology is to use light-weight static features from MobSF's fast static code analysis results to create a classification model for detecting mobile applications that transmit PII from those that do not, using a number of learning algorithms as depicted in Figure 4.1. Our methodology consists of three parts. The first part is "Features Extraction". We use MobSF to extract light-weight static features from applications and VULPIX to generate labels for PII transmissions applications. The next part is "Features Preparation". We select features and convert them into a format that can be used to train the classification model. The last part is "Model Training". We train each classification model by using different learning algorithms. Finally, we have classification models that can detect applications that transmit PII.

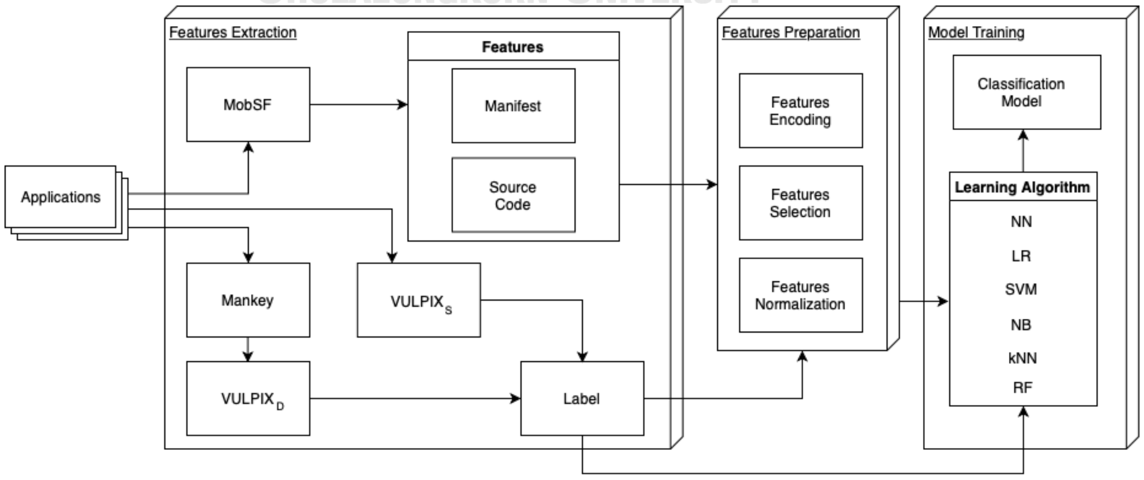

Figure 4.1: Overview of methodology to detect mobile applications that transmit PII.

We can guarantee fast detection in under a minute because extracting features from a new application that we have not previously analyzed using light-weight static analysis takes less than a minute, and running those features through our classification model takes less than a second. Next, we describe each of these parts in detail.

4.1 Feature Extraction

We extract light-weight static features from a mobile application package (.apk) file using MobSF. These features can be divided into two different sets of information based on the source of the feature: S_1 are features obtained from the Android manifest file and S_2 are features obtained from the source code as shown in Table 4.1. The extraction times for S_1 and S_2 are different, with the manifest file taking 5-10 seconds and the source code taking 15-30 seconds. Given that S_1 is much faster to obtain than S_2, we are interested in understanding if using S_1 alone can result in good classification performance. If so, the time it takes to classify applications can be even shorter.

The features in S_1 contain basic application information such as package name and version, manifest analysis, and permissions that the application requests from the device such as access to the camera, location, etc. Note that the largest number of features permission requests. There are 1005 different permission requests across all the applications we looked at. The features in S_2 contain results from MobSF's binary analysis (if the application is vulnerable to buffer overflow attacks), APKiD analysis (if the application uses vulnerable obfuscation techniques), code analysis (if the application performs methods that are considered insecure), signer certificate analysis, file analysis (if the application hardcodes the location of certificates or keystores), and lastly Android API analysis for all the Android API calls used by the application. More details and examples of features are described in Table 4.1.

Table 4.1: Feature sets for training based on source of information.

Feature set	Features	Description	#Features
S_1 Manifest	application info	basic information about the application: package name and real application name	2
	manifest analysis	activities and properties in the manifest that MobSF identifies as not secure such as an activity that is found to be shared with other apps on the device.	45
	permissions	permissions that the application requests from the device, including standard Android permissions (camera, location, internet, etc.) and custom permissions for application-specific objectives typically used to share resources and capabilities with other apps.	1005
S_2 Source code	binary analysis	vulnerability of the application binary that MobSF identifies as not secure, specifically, an executable (elf) built without the Position Independent Executable flag that helps prevent against buffer overflow attacks	1
	APKiD analysis	source code obfuscation techniques used by the application that MobSF identifies as not being secure such as Arxan obfuscator, Baidu packer, etc.	66
	code analysis	use of methods in the application source code that MobSF identifies as not secure such as reading or writing to external storage, copying data to clipboard, creating temp file, etc.	33
	signer certificate	type and status of certificate that is used to sign the application	5
	file analysis	a certificate or keystore is hardcoded in the application source code	1
	Android API	types of Android API used in application source code such as get System Service, send Broadcast, etc.	43

4.2 Feature Preparation

The light-weight static features we obtain directly from MobSF listed in Table 4.1 are not immediately ready to be used as input for training the classification model. To make these features more appropriate, we need to perform the following feature preprocessing tasks:

- Discard features that are unique for each application such as application names as they are not useful for learning.
- Convert features from strings to categorical values using one-hot encoding.
- Discard features that have the same value for every application.
- Select only important features that are useful for learning to avoid the curse of dimensionality that is detrimental to classification performance. The absolute value of the correlation between the feature and the label is used to select important features. The stronger the relationship between that feature and label, the higher the absolute value of correlation. We use a dropout threshold to determine which features to keep as discussed in the evaluation results in Section 5.
- Normalize all features to keep values within the same range.

4.3 Model Training

To train our model, we use six learning algorithms similar to those used in malware detection: Neural Network (NN), Logistic Regression (LR), Support Vector Machine (SVM), Naive Bayes (NB), k-Nearest Neighbor (kNN), and Random Forest (RF). The scikit-learn and Keras Python libraries are used to create classification models. For the Neural Network model, we also use dropout [41] and batch normalization layers [42] to improve model performance. The dropout layer is used to prevent a model from overfitting. Meanwhile, the batch normalization layer is used to reduce the gradient vanishing problem. For each algorithm, we perform 5-fold cross validation and evaluate performance based on the average of each model's performance metrics. We also use hyperparameters listed in Table 4.2 selected based on our initial evaluation of hyperparameters in each algorithm as shown in Table B.1 in the Appendix. The hyperparameter that gives a model its best F1 score is selected to use in experiments except for the Random Forest model. For Random Forest, we use 128 trees because there is no significant difference between the F1 scores of models using 128, 256, 512, and 1,024 trees. Since increasing the number of trees also increases the training time, using 128 trees in the model is a good performance trade off with a reasonable training time [43].

Table 4.2: Hyperparameters configured for each learning algorithms.

Learning Algorithm	**Configuration**
Neural Network	4 fully connected layers
Logistic regression	lbfgs solver
Support Vector Machine	rbf kernel
Naive Bayes	Multinomial Naive Bayes
k-Nearest Neighbor	k=7
Random Forest	number of trees = 128

Chapter V

RESULTS

In this section, we discuss the application dataset and label that was used in experiments. Then, we report on our experimental results to see how well features extracted by fast static code analysis can be used to classify mobile applications that transmit PII. We have three different test environments: (i) using features only from the manifest file S_1, (ii) using features only from the source code S_2, and (iii) using features from both manifest files S_1 and source code S_2. If the results from any of the above settings are comparable to heavy-weight dynamic and static detection, then we can conclude that light-weight static analysis is effective at classifying applications that transmit PII.

5.1 Application Dataset and Label

We are unable to use any publicly available application package datasets in our research. Applications collected and analyzed in previous work are (i) 3 to 8 years old and may accurately not reflect current application behavior, and (ii) too few to benefit from machine learning, mostly under 1,000 applications. As a result, we collected a new dataset of recent applications. We scrape applications from the Google Play Store by selecting top free applications from all categories in February 2019, obtaining a total of 22,905 distinct applications from 51 categories.

However, when we run VULPIX on all applications in our dataset to detect PII to use as our labels, 3,297 applications are unable to complete the VULPIX$_D$ process because of two problems. First, Appium, used by our dynamic testing tool Mankey, automatically focused on the wrong application activity when the application has multiple activities. This results in the application being detected as not running and failing to generate network traffic. This is a known issue and can be handled by

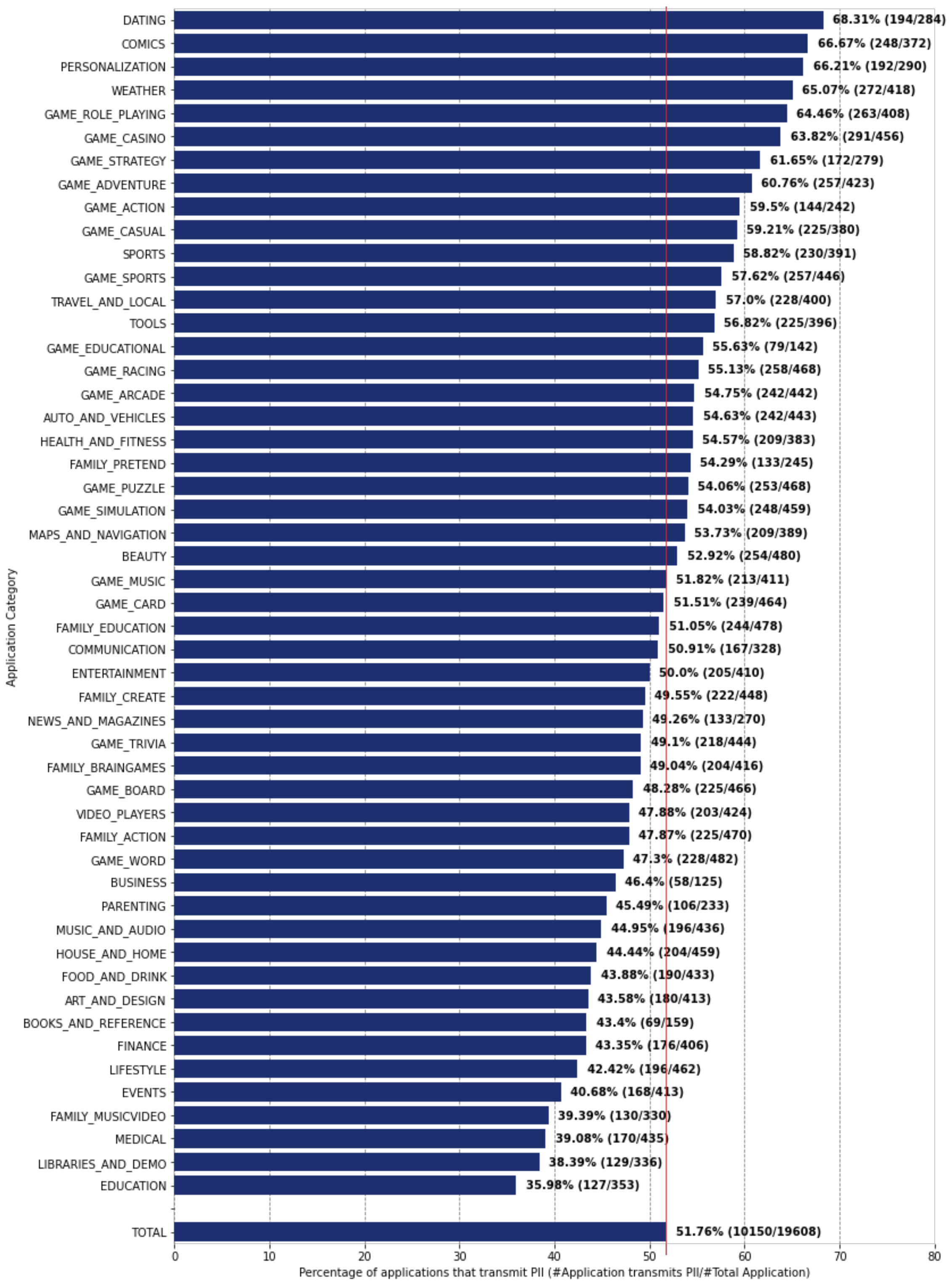

Figure 5.1: Overview of the percentage of applications that transmit PII in each application category.

iteratively specifying the activity name to Appium based on the activity that ran in the initial attempt. Secondly, some applications can detect that there is a man-in-the-middle proxy, and application servers refuse user connections.

After that, we obtain the initial detection result from VULPIX. However, this result may incorrectly classify some applications. So, we recheck the result with the actual traffic to confirm the correctness of the result. Then, we update detection algorithms to make sure they correctly classify the application. We repeat this process until making sure that our labels are accurate.

Of the 19,608 applications that successfully ran, VULPIX identified 10,150 (51.8%) applications as having PII transmissions and 9,458 (48.2%) applications as not having any PII transmissions. Figure 5.1 shows the percentage of applications that transmit PII for each application category. The x-axis represents the percentage of applications that transmit PII and the y-axis represents the application category. As an example, consider the dating category, 68.31% (194 of 284 applications) of the applications in this category transmit PII.

Overall, around half of all applications (51.8%) transmit PII. Considering that 75% of people have at least 11 applications downloaded and installed on their mobile devices [44], then there is a 51.8% chance a user downloads and installs an application that transmits PII. If users download and install 11 applications, this means that there is a 99.96% chance that they will have at least one application that transmits PII. We make the following observations about Figure 5.1.

- The five categories with the highest percentage of applications that transmit PII heavily rely on user PII for personalization. For example, dating applications need user information to personalize for the best matching and weather applications need user location to report the precise information
- The five categories with the lowest percentage of applications that transmit PII are less likely to require user PII for personalization.

- Game categories have a larger percentage of applications that transmit PII than other categories even though most games should not require access to PII.

Our dataset is the largest of its kind across all privacy-related studies as listed in Table 5.1. We detect that more that half of the tested applications transmit PII, which is significantly more than previously detected by ReCon (29.1% more), Agrigento (22.5% more), and COPPA (33% more).

We also consider how VULPIX is more comprehensive in detecting PII transmission when compared to the static or dynamic approach alone. Figure 5.2 shows the percentage of applications that transmit PII detected by VULPIX$_S$ and VULPIX$_D$ in each application category. The x-axis represents the percentage of applications that detected as transmitting PII by each approach of VULPIX and the y-axis represents the application category. As an example, consider the dating category, 23.71% of applications are detected as transmitting PII by the VULPIX$_S$ approach but not by VULPIX$_D$, 36.08% are detected as transmitting PII by VULPIX$_D$ approach but not but by VULPIX$_S$. And lastly, 40.21% are detected as transmitting PII by both VULPIX$_S$ and VULPIX$_D$ approach. This mean that if we use only VULPIX$_S$, 36.08% of applications in the dating category that actually transmit PII are not detected. On the other hand, 23.71% of applications in the dating category that actually transmit PII are not detected if we were to use only VULPIX$_D$.

Table 5.1: Comparison between dataset age, size, and percentage of applications that leak PII with previous work.

Research	Approach for obtaining features	Dataset date	#apps	% of apps w/ leaks
ReCon [16]	Dynamic Analysis	Aug 2015	950	22.7% (216/950)
Agrigento [19]	Dynamic Analysis	Aug 2015	950	29.3% (278/950)
COPPA [21]	Dynamic Analysis	Nov 2016 - March 2018	5,855	18.8% (1,100/5,855)
This thesis	VULPIX (Static & Dynamic Analysis)	Feb 2019	19,608	51.8% (10,150/19,608)

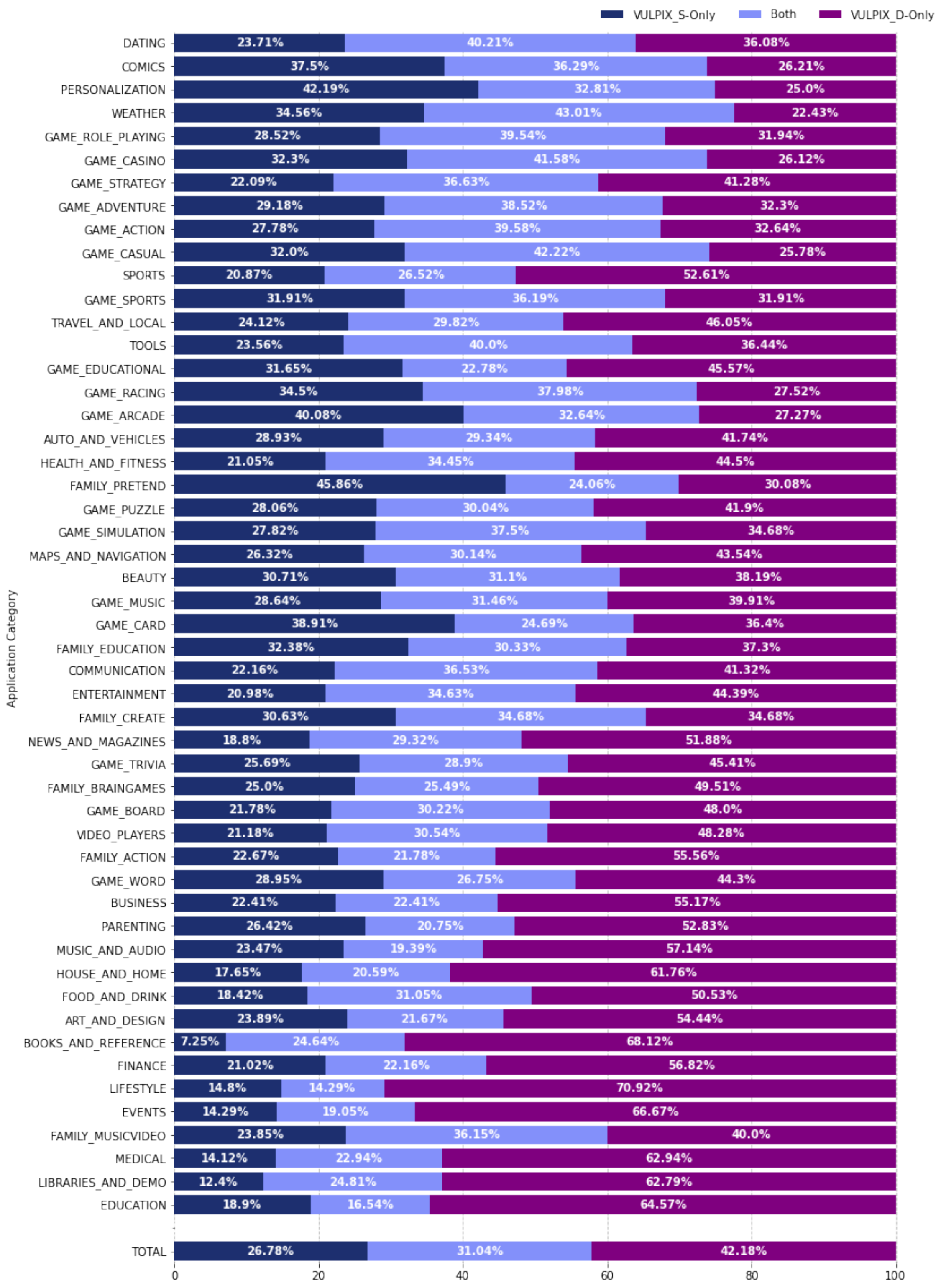

Figure 5.2: Proportion of applications that have PII transmission detected by VULPIX$_S$ and VULPIX$_D$ in each application category.

Overall, if we use only VULPIX$_S$, 42.05% of applications that actually transmit PII are not detected and labeled as no PII transmission. On the other hand, 26.83% of applications that actually transmit PII are not detected and labeled as no PII transmission if we use only VULPIX$_D$. Therefore, using VULPIX with both VULPIX$_S$ and VULPIX$_D$ provides the most comprehensive labels for our experiments.

5.2 Classification using the feature set from the manifest file S_1

In this experiment, we examine if classification using only the features from the manifest file is effective. Feature sets S_1 and S_2 as defined in Section 4 have a large number of features. We need to determine if we ought to use all of them for training because we need to avoid the curse of dimensionality that makes our model overfit. Figure 5.3 shows the cumulative distribution function (CDF) chart of the absolute value of correlation between each feature and the label. The Cumulative Distribution Function (CDF) is a function that calculates the probability that a random variable is less than or equal to the independent variable of the function. The x-axis represents the absolute value of correlation of each feature and the label and the y-axis represents the CDF. As an example, consider the black line, which has a CDF of 0.55 when the absolute value of correlation is 0.1. This means that 55% (0.55) of features in feature set S_2 have an absolute value of correlation less than or equal to 0.1.

From the CDF, we observe that a large percentage of features in S_1 and $S_1 + S_2$ are not correlated with the label. The correlation distribution is different between S_1 and S_2, suggesting that different correlation thresholds should be used to filter out features.

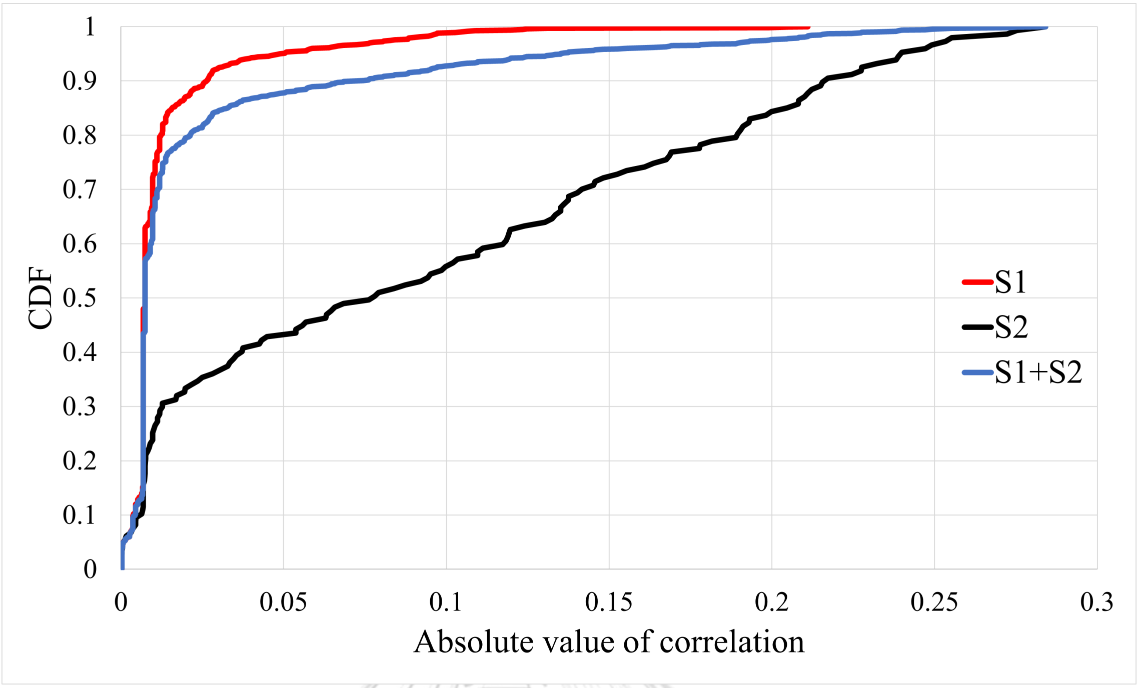

Figure 5.3: Cumulative distribution function of the absolute value of correlation between each feature and the label.

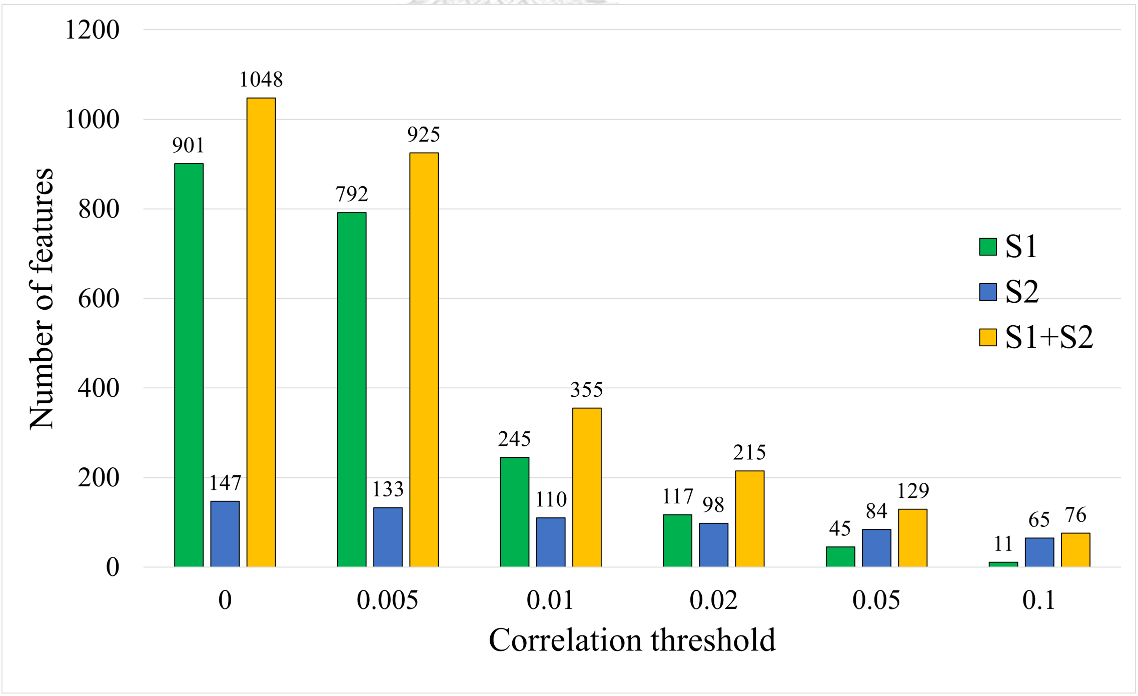

Figure 5.4: The number of features that have an absolute correlation value higher than the correlation threshold for each feature set.

To reduce the number of features, we select 5 correlation value as dropout thresholds based on Figure 5.3: 0.005, 0.01, 0.02, 0.05, and 0.1. Figure 5.4 shows the relation between the correlation threshold and the number of features for each feature set. The x-axis represents the correlation threshold and the y-axis represents the number of features in each feature set. For example, the green bar at 0.005 correlation threshold means that feature set S_1 has 792 features left when we set the correlation threshold to 0.005.

Table 5.2 shows the results of all six algorithms that are trained using only the features from S_1. The average value and standard deviation of the five models created during training using 5-fold cross-validation are shown in the table as performance metrics for each model. The higher the average value for each metric, the better the model's performance. The lower the standard deviation for each metric, the more stable the model's prediction performance. As an example, consider the first row for the NN (Neural Network) model that uses a correlation threshold of 0.005. This model has 0.6642 average accuracy and 0.0037 standard deviation. We explore various dropout thresholds ranging from low thresholds of 0.005 where we have 792 features to a more selective threshold of 0.1 where we are left with just 11 features as shown in Figure 5.4. Note that without using any dropout threshold, we would have 901 features in S_1.

Overall, the classification performance has high precision but low recall when only the features from the manifest file (S_1) are used. With a dropout threshold of 0.005, the Random Forest model has the highest average F1 score of 0.6623. We can train effective Random Forest models even when there are a large number of features to learn.

Increasing dropout thresholds, i.e., dropping more features, too aggressively results in worse performance in terms of lower F1 scores across all algorithms because there are fewer features to work with when training the model except for Naive Bayes. When the dropout threshold is increased from 0.005 to 0.1, average F1 scores improve from 0.5816 to 0.6073 for Naive Bayes. However, even the best Naive Bayes model performed poorer than other algorithms like Neural Network and Random Forest.

Table 5.2: Classification performance using the feature set from the manifest file S_1.

Algorithm	**Correlation Threshold**	**Accuracy**	**Precision**	**Recall**	**F1**
NN	0.005	0.6642±0.0037	0.7047±0.0148	0.6057±0.0256	0.651±0.0105
	0.01	0.6632±0.0032	0.7114±0.0165	0.589±0.0212	0.644±0.0088
	0.02	0.6671±0.0067	0.7081±0.0127	0.6074±0.0185	**0.6538±0.0115**
	0.05	0.6584±0.0056	0.7099±0.0293	0.5797±0.04	0.6368±0.0141
	0.1	0.6354±0.0058	0.6643±0.0221	0.6013±0.0447	0.63±0.0184
LR	0.005	0.6371±0.0021	0.6651±0.0068	0.6021±0.0035	0.632±0.0033
	0.01	0.6398±0.0021	0.6677±0.0053	0.6055±0.0021	0.6351±0.0019
	0.02	0.6397±0.002	0.667±0.0088	0.6069±0.0044	**0.6355±0.0055**
	0.05	0.636±0.0049	0.6626±0.012	0.6046±0.0069	0.6322±0.0083
	0.1	0.6301±0.0047	0.6494±0.0125	0.6205±0.0109	0.6346±0.0085
SVM	0.005	0.6568±0.0024	0.692±0.0086	0.6071±0.0127	0.6467±0.0079
	0.01	0.6567±0.0025	0.6915±0.0071	0.608±0.0126	**0.647±0.0078**
	0.02	0.6571±0.0025	0.6944±0.0085	0.603±0.0134	0.6454±0.0086
	0.05	0.6546±0.0038	0.6906±0.0108	0.6026±0.0118	0.6435±0.0093
	0.1	0.636±0.0054	0.6589±0.0086	0.6151±0.0099	0.6362±0.0087
NB	0.005	0.6142±0.0061	0.663±0.0074	0.518±0.0127	0.5816±0.0097
	0.01	0.6142±0.0054	0.6642±0.0076	0.5149±0.0117	0.5801±0.0097
	0.02	0.6086±0.0041	0.6458±0.0082	0.5401±0.0059	0.5882±0.0066
	0.05	0.5996±0.0064	0.6593±0.0122	0.4684±0.0121	0.5476±0.0122
	0.1	0.5787±0.0114	0.5868±0.0162	0.6295±0.0165	**0.6073±0.013**
kNN	0.005	0.6336±0.0051	0.666±0.0189	0.5879±0.0167	0.6242±0.0044
	0.01	0.6348±0.0052	0.667±0.0194	0.5899±0.0138	0.6257±0.0032
	0.02	0.6319±0.006	0.6636±0.0198	0.5877±0.0133	0.623±0.0025
	0.05	0.6352±0.0056	0.6607±0.0132	0.6067±0.0076	**0.6325±0.0095**
	0.1	0.6065±0.0087	0.6335±0.0193	0.5712±0.0226	0.6003±0.0097
RF	**0.005**	0.6592±0.0018	0.6799±0.0108	0.6457±0.0072	**0.6623±0.0033**
	0.01	0.6567±0.0025	0.6915±0.0071	0.608±0.0126	0.647±0.0078
	0.02	0.6571±0.0025	0.6944±0.0085	0.603±0.0134	0.6454±0.0086
	0.05	0.6546±0.0038	0.6906±0.0108	0.6026±0.0118	0.6435±0.0093
	0.1	0.6282±0.0066	0.66±0.0123	0.5809±0.0124	0.6179±0.0107

To gain insight on which features help to detect PII transmissions, we look at the top 10 highly-correlated features listed in Table. 5.3. We find that applications that have "an activity that is found to be shared with other apps on the device" is correlated to PII transmissions. Furthermore, a number of permissions are also highly correlated. We observe that *both standard Android permissions* such as accessing the device's Wifi state and *custom permissions* such as receiving a broadcast when the device's screen is lit contribute to constructing good models.

Table 5.3: Top 10 most correlated features to PII transmissions in S_1.

Feature number	**Type**	**Correlation**	**Description**
Manifest-7	Manifest analysis	0.2899	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Broadcast Receiver is explicitly exported.
Perm-ACCESS_ WIFI_STATE	Standard permission	0.2769	Allows applications to access information about Wi-Fi networks.
Manifest-6	Manifest analysis	0.2743	A Broadcast Receiver is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device.
Manifest-15	Manifest analysis	0.2611	If taskAffinity is set, then other applications could read the Intents sent to Activities belonging to task.
Manifest-13	Manifest analysis	0.2130	By setting an intent priority higher than another intent, the app effectively overrides other requests.
Manifest-3	Manifest analysis	0.1904	An Activity should not be having the launch mode attribute set to "singleTask/singleInstance" as it becomes root Activity and it is possible for other applications to read the contents of the calling Intent. So it is required to use the "standard" launch mode attribute when sensitive information is included in an Intent.
Perm-WAKE_LOCK	Standard permission	0.1753	Allows using PowerManager WakeLocks to keep the processor from sleeping or screen from dimming.
Perm-WRITE_ SETTINGS	Standard permission	0.1681	Allows an application to read or write the system settings.
Perm-RECEIVE_ USER_PRESENT	Custom permission	0.1420	Allows receive a broadcast when the screen is lit.
Manifest-11	Manifest analysis	0.1288	A Service is found to be shared with other apps on the device therefore leaving it accessible to any other application on the device. The presence of intent-filter indicates that the Service is explicitly exported.

5.3 Classification using the feature set from source code S_2

Features from source code in S_2 take 15-30 seconds to extract, which is longer than features from the manifest file in S_1 that take only 5-10 seconds. In this experiment, we examine if classification using only features from the source code is effective. Table 5.4 shows the results of all six algorithms that are trained using only the features from S_2. We explore various dropout thresholds ranging from low thresholds of 0.005 where we have 133 features to a more selective threshold of 0.1 where we are left with just 65 features as shown in Figure 5.4. Note that without using any dropout threshold, we would have 147 features in S_2.

Overall, across all learning algorithms, the classification performance when using the feature set from source code alone (S_2) has higher average F1 scores than when using the feature set from the manifest file alone (S_1), with increases in average F1 scores ranging from 0.03 to 0.05. When using a low dropout threshold of 0.005, the model with the highest F1 score of 0.6981 is based on Random Forest once again. Furthermore, all algorithms had better precision and significantly better recall.

To gain insight on which features help to detect PII transmissions, we look at the top 10 highly-correlated features listed in Table 5.5. Results from Android API analysis, APKiD analysis and code analysis are highly correlated to PII transmission. In fact the results of the Android API analysis detects calls to PII-related methods such as “Get Device ID, IMEI, MEID/ESN, etc.” and “Get SIM Provider Details” are clearly related to PII transmissions. Similarly, APKiD analysis, and code analysis results related to PII such as “SIM operator” and “subscriber ID” for APKiD analysis and “application uses raw SQL query that could lead to SQL injection and information leakage” are also clearly contributing to the effectiveness of the models.

Table 5.4: Classification performance using the feature set from source code S_2.

Algorithm	**Correlation Threshold**	**Accuracy**	**Precision**	**Recall**	**F1**
NN	0.005	0.7013±0.0118	0.7515±0.0302	0.6363±0.0409	0.6877±0.0129
	0.01	0.7015±0.0088	0.759±0.0308	0.6231±0.0219	0.6836±0.0026
	0.02	0.7005±0.0133	0.7409±0.0362	0.6538±0.0397	**0.693±0.0088**
	0.05	0.6994±0.0057	0.7369±0.0175	0.6533±0.0202	0.6922±0.0058
	0.1	0.698±0.0082	0.7431±0.0089	0.6371±0.0197	0.6858±0.0104
LR	0.005	0.674±0.0069	0.7018±0.0062	0.6435±0.0098	0.6714±0.0078
	0.01	0.6728±0.0078	0.7005±0.0071	0.6425±0.0094	0.6702±0.0082
	0.02	0.6742±0.0075	0.7021±0.0068	0.6436±0.0074	0.6716±0.007
	0.05	0.6744±0.0069	0.7024±0.0052	0.6436±0.0067	0.6717±0.0059
	0.1	0.6734±0.0068	0.7±0.0048	0.6457±0.0061	**0.6718±0.0053**
SVM	0.005	0.6971±0.0054	0.7373±0.0037	0.6445±0.0102	0.6877±0.0064
	0.01	0.6983±0.0048	0.7398±0.0032	0.6435±0.0085	0.6883±0.0059
	0.02	0.6985±0.0048	0.7402±0.0043	0.6432±0.0077	0.6883±0.0057
	0.05	0.6973±0.0072	0.7363±0.0063	0.6467±0.0115	0.6886±0.0087
	0.1	0.6971±0.0061	0.7338±0.005	0.651±0.0093	**0.6899±0.0067**
NB	**0.005**	0.6607±0.0063	0.709±0.0073	0.5842±0.0121	**0.6405±0.0096**
	0.01	0.6603±0.0063	0.7084±0.0071	0.5841±0.0124	0.6402±0.0097
	0.02	0.6594±0.0077	0.709±0.0085	0.5801±0.014	0.638±0.0111
	0.05	0.6557±0.0066	0.7088±0.0094	0.5681±0.0124	0.6307±0.0104
	0.1	0.6542±0.0073	0.7082±0.0102	0.5643±0.0093	0.6281±0.0093
kNN	0.005	0.6761±0.0087	0.7164±0.0134	0.6195±0.0108	0.6644±0.0108
	0.01	0.6762±0.0088	0.7168±0.0135	0.619±0.0116	0.6643±0.0111
	0.02	0.6757±0.0109	0.7167±0.016	0.6174±0.0116	0.6634±0.0124
	0.05	0.6765±0.0091	0.7149±0.0122	0.6238±0.0065	0.6663±0.0083
	0.1	0.677±0.0055	0.711±0.0077	0.6335±0.0095	**0.67±0.0069**
RF	**0.005**	0.6958±0.0069	0.7178±0.0119	0.6796±0.0061	**0.6981±0.0073**
	0.01	0.6957±0.008	0.7184±0.0096	0.6779±0.0053	0.6976±0.0063
	0.02	0.694±0.0072	0.7165±0.0096	0.6766±0.0065	0.696±0.0073
	0.05	0.6943±0.0054	0.7167±0.008	0.6771±0.0074	0.6963±0.0054
	0.1	0.695±0.0072	0.7203±0.0077	0.6714±0.0057	0.695±0.0064

Table 5.5: Top 10 most correlated features to PII transmissions in S_2.

Feature number	Type	Correlation	Description
API-19	Android API	0.3565	Using “Get Device ID, IMEI,MEID/ESN, etc.” API.
APKiD-47	APKiD analysis	0.3397	“SIM operator” is modified by anti VM techniques.
API-7	Android API	0.3372	Using “Get SIM Provider Details” API.
APKiD-46	APKiD analysis	0.3289	“Network operator name” is modified by anti VM techniques.
DANG-7	Code analysis	0.3134	App uses SQLite Database and executes raw SQL query. Untrusted user input in raw SQL queries can cause SQL Injection. Also sensitive information should be encrypted and written to the database.
API-20	Android API	0.3060	Using “Get Subscriber ID” API.
API-21	Android API	0.3046	Using “Get Cell Location” API.
API-40	Android API	0.3030	Using “Get SIM Operator Name” API.
API-24	Android API	0.3000	Using “Sending Broadcast” API.
APKiD-25	APKiD analysis	0.2920	“Build.BOARD” is modified by anti VM techniques.

Table 5.6: Classification performance using the combined feature set from the manifest file and source code S_1+S_2.

Algorithm	**Correlation Threshold S_1**	**Correlation Threshold S_2**	**Accuracy**	**Precision**	**Recall**	**F1**
NN	0.02	0.02	0.6998±0.0065	0.7324±0.0227	0.6653±0.0452	0.6959±0.0161
LR	0.02	0.1	0.6767±0.0071	0.7048±0.0092	0.646±0.0047	0.6741±0.0062
SVM	0.01	0.1	0.6993±0.0062	0.7395±0.0083	0.6467±0.0094	0.69±0.0088
NB	0.1	0.005	0.6592±0.0055	0.7095±0.0069	0.5783±0.0103	0.6372±0.0081
kNN	0.05	0.1	0.6872±0.0076	0.7281±0.0099	0.6314±0.01	0.6763±0.0089
RF	0.005	0.005	0.7016±0.0064	0.7231±0.008	0.6863±0.0084	**0.7042±0.0058**

5.4 Classification using the combined feature set from the manifest file and source code $S_1 + S_2$

We can conclude from the results of the first two experiments that using S_1 or S_2 alone can achieve good detection performance, with S_2 outperforming S_1. According to the high F1 scores, using machine learning on features from fast static code analysis has comparable performance to heavy-weight PII detection.

In this experiment, we examine if combining the features from the manifest file and source code, $S_1 + S_2$, results in even better performance. Table 5.6 shows the result of all 6 algorithms trained using the features from $S_1 + S_2$ with the best dropout threshold for each feature set from the previous experiments. As a result, all learning algorithms have slightly higher F1 scores with score increases ranging from 0.0001 to 0.006 except for Naive Bayes which has lower F1 scores. Again, the Random Forest model had the highest average F1 score of 0.7042. The Neural Network model has a comparable average F1 score to the Random Forest model. However, the Neural Network model requires twice as much time to train as the Random Forest model. In terms of performance and training time, the Random Forest model is a better choice.

To gain more insight, we examine the performance of the 5 random forest models that were trained using 5-fold cross-validation to gain more insight. Table 5.7 shows the performance of all five models, as well as the average and standard deviation. The number of positive (PII transmission) and negative (no PII transmission) labels detected by each model varies slightly. However, they have little effect on the performance metrics as shown by the low standard deviation.

Experimental results indicate that classification effectiveness increases slightly when we used the combined feature set $S_1 + S_2$. However as a trade-off, using only S_2 as can reduce extraction time by 15-30 seconds which may more suitable for real-time use, with a reduced F1 score of only 0.01.

Table 5.7: Classification performance of random forest models using the combined feature set S_1+S_2 while setting correlation threshold of S_1 to 0.005 and S_2 to 0.005.

Model number	**Accuracy**	**Precision**	**Recall**	**F1**	**TP**	**FP**	**TN**	**FN**
1	0.6986	0.7115	0.688	0.6995	1376	558	1364	624
2	0.7017	0.7269	0.688	0.7069	1411	530	1341	640
3	0.6951	0.7192	0.6768	0.6974	1378	538	1348	658
4	0.7121	0.7258	0.6985	0.7119	1395	527	1397	602
5	0.7003	0.7322	0.6801	0.7051	1405	514	1341	661
Average	0.7016	0.7231	0.6863	0.7042	1393	533.4	1358.2	637
SD	0.0064	0.008	0.0084	0.0058	15.7003	16.2419	23.6368	24.5967

We can effectively detect PII transmission using only features extracted by fast static code analysis from MobSF. These features allow us to classify applications not only faster than traditional heavy-weight approaches, but also with comparable accuracy and precision. Experimental results show that the Random Forest model provided the best classification results with a 0.7042 average F1 score. This model can train and classify applications in a very short amount of time. For real-time use of our work, we can further reduce the 45 seconds classification time without sacrificing performance by choosing one of two options: (i) only extracting features from the application manifest file, S_1, for analysis reduces classification time to around 5-10 seconds, resulting in a slightly lower average F1 score of 0.6623, or (ii) only extracting features from the application source code, S_2, for analysis reduces classification time to around 15-30 seconds, resulting in a slightly lower average F1 score of 0.6981.

Chapter VI

CONCLUSION AND FUTURE WORK

In this section, we conclude the experimental results and discuss the future work for detecting PII transmission using features extracted from fast static code analysis.

6.1 Conclusion

We present a novel approach to detecting PII transmissions in mobile applications using features extracted from fast static code analysis to develop a classification model. The Random Forest model provided the best classification results in the experiments, with an F1 score of 0.7042. In a short amount of time, this model can train and classify applications. As a result, we believe our approach is more suitable for real-time application evaluation than existing heavy-weight PII detection approaches. Our work will benefit individual users who want to quickly check if an application transmits PII before installing it on their mobile devices.

Large application distribution markets, such as Google Play Store, may benefit from our work because they must verify that applications clearly and correctly disclose whether or not they transmit PII in order to comply with data privacy laws in various countries. Using our approach, distribution markets can analyze a submitted application for review in under a minute.

6.2 Future Work

The model's performance could be improved by considering a wider range of features, such as features derived from combining static features and features derived from dynamic analysis. Tuning performance for each application category by adjusting Mankey and learning algorithms to matching application behavior in each category may also produce better performance.

Extending the scope of our work to detect which PII is leaked from the application may also be useful. According to the preliminary results, the amount of data for each PII data element is highly imbalanced which makes the problem more challenging.

Lastly, exploring how applications can try to avoid our detection is also important for maintaining robust detection. We believe that application developers may try to avoid detection by tampering or obscuring features obtained through fast static code analysis. For example, they could try to modify the manifest file, which is simple to do, in order to avoid detection by only using features from the manifest (S_1). They may also want to obfuscate the source code (S_2). Some of the features we used in this study already reflect the obfuscation attempt and the detection results for S_2 with obfuscation are already robust. This means that if we use our detection on S_2, which is more resistant to adversaries, we should still be able to detect PII transmissions with good performance.

REFERENCES

[1] ITU, ITU Statistics (2019).
URL https://www.itu.int/en/ITU-D/Statistics/Pages/stat/default.aspx

[2] StatCounter, Mobile Operating System Market Share Worldwide (2019).
URL https://gs.statcounter.com/os-market-share/mobile/worldwide/2019

[3] 42matters, Store Stats for Mobile Apps (2019).
URL https://42matters.com/stats

[4] I. Consulting, General Data Protection Regulation (2019).
URL https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[5] I. Torre, et al., Supporting Users to Take Informed Decisions on Privacy Settings of Personal Devices, Personal Ubiquitous Comput. (2018) 345–364 (2018). doi:10.1007/s00779-017-1068-3.

[6] HP, HP Reveals Applications Vulnerable to Attack (2019).
URL https://www8.hp.com/us/en/hp-news/press-release.html?id=1528865

[7] M. Conti, Q. Q. Li, A. Maragno, R. Spolaor, The Dark Side(-Channel) of Mobile Devices: A Survey on Network Traffic Analysis, IEEE Communications Surveys Tutorials 20 (4) (2018) 2658–2713 (Fourthquarter 2018). doi:10.1109/COMST.2018.2843533.

[8] Z. Yang, M. Yang, Leakminer: Detect Information Leakage on Android with Static Taint Analysis, in: 2012 Third World Congress on Software Engineering, 2012, pp. 101–104 (Nov 2012). doi:10.1109/WCSE.2012.26.

[9] C. Gibler, J. Crussell, J. Erickson, H. Chen, AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale,

in: S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, X. Zhang (Eds.), Trust and Trustworthy Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 291–307 (2012).

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, ACM, New York, NY, USA, 2014, pp. 259–269 (2014). doi:10.1145/2594291.2594299.
URL http://doi.acm.org/10.1145/2594291.2594299s

[11] N. Wongwiwatchai, P. Pongkham, K. Sripanidkulchai, Comprehensive Detection of Vulnerable Personal Information Leaks in Android Applications, in: 2020 IEEE INFOCOM WKSHPS: MobiSec 2020: Security, Privacy, and Digital Forensics of Mobile Systems and Networks, 2020 (2020).

[12] R. Stevens, C. Gibler, J. Crussell, J. Erickson, H. Chen, Investigating user privacy in android ad libraries, in: Workshop on Mobile Security Technologies (MoST), Vol. 10, Citeseer, 2012 (2012).
URL https://pdfs.semanticscholar.org/fa2c/7383769184aae4e301f0361758ae2ddb1daf.pdf

[13] H. Kuzuno, S. Tonami, Signature generation for sensitive information leakage in android applications, in: 2013 IEEE 29th International Conference on Data Engineering Workshops (ICDEW), 2013, pp. 112–119 (April 2013). doi:10.1109/ICDEW.2013.6547438.

[14] A. Rao, A. M. Kakhki, A. Razaghpanah, A. Tang, S. Wang, J. Sherry, P. Gill, A. Krishnamurthy, A. Legout, A. Mislove, et al., Using the middle to meddle with mobile, CCIS, Northeastern University, Tech. Rep., December (2013).

[15] Y. Song, U. Hengartner, PrivacyGuard: A VPN-based Platform to Detect Information Leakage on Android Devices, in: Proceedings of the 5th Annual

ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15, ACM, New York, NY, USA, 2015, pp. 15–26 (2015). doi:10.1145/2808117.2808120.
URL http://doi.acm.org/10.1145/2808117.2808120

[16] J. Ren, A. Rao, M. Lindorfer, A. Legout, D. Choffnes, ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic, in: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '16, ACM, New York, NY, USA, 2016, pp. 361–374 (2016). doi:10.1145/2906388.2906392.
URL http://doi.acm.org/10.1145/2906388.2906392

[17] E. Vanrykel, G. Acar, M. Herrmann, C. Diaz, Leaky Birds: Exploiting Mobile Application Traffic for Surveillance, in: J. Grossklags, B. Preneel (Eds.), Financial Cryptography and Data Security, Vol. 9603, Springer Berlin Heidelberg, Berlin, Heidelberg, 2017, pp. 367–384 (2017). doi:10.1007/978-3-662-54970-4_22.
URL http://link.springer.com/10.1007/978-3-662-54970-4_22

[18] Z. Cheng, X. Chen, Y. Zhang, S. Li, Y. Sang, Detecting Information Theft Based on Mobile Network Flows for Android Users, in: 2017 International Conference on Networking, Architecture, and Storage (NAS), 2017, pp. 1–10 (Aug 2017). doi:10.1109/NAS.2017.8026853.

[19] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, G. Vigna, Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis, in: Proceedings 2017 Network and Distributed System Security Symposium, Internet Society, San Diego, CA, 2017 (2017). doi:10.14722/ndss.2017.23465.
URL https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/obfuscation-resilient-privacy-leak-detection-mobile-apps-through-differential-analysis/

[20] I. Reyes, P. Wiesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, C. Kreibich, “Is Our Children’s Apps Learning?” Automatically Detecting COPPA Violations, In Workshop on Technology and Consumer Protection (ConPro 2017), in conjunction with the 38th IEEE Symposium on Security and Privacy (IEEE S&P 2017), 25 May 2017, San Jose, CA, USA, 2017, p. 6 (May 2017).
URL `http://eprints.networks.imdea.org/1557/`

[21] I. Reyes, P. Wijesekera, J. Reardon, A. E. B. On, A. Razaghpanah, N. Vallina-Rodriguez, S. Egelman, “Won’t Somebody Think of the Children?” Examining COPPA Compliance at Scale, Proceedings on Privacy Enhancing Technologies 2018 (3) (2018) 63–83 (Jun. 2018). `doi:10.1515/popets-2018-0021`.
URL `http://content.sciendo.com/view/journals/popets/2018/3/article-p63.xml`

[22] A. Shuba, E. Bakopoulou, M. A. Mehrabadi, H. Le, D. Choffnes, A. Markopoulou, AntShield: On-Device Detection of Personal Information Exposure, arXiv preprint arXiv:1803.01261 (Mar. 2018).
URL `http://arxiv.org/abs/1803.01261`

[23] A. Shuba, E. Bakopoulou, A. Markopoulou, Privacy Leak Classification on Mobile Devices, in: 2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC), 2018, pp. 1–5 (June 2018). `doi:10.1109/SPAWC.2018.8445948`.

[24] W. Nayam, A. Laolee, L. Charoenwatana, K. Sripanidkulchai, An Analysis of Mobile Application Network Behavior, in: Proceedings of the 12th Asian Internet Engineering Conference, AINTEC ’16, Association for Computing Machinery, New York, NY, USA, 2016, p. 9–16 (2016). `doi:10.1145/3012695.3012697`.
URL `https://doi.org/10.1145/3012695.3012697`

[25] Google, UI/Application Exerciser Monkey Android Developers (2019).
URL `https://developer.android.com/studio/test/monkey`

[26] Appium, appium/appium, original-date: 2013-01-09T22:49:50Z (Sep. 2019).
URL `https://github.com/appium/appium`

[27] K. Mao, M. Harman, Y. Jia, Sapienz: multi-objective automated testing for Android applications, in: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016, ACM Press, 2016 (2016). `doi:10.1145/2931037.2931054`.
URL `https://doi.org/10.1145/2931037.2931054`

[28] S. R. Choudhary, A. Gorla, A. Orso, Automated Test Input Generation for Android: Are We There Yet? (E), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 429–440 (Nov 2015). `doi:10.1109/ASE.2015.89`.

[29] ibotpeaches, Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. (2019).
URL `https://ibotpeaches.github.io/Apktool/`

[30] skylot, skylot/jadx, original-date: 2013-03-18T17:08:21Z (Sep. 2019).
URL `https://github.com/skylot/jadx`

[31] B. Gruver, JesusFreke/smali, original-date: 2012-09-23T19:51:16Z (Sep. 2019).
URL `https://github.com/JesusFreke/smali`

[32] MobSF, MobSF/Mobile-Security-Framework-MobSF, original-date: 2015-01-31T04:36:01Z (Sep. 2019).
URL `https://github.com/MobSF/Mobile-Security-Framework-MobSF`

[33] A. Naway, Y. Li, Android Malware Detection Using Autoencoder, CoRR abs/1901.07315 (2019). `arXiv:1901.07315`.
URL `http://arxiv.org/abs/1901.07315`

[34] Y. Zhang, Y. Yang, X. Wang, A Novel Android Malware Detection Approach Based on Convolutional Neural Network, in: Proceedings of the 2Nd International Conference on Cryptography, Security and Privacy, ICCSP 2018, ACM, New York, NY, USA, 2018, pp. 144–149 (2018). doi:10.1145/3199478.3199492.
URL http://doi.acm.org/10.1145/3199478.3199492

[35] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, Drebin: Effective and Explainable Detection of Android Malware in Your Pocket, in: Proceedings 2014 Network and Distributed System Security Symposium, Internet Society, 2014 (2014). doi:10.14722/ndss.2014.23247.
URL https://doi.org/10.14722/ndss.2014.23247

[36] S. Y. Yerima, G. McWilliams, S. Sezer, Analysis of Bayesian classification-based approaches for Android malware detection, IET Information Security 8 (1) (2014) 25–36 (Jan. 2014). doi:10.1049/iet-ifs.2013.0095.
URL https://doi.org/10.1049/iet-ifs.2013.0095

[37] J. Hegedus, Y. Miche, A. Ilin, A. Lendasse, Methodology for Behavioral-based Malware Analysis and Detection Using Random Projections and K-Nearest Neighbors Classifiers, in: 2011 Seventh International Conference on Computational Intelligence and Security, IEEE, 2011 (Dec. 2011). doi:10.1109/cis.2011.227.
URL https://doi.org/10.1109/cis.2011.227

[38] S. Sachdeva, R. Jolivot, W. Choensawat, Android malware classification based on mobile security framework, IAENG International Journal of Computer Science 45 (2018) 514–522 (01 2018).

[39] Google, API reference: Android Developers (2019).
URL https://developer.android.com/reference

[40] mitmproxy, mitmproxy (2019).
URL https://mitmproxy.org/

[41] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, The journal of machine learning research 15 (1) (2014) 1929–1958 (2014).

[42] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: International conference on machine learning, PMLR, 2015, pp. 448–456 (2015).

[43] T. M. Oshiro, P. S. Perez, J. A. Baranauskas, How many trees in a random forest?, in: International workshop on machine learning and data mining in pattern recognition, Springer, 2012, pp. 154–168 (2012).

[44] S. R. Department, Number of apps installed by U.S. users 2019 (Feb 2021). URL `https://www.statista.com/statistics/267309/number-of-apps-on-mobile-phones/`

Appendix I

COMPREHENSIVE PII LIST

Table A.1: PII defined and detected by previous work and VULPIX.

PII Type	Data Element	Static Analysis			Dynamic Analysis								VULPIX		
		LeakMiner [8]	AndroidLeaks [9]	FlowDroid [10]	ReCon [16]	Leaky Birds [17]	Network Flow [18]	Agrigento [19]	COPPA1 [20]	COPPA2 [21]	AntShield [22]	Shuba et al. [23]	VULPIX$_S$	VULPIX$_D$	This Thesis Dataset
Device	Advertiser ID				✓	✓				✓	✓	✓	✓	✓	✓
	Android ID				✓	✓		✓		✓	✓	✓	✓	✓	✓
	Device Serial Number				*	✓			✓	✓	✓	✓	✓	✓	✓
	Google Services Framework ID									✓			✓	✓	
	IMEI (International Mobile Equipment Identity)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	MAC Address			✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
SIM Card	GSM Cell ID			✓									✓	✓	✓
	ICCID (SIM Serial Number)	✓		✓	✓	✓		✓		✓	✓	✓	✓	✓	✓
	IMSI (International Mobile Subscriber Identity)	✓	✓	✓	✓	✓		✓	✓				✓	✓	✓
	Location Area Code			✓									✓	✓	✓
	Phone Number	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
User	Age						✓							✓	
	Audio			✓					✓				✓	✓	
	Calendar Event	✓											✓	✓	
	Contract Book	✓			✓		✓	✓					✓	✓	✓
	Country			✓									✓	✓	✓
	Credit Card Number and CCV										✓	✓		✓	
	Date Of Birth				✓									✓	
	Email			✓	✓					✓	✓	✓	✓	✓	✓
	Gender				✓						✓	✓		✓	✓
	Name				✓				✓		✓	✓		✓	✓
	Password			✓	✓						✓	✓	✓	✓	✓
	Photo						✓		✓				✓	✓	
	Physical Address				✓				✓		✓	✓		✓	✓
	Relationship Status				✓									✓	
	SMS Message	✓					✓						✓	✓	
	SSN (Social Security Number)								✓					✓	
	Time Zone			✓									✓	✓	✓
	Username			✓	✓				✓		✓	✓	✓	✓	✓
	Video								✓				✓	✓	
	Web Browsing Log			✓			✓						✓	✓	
Location	GPS (Latitude and Longitude)	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓

* Not stated in their paper, but defined in their source code

Appendix II

HYPERPARAMETER EVALUATION

Table B.1: Classification performance using different hyperparameters configured for each learning algorithms.

Algorithm	Hyperparameter	Accuracy	Precision	Recall	F1
NN	3 FC layers	0.7011±0.0071	0.7452±0.0087	0.6419±0.0152	0.6896±0.0097
	4 FC layers	0.6995±0.0088	0.7398±0.0228	0.6489±0.0264	**0.6908±0.0116**
	5 FC layers	0.7012±0.0059	0.7469±0.0236	0.6415±0.0254	0.6895±0.0082
	6 FC layers	0.7023±0.009	0.7517±0.0239	0.6363±0.0163	0.6887±0.0033
LR	**solver = lbfgs**	0.6768±0.0052	0.7066±0.0092	0.6422±0.0045	**0.6729±0.0062**
	solver = newton-cg	0.6762±0.0059	0.7064±0.0097	0.641±0.0045	0.6721±0.0068
	solver = sag	0.6763±0.0059	0.7064±0.0096	0.641±0.0044	0.6721±0.0066
	solver = saga	0.6762±0.0059	0.706±0.0096	0.6416±0.005	0.6722±0.007
SVM	kernel = linear	0.6782±0.0048	0.7196±0.011	0.6202±0.0039	0.6662±0.0049
	kernel = poly	0.7011±0.0041	0.7669±0.0083	0.607±0.0109	0.6776±0.009
	kernel = rbf	0.6998±0.0055	0.7403±0.0087	0.6468±0.0097	**0.6904±0.0091**
	kernel = sigmoid	0.5691±0.006	0.5839±0.011	0.5825±0.0074	0.5832±0.0079
NB	Gaussian NB	0.4984±0.0109	0.7445±0.0258	0.0471±0.0108	0.0883±0.019
	Multinomial NB	0.6598±0.0036	0.7114±0.0082	0.5768±0.007	**0.6371±0.0067**
kNN	k = 3	0.6716±0.0037	0.6996±0.009	0.6409±0.0125	0.6689±0.0069
	k = 5	0.678±0.0061	0.7123±0.0097	0.6341±0.0133	0.6709±0.0098
	k = 7	0.6867±0.0079	0.7294±0.0121	0.6271±0.0138	**0.6744±0.0121**
	k = 9	0.689±0.0072	0.7375±0.0115	0.6195±0.0155	0.6733±0.0126
RF	trees = 16	0.69±0.0053	0.7171±0.0103	0.6625±0.0063	0.6887±0.0056
	trees = 32	0.69±0.0044	0.7146±0.0081	0.6679±0.0049	0.6904±0.0034
	trees = 64	0.6999±0.0023	0.7232±0.0072	0.681±0.0083	0.7014±0.0043
	trees = 128	0.7008±0.0028	0.7218±0.0103	0.6867±0.0055	**0.7038±0.0047**
	trees = 256	0.7022±0.0033	0.7244±0.0067	0.6855±0.0065	0.7043±0.0043
	trees = 512	0.7027±0.004	0.7244±0.0097	0.6871±0.0061	0.7052±0.0047
	trees = 1024	0.7027±0.0041	0.7249±0.0081	0.6859±0.0065	0.7048±0.0049

Table B.1 depicts the performance of all 6 algorithms trained using different hyperparameters. We use the full set of features from both S_1 and S_2. The table shows the performance metrics of each model in the form of the average value and the standard deviation of the five models that were created during training using 5-fold cross-validation. Higher average value for each metric means better performance for the model. And, lower standard deviation for each metric indicates more stable prediction performance for the model. As an example, consider the the first row for the NN (Neural Network) model that uses 3 full connected layers. This model has 0.7011 average accuracy and 0.0071 standard deviation. We explore various hyperparameters available for each algorithm.

With the exception of the Random Forest model, the hyperparameter that gives a model its best F1 score is chosen to use in experiments. We use 128 trees in Random Forest because there is no significant difference in F1 scores between models with 128, 256, 512, and 1,024 trees. Because increasing the number of trees also increases the training time, using 128 trees in the model is a good performance trade off with a reasonable training time [43].

Appendix III

LIST OF PUBLICATIONS

1. N. Wongwiwatchai, P. Pongkham, K. Sripanidkulchai, Comprehensive Detection of Vulnerable Personal Information Leaks in Android Applications, in: 2020 IEEE INFOCOM WKSHPS: MobiSec 2020: Security, Privacy, and Digital Forensics of Mobile Systems and Networks, 2020 (2020).

2. N. Wongwiwatchai, P. Pongkham, K. Sripanidkulchai, Detecting personally identifiable information transmission in android applications using lightweight static analysis, Computers & Security 99 (2020) 102011 (2020).

Biography

Nattanon Wongwiwatchai is a graduate student in the Department of Computer Engineering at Chulalongkorn University, Thailand. He received his B.Eng. in Computer Engineering from Chulalongkorn University in 2019. His research interests are in mobile application security and privacy, and data analysis.