# APPLYING SEMANTIC ANALYSIS FOR THE ALIGNMENT OF NATURAL LANGUAGE PRIVACY POLICIES WITH APPLICATION CODE

by

ROCKY SLAVIN, B.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

COMMITTEE MEMBERS:
Jianwei Niu, Ph.D., Chair
Ram Krishnan, Ph.D.
Tongping Liu, Ph.D.
Xiaoyin Wang, Ph.D.
Meng Yu, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
August 2017

## DEDICATION

*For my daughter, Elene, and my wife, Sarah. You are the two halves of my heart.*

## ACKNOWLEDGEMENTS

I would first like to thank Dr. Jianwei Niu for her guidance and support. I could ask for no better advisor.

I am also grateful to my colleagues, friends, and coauthors, Jaspreet Bhatia, Dr. Travis D. Breaux, Dr. Omar Chowdhury, James Hester, Hanan Hibshi, Mitra Bokaei Hosseini, Apostolos Kotsiolis, Dr. Ram Krishnan, Jean-Michel Lehker, Dr. Hui Shen, and Dr. Xiaoyin Wang. My achievements would be nothing without the insight and expertise of these people. Thank you all for giving me the great honor of working with you.

I thank my family for having faith in me for all this time. Thank you mom, dad, and Rio. Most importantly, thank you Sarah. You are a woman of immense patience. Thank you for keeping me going.

Lastly, I thank my dear friend Dr. Andreas Gampe for everything he has done to support and encourage me. Without his persistence, I would not be where I am today. Most importantly, he taught me what it means to have strength of the heart. It's been a long road, my friend.

This work was supported, in part, by the National Security Agency (NSA) grant on Science of Security Research Lablet and National Science Foundation (NSF) grant No. CNS-0964710. Ontology construction was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

August 2017

*This Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.*

*It is acceptable for this Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Doctoral Dissertation attest to the accuracy of this statement.*

**APPLYING SEMANTIC ANALYSIS FOR THE ALIGNMENT OF NATURAL LANGUAGE PRIVACY POLICIES WITH APPLICATION CODE**

Rocky Slavin, Ph.D.
The University of Texas at San Antonio, 2017

Supervising Professor: Jianwei Niu, Ph.D.

Privacy is increasingly becoming a major concern as the convenience of technology grows. In an effort to disclose privacy-related practices and follow federal guidelines for software applications, many publishers provide natural language privacy policies disclosing what private information may be used, collected, or shared by the application. However, these policies are a completely separate body from the code and are written in natural language, thus being susceptible to, sometimes unintentional, misalignments. Such misalignments not only affect end-user privacy, but can result in costly fines and audits for developers.

To mitigate this problem and improve developer and end-user confidence, I introduce a scalable framework for aligning natural language privacy policies with application code. The approach utilizes a phrase to API mapping which effectively bridges the semantic gap between code and policy. In turn, the mapping provides the basis to facilitate static and dynamic code analysis for the detection of privacy misalignments. Included in the approach are natural language processing techniques to determine relevant policy phrases and identify intent with regard to collection (i.e. contextual polarity). Practical efficacy is demonstrated through an instantiation of this framework upon the Android OS environment and the successful detection of misalignments among top applications. Furthermore, I introduce the POLIDROID tool suite which includes practical alignment tools for source and byte code. These tools provide developers, end users, and auditors with the means to apply this framework to real-world situations. Finally, I present an empirical evaluation of the framework and the tools based on the framework which exhibits positive results with regard to precision, performance, and usability.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1: INTRODUCTION

As convenience increases with the rise of the information age, privacy is, in turn, becoming a growing concern. For example, an individual's medical history is generally stored in ways that allow medical professionals easy access to important information, but also results in databases of highly sensitive information which, if leaked, could expose such private information. Similarly, mobile applications frequently access sensitive personal information to meet user or business requirements. This is particularly notable due to the various sensors mobile applications have access to (e.g., global positioning, near-field communication, Bluetooth, etc). Regulators increasingly require application developers to publish information in natural language privacy policies that describe what is collected. Unfortunately, the relationship between these privacy policies and the actual code they represent is often vague, convoluted, or simply inaccurate. This is due to the semantic gap that exists between source code written by developers, and the natural language used in policies which are often written by legal experts. The goal of this research is to mitigate this problem by developing a grounds for developing tools and techniques with which auditors, developers, policy writers, and end users can check applications for alignment with their privacy policies.

To demonstrate the efficacy of the approach, I especially focus on mobile apps developed by third-party developers for the Android operating system (Android). This domain is particularly important to privacy due to its vast integration to the general public and the high number of sensors available on most Android devices. In early 2015, Android accounted for 78.0% of the worldwide smartphone market share [4]. With this sizable presence comes an increase to end user privacy risk since mobile applications (apps) built for Android have access to sensitive personal information such as users' locations, network information, and unique device information. The rise of the smart phone in general has introduced many luxuries to the general public. For example, a standard map application can provide information about traffic, directions, and a bird's-eye view of the area all based on the user's real-time location. This kind of innovation is not readily available on a conventional computing system. Sensors and features such as global positioning, carrier-based

Internet connectivity, Bluetooth, near-field communication (NFC) enable various conveniences through Android devices. Furthermore, the ability for any developer or software company to produce their own apps for these devices gives them the direct access these sensors and provides a wide range of functionality for end users. With this convenience, however, comes a risk to privacy. Since apps are generally not open-sourced, it is possible for an app to access sensitive information (e.g., precise geographical location, unique device identifiers, etc.) without the user realizing it. This means a certain level of trust must be given to both the developer who created the app as well as the operating system that interfaces the app to the device and its sensors. In an effort to protect privacy in such an environment, regulators such as the U.S. Federal Trade Commission (FTC) have relied on natural language privacy policies to enumerate how applications collect, use, and share personal information. Furthermore, the California Attorney General Kamela Harris negotiated with the Google Play app store in 2013 to require mobile app developers to post privacy policies [33], thus providing a large corpus for study. For these reasons, I limit the scope of this research to Android. However, the philosophy is generalizable to arbitrary systems where policy and code alignment is relevant.

**Problem** Despite these efforts to produce policies, as with any software documentation, there are opportunities for them to become inconsistent with the code. These policies can be written by people other than the developers, such as lawyers, or the code can change while the policy remains static. Such inconsistencies regarding an end user's personal data, intentional or not, can have legal repercussions that can be avoided with proper consistency checks. For example, the FTC, under their unfair and deceptive trade practices authority, requires companies to be honest about their data practices in their privacy policies. Companies, such as SnapChat, Fandango, and Credit Karma, often settle with the FTC for inconsistent policies and practices by accepting 20 years of costly privacy and security audits [1, 2]. It is therefore good practice for mobile apps to clearly state in their privacy policies what data is collected and for what purpose.

Addressing this policy-code inconsistency problem can have positive effects for at least four different stakeholders. First, it is important for **developers** to be aware of the data their code

is collecting along with what their policy says they are collecting not only for legal and ethical reasons, but for the production of quality apps. As more data is entrusted to technology, end users become more aware of the ramifications of mishandled private data [59]. Thus, software engineers are entrusted by end users to not only care for their data, but disclose what exactly is being collected. For large companies, this task is commonly assigned to a team of legal experts. However, mobile app developers are frequently small start-ups with 1-5 developers [31] where such a task is not easily assigned and thus poses a larger problem. On the other hand, even if the company has expert **policy writers**, they may not be knowledgeable in the corresponding code. Tools for enabling the interpretation of the code can help to mitigate misalignments that may otherwise unintentionally be produced. For **auditors**, an automated misalignment tool would directly improve their primary focus. Misalignment detection may be particularly difficult for auditors since they may be unfamiliar with *both* the policy language and the application code. Perhaps the most affected stakeholder in problem is the **end user**. Even with a clear privacy policy, the app's user must place some level of trust in the developer that they are adhering to their privacy policy. Furthermore, the average end user may not be able to clearly understand what a privacy policy is attempting to convey. With the necessary tools enabled by this research, the end user would have the ability to verify the honestly of the policy in question and thus feel more confident in the use of the app. In essence, a solid basis to identify and reconcile inconsistencies between code and policy would have a great potential impact on anyone with a stake in privacy.

## 1.1 Challenges

The major challenge of this approach is to relate natural language to code. Android application source code is typically written in the Java programming language and compiled into Dalvik byte-code, neither of which are made to convey semantics in a similar manner to English natural language. Even with a direct mapping between natural language and code, many challenges exist to create a usable framework for misalignment detection. The following are the main challenges for this research.

**Policy Enforceability** Privacy policies alone can be unenforceable due to logically-contradicting clauses. For example, if a policy states **a:** "we only collect location information" and **b:** "in order to send friend requests, we collect your list of contacts", it is impossible for both **a** and **b** to be true. To better focus the scope of this project, I do not consider such possibilities.

**Sentiment Analysis** A difficult problem in natural language processing is that of determining contextual polarity, or sentiment analysis (i.e., whether a sentence is positive or negative). For example, the phrases "we do *not* collect your location" and "we collect your location when the app is *not* running in the foreground" have two different different meanings, but as exemplified in the latter phrase, simply identifying negating words will not always produce correct results. Natural language, in this case English, has virtually infinite ways of relaying meaning. In the context of privacy policies, this is a particularly important challenge since policy statements can be both expressed in a negative or positive manner (e.g., "we *do* collect…" and "we *do not* collect…").

**Scalability** This approach should be scalable to a large number of apps. The Google Play store alone has over 2.4 million apps as of September 2016 [83]. This implies a diverse set of policies and code. This framework should also be applicable to the apps at large regardless of the varying development or writing styles. Furthermore, an auditor should be able to use the framework on large sets of data. If preliminary preparation for each app or policy is required (e.g., formalization), the usability of the approach decreases as the amount of apps increase. A major pitfall of form policy specification languages is that they are not scalable to large amounts of apps since they would first require translation to the language.

**Precision** Even if the approach is scalable, it must provide a certain level of precision. Due to the dynamic nature of the apps developed for Android, guaranteeing both precision and efficiency of static analysis is difficult. For example, FlowDroid, one of the most precise taint analysis tools for Android exhibits an average precision of 86% and can only handle reflective calls if their arguments are string constants [9]. In the area of privacy where sensitivity and liability are major concerns,

high precision is an important factor in detecting misalignments. In order to reduce false positives to a more reliable level, other approaches such as runtime analysis should be utilized. Precision also comes into account when detecting method invocations. Simply invoking a privacy-sensitive method such as `getLongitude()` does not necessarily constitute a privacy leak. A map app, for instance, needs to access this method to detect user location, but it may not necessarily leak that data. The information produced by a method must be collected or otherwise sent away from the device to be considered a breach of privacy. For this reason, data flow analysis is necessary to detect invocations that are not simply used for local functionality.

**Runtime Monitoring** Dynamic analysis is difficult due to the nature of the ecosystem. Features such as callbacks and heavy user input are unpredictable. Source code is not always available and thus direct instrumentation is not feasible. A valid solution for this approach should allow for real-time monitoring of an app at runtime without necessitating direct access to the source code. Furthermore, for the sake of usability, the monitoring should be practical enough to be used by developers and app users alike.

**Generalizability** Though this research focuses on Android as a case study, the philosophy involved should be generalizable to other systems and thus provide confidence for policy alignment in general. Privacy policies exist in various domains of software. The contributions of this research focus on the extraction of natural language from such policies and their relationship to code. Mapping, information flow analysis, and natural language processing can be instantiated using the relevant tools for other domains. Thus, the approaches used in this research should be able to be leveraged in such cases.

**Usability** In solving these challenges, a level of practicality must also be maintained. To this end, usability must be taken into account for the solutions to the previously described challenges. For example, precision can be improved by increasing the memory space, however the average developer or end user does not have access to the necessary technology to scale in this way. Simi-

larly, runtime monitoring should be applicable to the standard Android devices used by the general public. In essence, to maintain the core goals of this framework, it should be applicable to end user tools. Furthermore, it should be automated to a point that usability is not lost.

## 1.2 State of the Art

Some approaches exist that both attempt to formalize natural language privacy policies and improve policy-code alignment in the Android environment.

Often in the case of Android, its built-in permission system is used as a basis for detecting misalignments. Because of their level of granularity, permissions alone are not sufficient to reconcile a policy's statements with the app's actual behavior. For example, an app's policy may claim to only access your bearing (i.e., the direction you are facing), but the relevant permission, `ACCESS_FINE_LOCATION`, provides access to more sensitive information such as longitude and latitude which convey your precise location. However, many privacy tools for Android focus on permissions [10, 28, 84]. Furthermore, these tools use static analysis techniques to detect permission access. As mentioned above, static analysis on Android apps has limited precision in its current state. In order to be applicable to a sensitive topic such as privacy, alternative detection techniques must be integrated. Furthermore, the permission-based approaches are not necessarily generalizable to other domains.

Similarly, existing work by Petronella includes a tool that relates natural language in privacy policies to Android permissions [60]. The tool works by providing the user with the list of permissions for an app along with the sentences from the privacy policy that are related to each permission. This is similar to my work in that it maps natural language phrases to potential data collection actions in the app's implementation. This permission-based approach is however, limited to the granularity of the Android permission system. The phrase-permission mapping is also limited to simple string searching with limited natural language processing and no detection of contextual polarity. In order to create a precise and robust alignment tool, a more rigorous semantic analysis is necessary.

Developer-oriented approaches exist as IDE plugin implementations [13, 69]. These plugins focus on implementing policies along side the development of an app. The basis for these tools, however, is limited to the knowledge of the developer or the permissions accessed by the app. This has the result of decreased scalability and precision. In order to provide a better resource for developers, a wider-scale approach that is adaptable to multiple policy and development styles would provide better impact.

Much work exists in formalizing privacy policies for model checking using specification languages [11, 25, 30, 47, 67]. These approaches can precisely model natural language and accurately verify consistency. They are, however, neither scalable nor easily-usable for target audience of this work. In practice, it is difficult to accurately formalize a policy and end users are not trained to read such formalizations. Verification would require each policy to be formalized, which is not viable for large-scale operations. This research puts emphasis on automated natural language processing in order to remove the burden from the user.

## 1.3 Approach

The semantic gap between natural language privacy policies and their corresponding application code can be effectively bridged by empirically mapping privacy-relevant phrases to information-producing programming language elements thus providing a basis for static and dynamic analyses for the reconciliation of arbitrary apps and their corresponding privacy policies. Consequently, such techniques can be beneficial to developers, auditors, policy writers, and end users through practical tools for policy-code alignment.

I demonstrate this approach through a manually-generated mapping as a proof-of-concept from which a more scalable approach can be developed. Such mapping can then be used in turn with information flow analysis to compare natural language clauses with relevant programming languages. Thus, relationships in semantics can be utilized in the expansion of this framework through static and dynamic analysis of code and natural language to a full privacy misalignment ecosystem. Chapter 7 exemplifies the practical uses and benefits of this approach with publicly-available and

open source applications which utilize the code-phrase mapping to identify policy misalignments in compiled Android apps and Android app source code. An evaluation of this approach, including static and dynamic misalignment detection based on the mapping as well as related tools, is described in Chapter 9.

## 1.4 Contributions

This body of research includes multiple works on privacy policies which contribute to the misalignment detection approach therein. I introduce these works here.

### 1.4.1 API-Phrase Mapping

As a basis for an automated natural-language-oriented approach, we have constructed an empirically-based mapping to link the code to relevant privacy-related English phrases [78]. The mapping is tied to the Android API due to its direct relationship to information-producing sensors (Section 2.2.2). The construction of the mapping was designed to use the popular crowd-sourcing tool Amazon Mechanical Turk[1] to produce the necessary lexicons for both API and privacy policy language. The mapping described in this work (Section 4.1) is an instantiation of the methodology which can be both expanded to be more comprehensive and instantiated for arbitrary domains of interest.

### 1.4.2 Static Misalignment Detection

By having a mobile privacy phrase to API mapping, many approaches can be taken for misalignment detection. The initial contribution based on this mapping is PVDetector, a tool for statically analyzing information flows from private-information-producing API methods to network sinks. Such flows signify a potential privacy leak in that the private information is sent out to the network. PVDetector uses state-of-the-art information flow analysis tools to detect potential data leaks based on the interpretation of data collection parsed from natural language privacy policies.

[1] http://mturk.com

This tool serves not only as a contribution to privacy detection for developers, auditors, and end users, but as validation of the API to phrase mapping described above. PVDetector and its methods for statically analyzing code and policies to detect misalignments are described in Chapter 5.

### 1.4.3 Runtime Misalignment Detection

Static information flow analysis is useful in arriving at a more complete inspection of the code with regard to misalignment detection. This generally comes at the cost of the time involved to perform a full analysis. To complement this approach, I extend the misalignment framework with runtime detection of policy misalignments. The tool, Parameter Reporter, utilizes existing libraries to modify app code directly on the device at runtime and report relevant data leaks based on known network sinks. This dynamic misalignment detector provides the means to verify static detection findings, detect leaks that are not based on API calls (i.e., those implemented with native code), and improve precision with regard to the specific use cases in which the leak occurs all at runtime with minimal overhead. Parameter Reporter and its method for detecting misalignments at runtime are described in Chapter 6.

### 1.4.4 Polidroid Tool Suite

I have developed an initial tool suite, POLIDROID[2], based on the current state of the misalignment framework as a means to demonstrate the efficacy of this work. POLIDROID includes various tools for aligning policies with code. These tools have been created as a demonstration of the practical application of the misalignment framework. Being part of a web-based suite, POLIDROID is designed to be used by developers, auditors, policy writers, and end users. The tool includes both byte code and source code analysis for the use of compiled and in-development apps, respectively. Furthermore, I have developed a plugin for the predominant Android IDE, Android Studio, that specializes in misalignment detection directly during the development process [79]. The plugin provides the developer with real-time information regarding the alignment of their source code

[2]http://polidroid.org

with privacy policy. The plugin, PoliDroid-AS, also has the capability of generating privacy-oriented specifications with which developers can generate accurate, code-based privacy policies. POLIDROID and its constituent tools are described in Chapter 7.

### 1.4.5 Contextual Polarity

A difficulty in natural language processing is the ability to detect the contextual polarity of a sentence. Specifically, the negation of clauses (e.g., "We *do not* collect your information") are not trivial to detect and are of particular importance to privacy policies since the ultimate goal is to parse what information is or is not collected by the application. To address this issue, I have incorporated a Sentiment Treebank [82] to the natural language processing of privacy policies to estimate the contextual polarity of phrases regarding data collection. By identifying the sentiment of such collection phrases, misalignment detections where the natural language phrase was misinterpreted as indicating data collection when the opposite was meant (i.e., false positives) can be reduced. For example, a policy may state "We *do not* collect your location information" in which a simple text search for data-related phrases would parse the meaning as collecting location information when, in fact, the opposite is true. It should be noted that detecting contextual polarity is not so trivial as detecting negative words (e.g., "not") as phrases such as "We collect information including, but **not** limited to,..." and "We collect information when the app is **not** in the foreground" are common, but not easily anticipated. Therefore, incorporating sentiment analysis improves the robustness of the approach. The inclusion of sentiment analysis to the POLIDROID tool suite is described in Chapter 8.

### 1.4.6 Security Requirements Pattern Selection

An underlying issue in privacy policies is the correctness of data security practices. Without strong security requirements, privacy policies can fail. To this end, I have constructed an approach for storing and selecting security requirements patterns which I have shown to improve pattern selection practices among novice users (i.e., non-experts with regard to security) [76]. The approach

utilizes feature diagram hierarchies and the Inquiry Cycle model [62] to guide software practitioners to the most appropriate security requirements patterns for their situation. Privacy policies are based on the assumption that data practices are secure and unknown leaks do not exist. Improving security pattern selection can foster more secure software, thus strengthening such assumptions. The approach for improving pattern selection is described in Section 10.2.

### 1.4.7 Sequence Diagram Aided Policy Specification

As a preamble to this work, I have implemented a tool, Sequence Diagram Translator (SDT), which translates privacy policies and regulations (e.g., Health Insurance Portability and Accountability Act (HIPAA)) represented as UML2 sequence diagrams [55] into linear temporal logic (LTL) [74]. Flows and policies can then be validated for consistency using existing model checkers (e.g., NuSMV). The graphical nature of Sequence Diagrams allow domain experts to visually confirm the intuition, for example, behind HIPAA privacy rules. Once confirmed, those Sequence Diagrams can be formally codified using SDT and analyzed. Unfortunately, such an approach, though highly precise, is not feasible for developers and end users due to the complexities of the intermediate representations and the limited scalability of model checking. However, this exemplifies the underlying issue of complex policy specification and serves as motivation for the remainder of the research. The approach for specifying policies through sequence diagrams is described in Section 10.1.

### 1.4.8 Empirical Evaluation

By implementing practical tools to exemplify the misalignment framework, evaluation can be done over real-world applications and app developers. To this end, Chapter 9 describes the validation of the static and dynamic analysis tools based upon this work.

The static analysis approach used in PVDetector was evaluated through a case study of the top 300 apps from Google Play to verify the feasibility of using such a mapping for misalignment detection (Section 9.1). As a result, 402 potential misalignments were detected with an accuracy

of 80%. Evaluation of the dynamic analysis was conducted as a case study over the detection of misalignments predicted using an existing static analysis approach. The approach by Wang et al. predicts potential policy misalignments caused by native code (i.e., not API calls as with this work) [86]. By using these predictions as a baseline of potential misalignments, Parameter Reporter's ability to detect such misalignments in real-time was validated as being 100% accurate in that it was able to verify every predicted misalignment. Furthermore, the overhead incurred by using the Xposed framework and the Parameter Reporter module was found to be approximately 0.6ms per suspected call. This evaluation is described in Section 9.2. Finally, a user study demonstrating the usability and effectiveness of the POLIDROID tool suite is described in Section 9.3.

# CHAPTER 2: BACKGROUND

In this chapter, I summarize the necessary background information for understanding this project's contributions.

## 2.1 Mobile Ecosystem

The methods in this research are applied to mobile applications due to the wide variety of sensors commonly utilized by mobile applications. Typical sensors include Global Positioning System (GPS) sensors, Near Field Communicators (NFC), cameras, microphones, Bluetooth, network interfaces, and accelerometers among many others. The mobile ecosystem itself consists of various operating systems (e.g., Android OS, iOS, Symbian, Windows OS, etc) which utilize these device sensors to provide functionality to native applications. For example, a map application may access the device's GPS sensor to notify the user precisely where they are geographically. Such functionality, which is not as readily available in other common environments, makes the mobile ecosystem a prime domain for demonstrating this work.

## 2.2 Android OS

Android is an open source mobile operating system (OS) based on more than 100 open source projects including the Linux kernel. Android is developed by Google and has been reported more popular as a target platform for developers than Apple's mobile OS, iOS, in 2015 which makes it the most popular mobile operating system today [3]. Apps made to run on Android can be downloaded from multiple repositories, the most popular being Google Play[1]. In 2014, Google revealed that there were more than one billion active monthly Android users [87]. These characteristics make it attractive to startups and established companies alike.

Android provides a publicly-available software development kit (SDK) which includes the necessary endpoints and documentation for thorough analysis as well as a common application struc-

[1] https://play.google.com/

ture among the majority of its apps. For these reasons, I limit the scope of this project and its case studies to apps within the Android environment. This does not, however, mean the methodology and ideas presented cannot be applied to other environments.

### 2.2.1 Permissions

Android utilizes the Linux security model and layers through a user-based permission system [94]. Apps can access resources through the permission system to resources such as the camera, GPS location, network connections, and other sensors [94]. Such permissions are granted to apps by users at installation time or when the app is running and performing operations that require the permissions at real-time. Besides the design defects such as misuses [27], low granularity [51], and ambiguity [40], Android's permission system is essentially insufficient in privacy protection and privacy-policy enforcement for two main reasons. First, the permission system merely controls the access of private information, but not where the information flows to and the corresponding purposes. Users may accept usage of their location for the app's functionality, but do not accept storing their location at a remote server or selling their location to third-parties. Second, the permission system can handle only private information from the Android platform, and cannot handle private information provided directly by users and third parties.

### 2.2.2 Android API

Applications can interact with underlying Android system using a framework API provided by the Android platform. The framework API contains set of packages, classes, and methods. The Android 4.2 framework is comprised of about 110,000 methods, some of which are specifically used to retrieve, insert, update, or delete sensor data through the Android OS [63]. The use of an API increases the level of security by not allowing apps to have direct access to all sensor data by default.

Before an app can access specific methods from the API, the required permissions must be requested by the app through a manifest file. An app's manifest file enumerates the app's required

permissions and is described to users when installing an app as well as on the app's download page on the Google Play store. Thus, there is a direct relationship between the permissions granted to an application by a user at installation time and eligible API method calls in the application source code.

## 2.3 Privacy Policies

A privacy policy, also called a privacy notice, serves as the primary means to communicate with users regarding what private information will be accessed, collected, stored, shared (app to app, and to third party) and used, as well as the purpose of the information collection and processing. Privacy policies generally consist of multiple paragraphs of natural language, such as the following excerpt from the Indeed Job Search app's privacy policy[2] listed on Google Play:

> Indeed may create and assign to your device an identifier that is similar to an account number. We may collect the name you have associated with your device, device type, telephone number, country, and any other information you choose to provide, such as user name, geo-location or e-mail address. We may also access your contacts to enable you to invite friends to join you in the Website.

One reason privacy policies are so important is that the United States takes a "notice and choice" approach to address privacy online [65]. Under this framework, app companies post their privacy policies and users are expected to read the policies and make informed decision on whether to accept the privacy terms before installing the apps [65]. However, most privacy policies prepared by policy authors are difficult to understand due to their verbose and ambiguous nature, and this can lead to users who skip reading policies, even if they have concerns about the app's information collection practices. A major contributor to this problem is the lack of a canonical format for presenting the information. The language, organization, and detail of policies can vary from app to app. More significantly, these problems can lead to app developers having difficulty

[2] http://www.indeed.com/legal

complying with their own privacy policies effectively. To address these issues, this framework provides a means to achieve alignment between apps' privacy policies and implementation code, and to provide better communication among software developers and policy writers.

Privacy policy *regulations* also exist as public law. For example, the Health Insurance Portability and Accountability Act (HIPAA) [5] regulates the sharing of private health information among multiple parties. Such regulations are similar to privacy policies provided by app developers in that they are described in natural language what and how information can be shared among parties. For this reason, the approach presented here is applicable to privacy regulations. However, the scope of this work is limited to the privacy policies provided with the apps in question so privacy policy regulations are not considered in its evaluation.

# CHAPTER 3: OVERVIEW

In order to create a framework for aligning privacy policies with mobile app code, there are multiple facets of application running and development as well as the application environment that must be addressed. Figure 3.1 illustrates a general overview of the approach and includes interactions by various stakeholders. The approach uses a combination of information flow analysis and natural language processing to analyze interactions and produce relevant information for the alignment of code and policy. Much of the implementation is represented within the POLIDROID box in the figure. More on the POLIDROID tool suite is covered in Section 7.

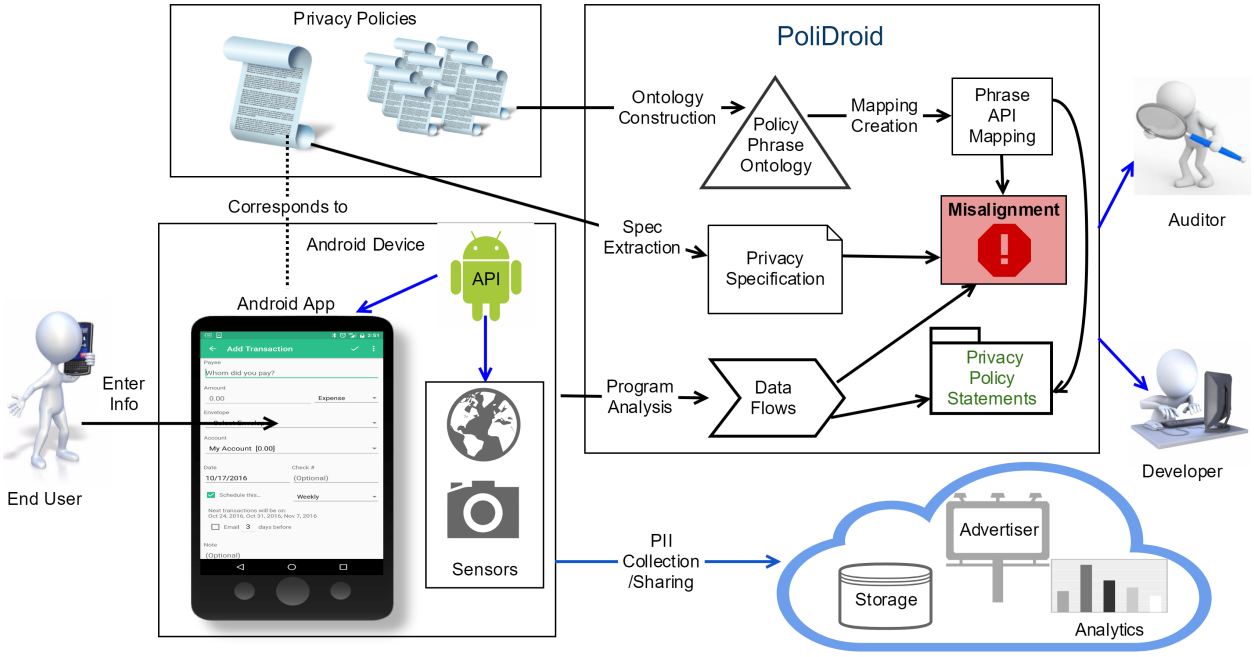



**Figure 3.1**: Overview

**Information Retrieval** Private information can generally be provided to an app in two primary ways: direct input through the user interface or through a third party API, and OS-level input from the physical device itself (e.g., GPS coordinates, camera, unique identifiers). For the first type, the app in question will have specific code for accessing the data. For example, if the app requires the user's first name, it may display a form with a text input labeled "First Name" in which

the user purposefully types their name in directly. An alternative is for the app to access a third party, such as Facebook, through a public API in order to retrieve the information [43]. In either case, the data access and information flow thereafter is dependent on how the developer chooses to implement it. On the other hand, if the app needs to access information from a sensor, such as GPS coordinates, a specific OS-level API method must be accessed to retrieve the information. These two types of information retrieval must be taken into account in order to more robustly detect misalignments. This work addresses private data collection through both methods. To this end, Chapter 5 describes the use of static analysis for the detection of policy misalignments based on Android API invocations and Chapter 6 describes runtime monitoring techniques for the detection of misalignments based on both API-produced data and user input.

**Privacy Policies** The language and layout of a privacy policy can vary from app to app. Often, these policies are embedded within larger legal texts (e.g., end user license agreements, terms of use, etc). Hurdles such as these require some level of parsing in order to extract the relevant information for aligning with app code. Section 4.1.2 describes how we generated a lexicon of privacy-relevant phrases from mobile application privacy policies. These phrases are used in combination with the Stanford CoreNLP tool to determine the relevant clauses describing the collection of private information. To establish a scope for this research, I focus strictly on privacy policies written in English.

**Phrase-Code Mapping** Misalignment between code and policy implies a semantic relationship between the two. In order for a privacy policy to align with its corresponding code, the code must adhere to the promises within the policy. Section 4.1 describes the approach we took involving mapping API-level methods to privacy-relevant phrases. This mapping enables many of the techniques described in this research by allowing natural language processing (NLP) on the policy to be used in conjunction with information flow analysis on the code. The mapping described in this work is an initial proof of concept that is scalable so that a more comprehensive mapping can be produced using the same methodology.

**Mobile Privacy Ontology** A common phenomena in natural language description is generalization, in which a more general phrase can be used to imply a number of sub-concepts of the phrase. For example, the phrase "technical information" may imply a wide range of technical data, while the phrase "device identifier" is more specific, but its concept is still covered by phrase "technical information". Since phrase generalization is often used to describe information collected, it is important to be able to distinguish these relationships between phrases in order to identify cases where a concept is represented in another phrase. To handle this, we created an ontology of mobile privacy-related phrases to be used as a cross reference during the identification of methods not represented in privacy policies (Section 4.2). This improves misalignment detection by providing more information about the type of misalignment (e.g., weakly or strongly disparate).

**Static Misalignment Detection** By leveraging existing program analysis tools such as FlowDroid [9], the above techniques can be utilized for identifying relevant data flows. For example, if the framework identifies that the privacy policy claims that location information is not collected, information flow analysis can detect if information produced by a GPS coordinate method is sent to an outbound data flow (e.g., a network sink). Such a situation would constitute a misalignment between code and policy. Section 5.3 describes the use of static program analysis to detect misalignments involving platform-level information sources. Misalignments resulting from user input are addressed in a related work described in Section 6.3.1.

**Dynamic Monitoring** Android has many dynamic features (e.g., callbacks, user-triggered events, forms, etc) and in some cases, static analysis is not sufficient. If an app uses native code to elicit private information directly from the user, prior knowledge of the app's implementation is necessary to know what information could potentially be private. By dynamically monitoring the input to outbound method sinks, private information leaks can be detected. For example, an app may have an input field that takes a user's full name as input and relays it to an outside source. Without knowing in advance that the variable is meant to hold the user's name, it may be impossible to catch such a data leak. Static analysis cannot anticipate the exact content of the data flows. Furthermore,

monitoring of network sinks at runtime can provide direct notification to the user exactly when a policy misalignment occurs in real-time. Section 6.1 describes my implementation of a runtime monitoring system for the detection of potential misalignments.

# CHAPTER 4: BRIDGING THE SEMANTIC GAP

A primary impact of this work is the facilitation of a means for detecting misalignments between natural language privacy policies and their corresponding app code. As an initial proof of concept, we have created an approach that identifies privacy promises in mobile app privacy policies and checks these against code using information flow analysis to raise potential policy misalignments [78]. As part of checking for data over-collection misalignments within the app, the approach uses information flow analysis to detect if the data is sent outside the app. The approach was evaluated through a case study of 477 top apps from Google Play to determine the accuracy of the framework. The approach is described as follows.

## 4.1 Mapping Approach and Annotations

In our approach, we created a mapping between API method signatures in the Android SDK and meanings shared between API documents and privacy policies. The shared meanings are described in an ontology that provides support for comparing two technical terms: we say that one term subsumes a second term, when either the first term is more general than the second term, called a *hypernym*, or when the second term is part of the first term, called a *meronym*. For example, "mobile device model" and "sensors" are parts of a "mobile device," whereas "mobile device model" is also a kind of "mobile device information." In addition, we define two terms as *synonyms* when the meaning is equivalent for our purposes (e.g., when "IP address" is a synonym for "Internet protocol address"). Because privacy policies tend describe technical information using more generic concepts, the ontology allows us to map from low-level technical terms to high-level technical categories, and vice versa. Once the ontology is constructed, we can use tools to automatically infer which terms should appear in privacy policies based on the API method calls in a mobile application.

I now describe the annotation process required for the construction of the mobile privacy ontology and API-phrase mapping. In each step, we employed research methods aimed at improving

construct and internal validity and reliability, which are discussed in Section 4.4.

### 4.1.1 Extracting the API Terminology

In our approach, a subject matter expert, who would typically be the maintainer of the Application Programmer Interface (API) documentation, annotates an API document. The annotations map key phrases in the API documents to low-level technical terminology in an API lexicon (e.g., "scroll bar width" or "directional bearing" are low-level technical terms). To bootstrap our approach, we chose to annotate the entire collection of API documents in the Android SDK, which includes 2,988 API documents containing over 6,000 public method signatures (here, the term "public" refers to the Java access modifier). Each API document consists of one or more method signatures, which each consist of the method name, input parameters, the return type, and a natural language description of the method's behavior.

The annotation procedure involves three steps: (a) we extract the method names, input parameters and natural language method descriptions from the API documentation to populate a series of crowd worker tasks; (b) for each crowd worker task, two investigators separately annotate the extracted fields by identifying which phrases correspond to a kind of privacy-related platform information; and (c) the resulting annotations are compiled into a mapping from the fully qualified method name, including API package name, onto each annotated phrase (i.e., each method name can map to one or more platform information phrases). We only compiled mappings where the two investigators both agreed that the phrase was a kind of privacy-related, platform information.

In the first step, the signatures were automatically extracted from the API documents, which were themselves expressed in HTML generated using the Javadoc toolset. The signatures were then segmented into sets of 20 signatures or less, and each set was presented in a separate crowd worker task. Applying the segmentation to the 2,988 API documents yields 310 crowd worker tasks.

The crowd worker task employs a web-based coding toolset developed by Breaux and Schaub [16] for annotating text documents using coding theory, a qualitative research method for extracting

data from text documents [70]. In coding theory, the annotators use a coding frame to decide when to code or not to code a specific item. In our study to annotate the API documents, our coding frame consisted of a single information code defined as information "related to personal privacy and accessed through the platform API." In the second step, two investigators used this web-based toolset to code the 310 crowd worker tasks, consuming 6.5 and 6.6 hours for each investigator to yield 195 and 196 annotations, respectively.

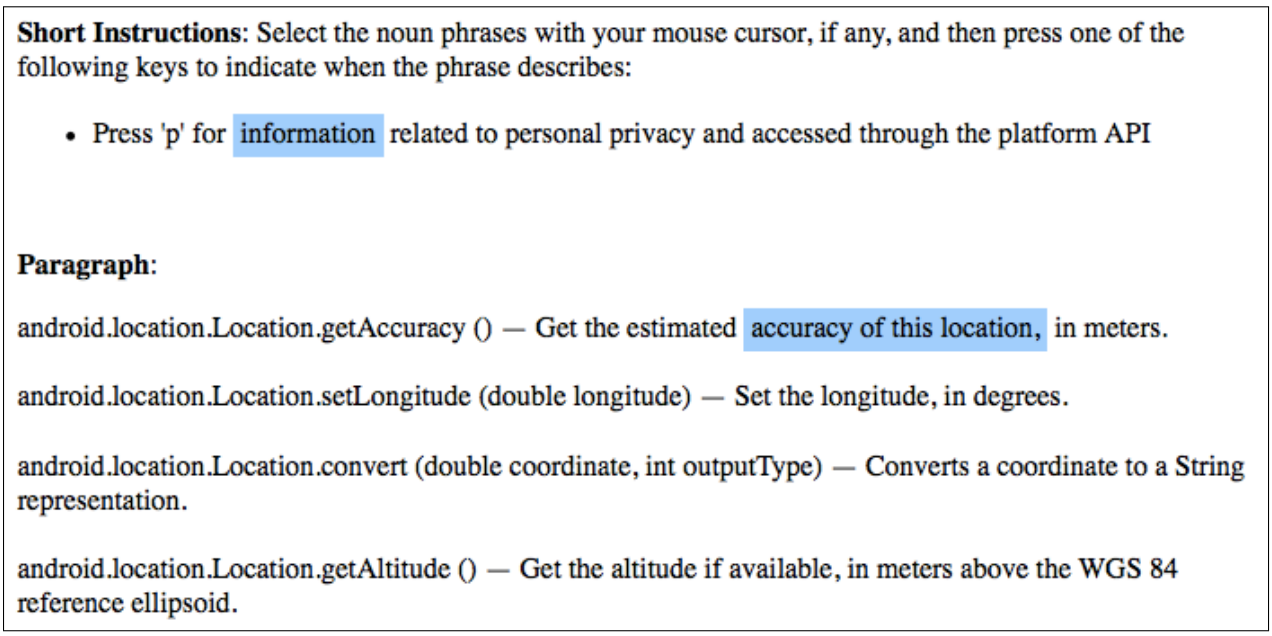



**Figure 4.1**: API Annotation Crowd Worker Interface

Figure 4.1 shows an excerpt from the crowd worker task, where a worker has annotated phrases in the `Location` package of the Android API. The toolset has been validated in a prior case study to extract privacy requirements from privacy policies [16]. The toolset also includes analytics for extracting overlapping annotations where $n$ or more workers agreed that the phrase should be annotated.

From the two investigators' combined annotations, we produced 219 unique annotations with duplicate annotations removed. The total 219 annotations were next compiled into a mapping between API method signatures and annotated phrases. The phrases in the mapping were normalized by the two investigators by converting the annotated text into simple noun phrases (described further in Section 4.3). This is necessary to reduce the variety of ways that method behaviors are

described into a concise, reusable API lexicon. The resulting lexicon contains 162 unique phrases and 169 total mappings between phrases and API method names. A total of 154 methods were annotated based on the criteria that they produce privacy related information.

### 4.1.2 Extracting the Privacy Policy Terminology

Each app page on Google Play includes a link to the app's privacy policy if it is specified by the developer. I created a Python script[1] to download the privacy policies from these links for the top 20 free apps in each app category[2]. The script locates an arbitrary app's privacy policy by accessing its Google Play page based on the package name and locating the link to the policy if it exists. This is possible since Google Play lists apps' privacy policies in a uniform way. We filtered theses policies based on their formatting, language (we only considered policies written in English), and whether or not a "Privacy Policy" section was explicitly stated in the document and randomly selected 50 from this pool for terminology extraction.

For our approach, we determine which kinds of technical information should appear in privacy policies to describe privacy-relevant API method calls. To bootstrap our method, we developed a privacy policy lexicon in which six investigators annotated the 50 mobile app privacy policies using our crowd worker task toolset [16]. Unlike the API lexicon, wherein we used only two investigators with programming experience, we used six annotators for extracting terms from privacy policies, because privacy policy terminology includes vague and ambiguous terms that span a broader range of expertise (e.g., "taps" corresponds to user input, whereas "analytics information" includes web pages visited, links clicked, browser information, and so on.) Thus, by increasing the number of annotators, we increased our likely coverage of potentially relevant policy terms.

The crowd worker task employs the same web-based coding toolset developed by Breaux and Schaub [16]. To prepare the policies for annotation, we first removed the following content: the introduction and table of contents, "contact us", security, U.S. Safe Harbor, policy changes and

[1] https://github.com/rslavin/Android-Policy-Downloader

[2] The list of app categories is available at the Google Play website, and the top 20 apps for each category was fetched on May 19th 2015.

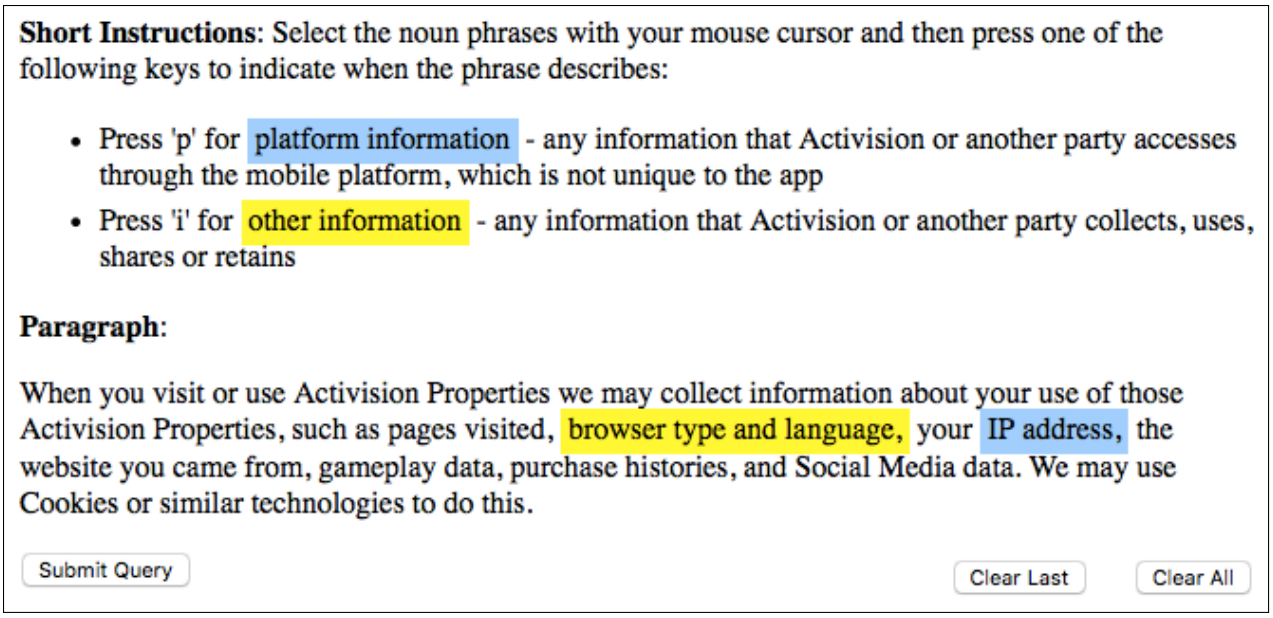



**Figure 4.2**: Policy Annotation Crowd Worker Interface

California citizen rights. This content generally appears in separate sections or paragraphs, which reduces the chance of inconsistency when removing these sections across multiple policies. While these sections do describe privacy-protecting practices, such as complying with the U.S. Safe Harbor, we have never observed descriptions of platform information in our analysis of over 100 privacy policies in our previous research [17]. Next, we manually split the remaining policy into spans of approximately 120 words (Figure 4.2). We preserve larger spans which either have an anaphoric reference back to a previous sentence (e.g. when "this information..." depends on a previous statement to understand the context of the information), or when the statement has subparts (e.g., (a), (b) etc.) that depend on the context provided by earlier sentence fragments. On average, we need 15 minutes per policy to complete the preparation.

The coding frame for the privacy policy terminology extraction consists of two codes: *platform information*, which we define as "any information that $company or another party accesses through the mobile platform, which is not unique to the app;" and *other information*, which we define as "any information that $company or another party collects, uses, shares or retains." We replace the $company variable with the name of the company whose policy is being annotated. Next, we compiled the annotations where two or more investigators agreed that the annotation was a kind

of platform information; we excluded non-platform information from this data set. We applied an entity extractor [14] to the annotations to itemize the platform information types into unique entities, which were then included in the privacy policy lexicon.

Among the 50 policies, we constructed 5,932 crowd worker tasks with an average word count of 98.6; the average words per policy was 2054.6. These tasks produced a total of 720 annotations across the 50 policies, which yielded a total of 368 unique platform information entities. The total time required to collect these annotations was 19.9 hours across six annotators.

## 4.2 Mobile Privacy Ontology

An ontology is a formal description of entities and their properties, relationships, and behaviors [32], and is described with formal languages such as OWL (based on Description Logic). In the context of phrase mapping, we use an ontology to represent a hierarchical classification of phrases. For example, in Figure 4.3, "IP Address" is a decedent of "Network Information", indicating that IP Address is a type of network information. The hierarchical nature of an ontology allows for transitive relationships that can be used for mapping API methods to phrases indirectly based on relationships between the phrases themselves.

The ontology is used to formally reason about the meaning of terminology found in the API documents and privacy policies. For an API lexicon $\hat{A}$ and a privacy policy lexicon $\hat{P}$ consisting of unique terms (or concepts), the ontology is a Description Logic (DL) knowledge base $KB$ that consists of axioms $C \sqsubseteq D$, which means concept $C$ is subsumed by concept $D$, or $C \equiv D$, which means concept $C$ is equivalent to concept $D$, for some concepts $C, D \in (A \cup P)$. Using our API lexicon, our aim is to map a method name $m$ from an API document to a concept $A \in \hat{A}$. Next, we aim to infer (in a forward direction) all policy concepts $\{P | P \in \hat{P} \wedge KB \models P \sqsubseteq A \vee KB \models P \equiv A\}$. In this respect, we can extract method names from method calls in a mobile app, then infer corresponding policy terms (among which at least one) should appear in the mobile app's privacy policy. Similarly, we can reason in the backward direction to check which policy terms mentioned in the app's policy map to which method names corresponding to method calls in the app.

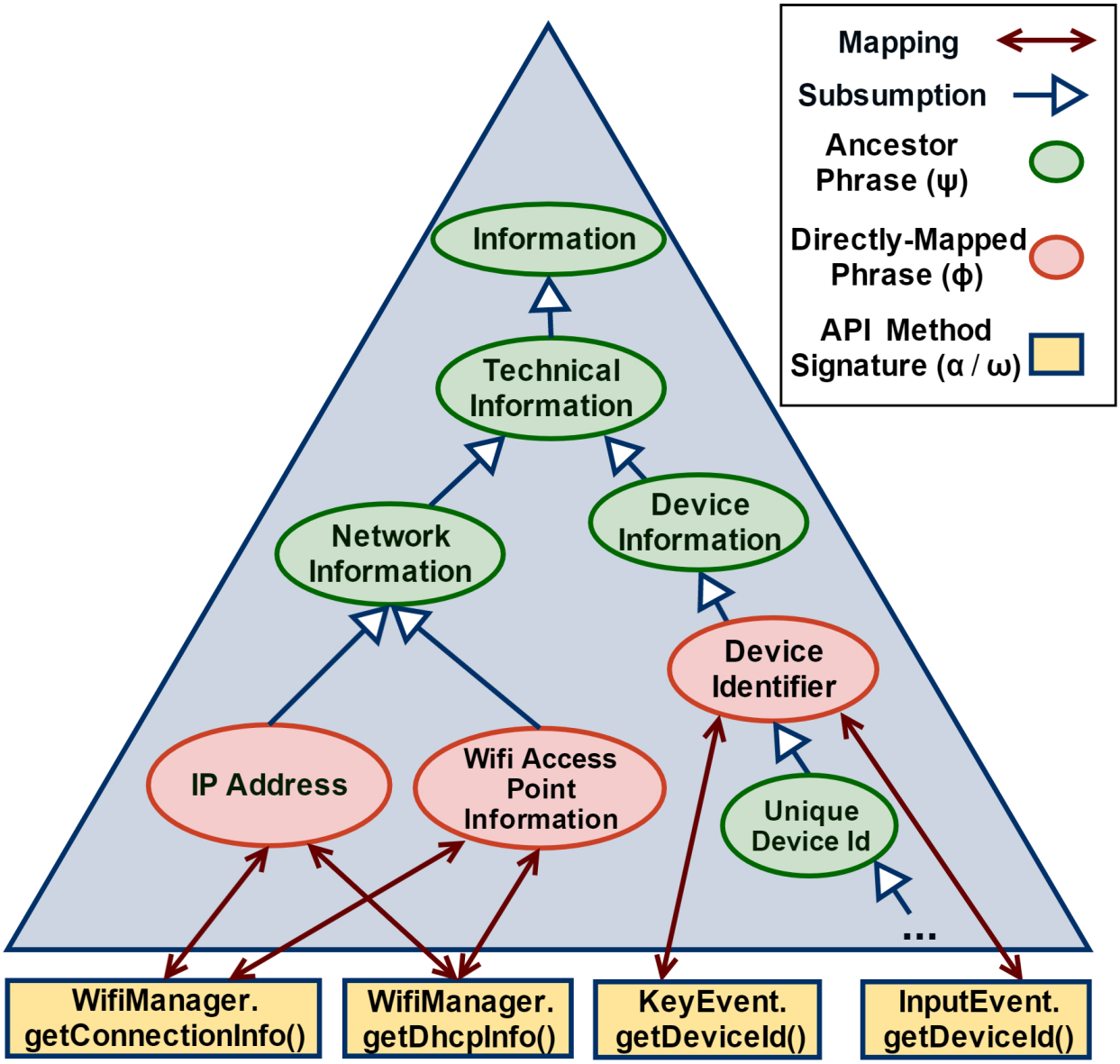



**Figure 4.3**: Abbreviated Ontology Example with Mapped API Methods

We constructed the ontology following a preexisting method for extracting ontologies from an existing lexicon [36]. First, we generated a basic ontology consisting of one concept for each term in the privacy policy lexicon; each concept was subsumed by the $\top$ concept, and no other relationships among concepts existed. Second, for two copies of the basic ontology $KB_1$ and $KB_2$, two investigators separately performed pairwise comparisons among term pairs $C$, $D$ in each ontology, respectively: if two terms were near synonyms, the first investigator created an equivalence relation $KB_1 \models C \equiv D$; else, if one term subsumed the other term, the first investigator created a subsumption relationship $KB_1 \models C \sqsubseteq D$. Due to the number of pairwise comparisons, it's not unreasonable to expect that a single investigator would produce an incomplete ontology, or an ontology that is inconsistent with another investigator's ontology. To check for completeness and consistency between two investigators, we compared all relationship pairs between $KB_1$ and $KB_2$, including cases were a relationship did not exist in one of the ontologies. Both investigators met to reconcile any differences, recognizing that classification differences can persist forward into our analysis of mobile app misalignments.

For two investigators, the resulting ontologies $KB_1$ and $KB_2$ consisted of 431 and 407 axioms, respectively. The first comparison yielded 321 differences and was evaluated using Cohen's Kappa to measure the degree of agreement above chance alone [20], which was 0.233. After the reconciliation process, the investigators were left with 12 differences and a Cohen's Kappa of 0.979.

## 4.3 Constructing the Mapping

With the ontology constructed from the privacy policy lexicon, individual API methods could then be mapped to one or more terms in the ontology based on their annotations from the API lexicon as well as their return types. Figure 4.4 shows how intermediate noun phrases were created as a canonical representation of the method's description and then mapped directly to terms in the ontology based on their relationships. As exemplified in the figure, WifiManager.getConnectionInfo() is described in its documentation as returning "dynamic information about the current Wi-Fi connection". It is difficult to infer the extent of information that is returned from this description

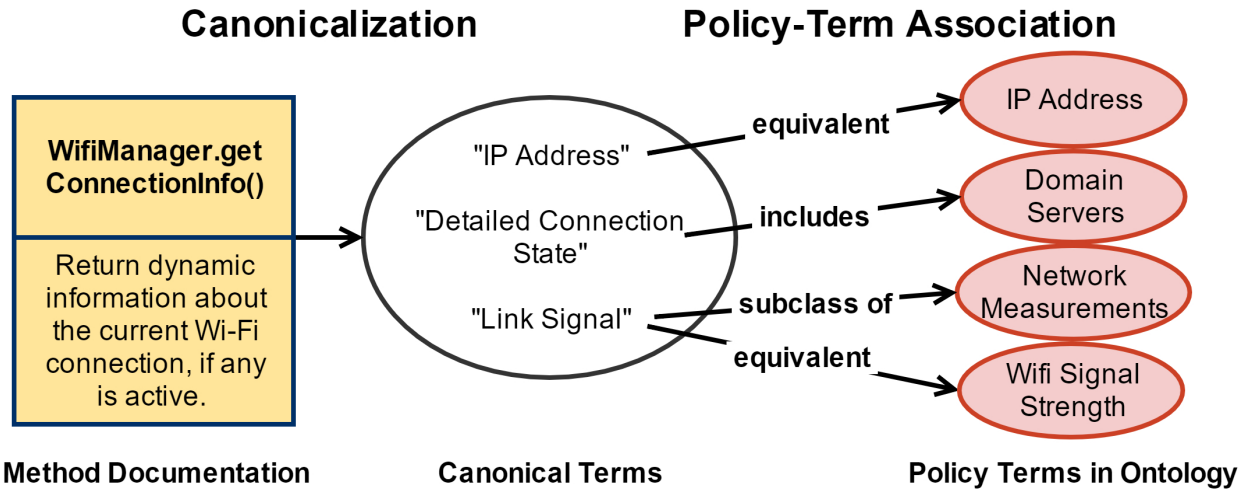



**Figure 4.4**: Mapping Process

alone.

This canonicalization process made explicit the domain knowledge about the methods (i.e., canonical terms) and the natural language used to describe the method in privacy policies (i.e., terms in the ontology). As exemplified in the figure, the documentation describes "dynamic information about the current Wi-Fi connection" as the data it produces. In cases such as this, where the description did not explicitly describe the information returned, we analyzed the object returned by the method. Here, the object (of type *WifiInfo*) provided multiple public fields and methods from which we were able to assign the canonical terms in the figure (as seen in the white circle). From there, the canonical terms were associated with related terms in the ontology based on their relationships. This effectively produces a mapping between each of the API methods and one or more terms in the ontology (assuming the method is privacy-relate) and vice versa. We refer to this many-to-many mapping relation, of which each element is a pair, $(policy\ term, API\ method)$, as $Mappings$ in the following sections.

## 4.4 Threats to Validity

**Construction Validity** is the extent to which we have measured what we think we are measuring [90]. In this study, we collected annotated phrases privacy policy terms and API terms that concern platform information. To address this threat, we provided annotators with the same pro-

cess and instructions for identifying relevant phrases, and we selected only those policy terms where two or more workers agreed to the annotation. When constructing the ontology, the investigators employed heuristics [36] to justify the classification and agreement was measured using a chance-corrected statistic at each iteration to identify disagreements for reconciliation. With respect to measuring misalignments, we introduced two kinds of misalignments: strong misalignments wherein the known policy terms were not found in the associated policies, and weak misalignments in which the app's policy includes vague terms that are generally associated with the information accessed by the app. These misalignment types are detailed in Section 5.1.

**Internal Validity** refers to whether the causal inferences we derive from the dataset are valid [90]. When mapping policy terms to API terms, the investigators made numerous inferences. To reduce this threat, the investigators decomposed the mapping process into multiple, independent tasks: annotating the policy excerpts to identify personal information accessed through the platform, identifying ontological relationships between policy terms, identifying canonical names for platform API, and classifying those names by the policy terms in the ontology. For each of these step, the work was limited to small tasks in which each datum was individually reviewed by multiple investigators. The entire process consumed more than 33 hours, however, it reduced the likelihood that inferences would be missed, for example, by identifying relevant API method descriptions and aligning them directly with policies in an otherwise ad hoc fashion.

**External Validity** refers to the extent to which our results generalize to other policies and domains. In this study, we manually examined 50 policies to construct the ontology and policy-API mapping. To reduce this threat, we selected the most frequently used apps from different app categories to enhance the representativeness of our data set. However, we only sampled from Android apps, thus it is possible that our results do not extend to iOS-based apps, or further to web-based applications. However, some of our apps only had a combined privacy policy for mobile and web-based applications, thus, we are confident that our method could be extended to web-based applications with further research.

# CHAPTER 5: STATIC MISALIGNMENT DETECTION

To detect potential privacy policy misalignments in an arbitrary app, we first identify API method invocations that produce data covered by a known policy term from the privacy policy lexicon. Next, we use information flow analysis to check whether that data flows to a remote server via a subsequent network API method invocation. Data collected by a method is considered a potential privacy policy misalignment if the method is not represented in the app's privacy policy through $Mappings$. An overview of the full process is in Figure 5.1.

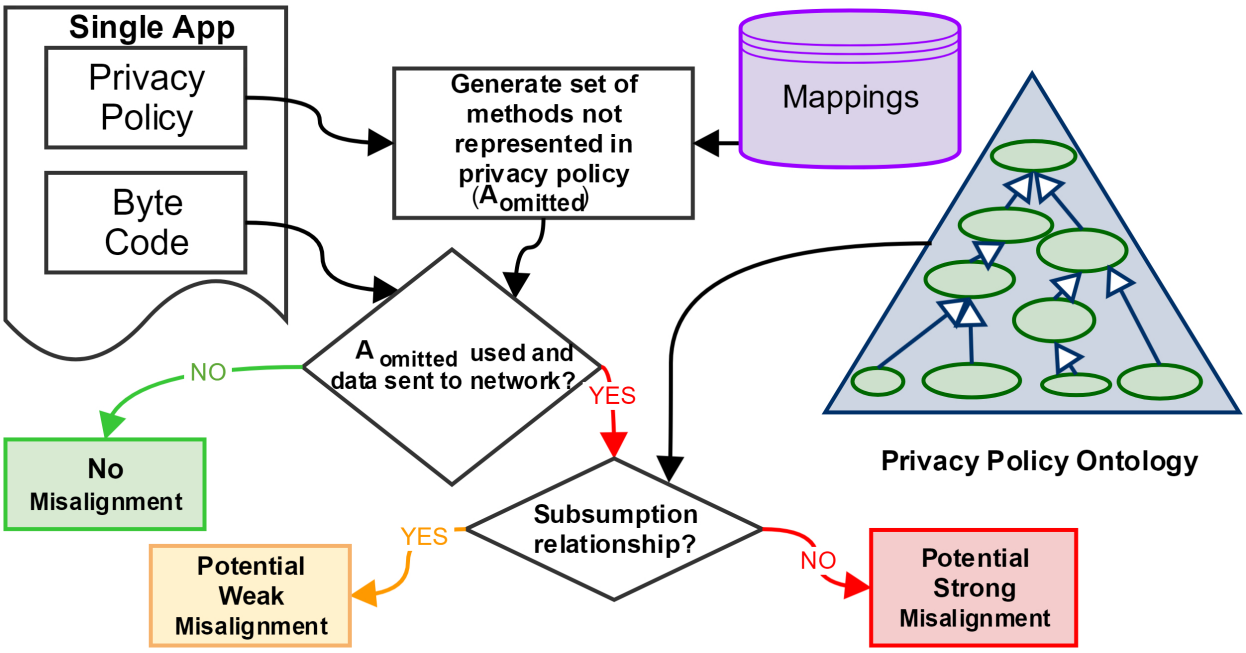



**Figure 5.1**: Misalignment Detection

## 5.1 Weak and Strong Misalignments

As discussed in Section 2.3, privacy policies serve to inform users about how their personal information is collected and used. These policies cover a wide range of practices, including in-store, client-side, and server-side practices, and they may describe all of a company's practices, or be limited to only those practices of a single product or service. For this work, we are only con-

cerned about client-side practices affecting mobile applications. In addition, privacy policies are not complete: they generally describe a subset of the company's practices. Therefore in our approach, we only detect *errors of omission*, in which the app collects a kind of information that is not described in the policy. Errors of omission are *potential* policy misalignments, because the collection may be unintended by the app developer. Moreover, because privacy includes notifying users about how their information is collected and used, errors of omission represent potential privacy misalignments. We detect two kinds of misalignments resulting from errors of omission: *strong misalignments* that occur when the policy does not describe an app's data collection practice, and *weak misalignments* that occur when the policy describes the data practice using vague terminology.

## 5.2 Detection of Suspicious Method Invocations

To begin the process, we preprocess the app's privacy policy to generate a set of method-related phrases that represent what the privacy policy states it collects. First, all words in each policy paragraph are converted to their base dictionary form (i.e., lemmatization) [50]. If the lemmatized paragraph contains a collection verb[1], the paragraph is kept for further analysis. Next, the intersection of the lemmatized words in the paragraphs and the phrases in the privacy policy ontology is calculated to produce the set of existing policy phrases $\Phi$. We use $\Phi$ and the many-to-many relation $Mappings$ to generate a list of method names, $A_{represented}$,

$$A_{represented} = \{\alpha \mid \phi \in \Phi \wedge \alpha \in map(\phi)\} \tag{5.1}$$

where $A_{represented}$ denotes the set of methods that are directly represented within the privacy policy based on the mapping, and $map(\phi)$ produces all the methods to which $\phi$ is mapped.

The set of omitted API method names can then be defined as the following, where $A_{mapped}$ represents the set of API method names that appear in $Mappings$ (i.e., all methods in the Android

[1]The collection verbs used in this study are the result of the manual annotation by two investigators of 25 random privacy polices from the set of 50 privacy policies described in Section 4.1.2.

API for which at least one mapping to a policy phrase exists).

$$A_{omitted} = A_{mapped} \setminus A_{represented} \tag{5.2}$$

The app's source code can now be scanned for any instances of $\alpha \in A_{omitted}$. Such instances would then be flagged as a suspicious invocation, $\omega$. To determine information about the invocation, the offending API method name, $\omega$, is cross-referenced with $A_{mapped}$ to determine all phrases to which it is mapped, $\Phi_\omega$. The ontology is then used to determine the set of terminological ancestors, $\Psi_\omega$, of all $\phi \in \Phi_\omega$ (Equation 5.3), which are those phrases that include in their interpretation the data accessed by the API method invocation. The phrases $\Psi_\omega$ semantically subsume (i.e., are more generally descriptive of) at least one member of $\Phi_\omega$ according to the ontology. The relationships between $\alpha$, $\omega$, $\Phi$, and $\Psi$ can be seen in Figure 4.3, where an $\alpha$ becomes an $\omega$ if it is a suspicious invocation. Due to the nature of an ontology, the set of nodes of highest level $HNodes = \{\text{“}information\text{”}, \text{“}software\text{”}, \text{“}technology\text{”}\}$ is generally descriptive of all of their subterms and thus not useful. Therefore, we use Equation 5.4 to describe the ontology.

$$\Psi_\omega = \{\psi \mid \phi \in \Phi_\omega \wedge \phi \sqsubseteq \psi\} \tag{5.3}$$

$$O' = O \setminus HNodes \tag{5.4}$$

The members of $\Psi_\omega$ represent terminological ancestors of $\omega$ that relate to $\omega$ through a subsumption relationship. These ancestors may include other interpretations that are not associated with $\omega$ (e.g., “technical information” includes “location information” and “usage information”, which are distinct and different kinds of information). Thus, $\psi \in \Psi_\omega$ is checked for matches in the privacy policy to determine if $\omega$ is described in the policy through a more general term. If a match is found, then a resulting misalignment is flagged as a weak misalignment (i.e., the policy contains a phrase transitively mapped to an API method name). Otherwise, the misalignment is flagged as a strong misalignment, which means there is no relationship between any phrase in the policy, the ontology,

and the corresponding API method invocation. While a strong misalignment is obviously harmful to the protection of the users' privacy due to the lack of notice, a weak misalignment is still potentially harmful, because it indicates the lack in sufficient detail about the data practice and it can be used as a guide for improving the clarity of the privacy policy [64].

## 5.3 Information Flow Analysis

As explained above, an API method invocation becomes a misalignment only if the information it fetches is sent to remote servers. Therefore, we need to further check the destination of the fetched data, and existing information flow analysis tools provide the technique needed for this goal. We also require a list of *sink* methods that send information to remote servers.

It should be noted that our framework works with any information analysis tool and sink methods list for network data transfer. In our implementation, we leveraged FlowDroid [9], the state-of-art technique for Android information flow analysis, to track the information flow within Android byte code. We also used the list of sink methods for network data transfer described by SUSI [63], a machine-learning tool for classifying Android sources and sinks. FlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications, which generates information flow paths with very high recall and precision. To achieve this goal, the developers built an analysis that is context-, flow-, field- and object-sensitive, created a complete model of Android's app lifecycle.

In our framework we use all suspicious API invocations as sources, and network sink methods collected by SUSI as sinks, and report a misalignment if there exists an information flow from a suspicious API invocation to network sinks. According to whether the suspicious method invocation is represented in the privacy and its relationship to the method based on the ontology, weak and strong misalignments are reported using the method described in Section 5.2.

# CHAPTER 6: RUNTIME MONITORING

The static analysis technique described in Chapter 5 relies on known data sources (e.g., Android API methods) for identifying policy misalignments. However, private information is not limited to such sources and such sources can not always be anticipated (i.e., mappings may not exist). Furthermore, static analysis requires powerful machines and can take a long time to complete (relative to the responsiveness of mobile apps), which is not practical for end users and auditors. Much of the time required for the analysis has to do with analyzing all possible data flows, many of which may never be executed in real-world use cases. If detection were implemented to monitor the code execution at runtime, users could accurately and easily detect misalignments for specific use cases. Therefore, to improve the robustness of the misalignment framework, I have devised a technique for efficient runtime detection.

Runtime detection requires either direct instrumentation of the app code itself or monitoring of the app's memory space to identify and report relevant behavior. My approach accomplishes the former with minimal overhead by leveraging the Xposed framework to alter potential network sinks in native application code so they log the data sent through them. Using this method, misalignment detection techniques as described in Section 4 can be applied to arbitrary applications at runtime.

## 6.1 Xposed

The Xposed framework[1] is a tool for the modification of compiled Android apps so they can behave differently at runtime. Xposed takes advantage of the *Zygote* Android daemon, from which all Android apps are forked. By overwriting the process with its own, the framework (and, by extension, its modules) is able to insert hooks into the bytecode of the app allowing the module to perform custom code before, during, and after hooked method calls.

Xposed is installed as a normal Android app and runs as a service on the device and thus is able to replace Zygote at boot time. Modules are also written and installed as Android apps

[1] http://repo.xposed.com

using Java. Due to the nature of the framework, the Xposed app itself requires root access to the device it is installed upon. While this can be considered a security risk, the preferred methods for "rooting" Android devices places an application-level layer between apps requesting root access and the actual `su` binary which restricts all access based on the user's specifications. The access-managing app, SuperSu, is available on Google's own Play store and, as of the time of this writing, has over 50,000,000 downloads[2].

The framework was created as a means to modify the behavior of Android apps at runtime without having to rewrite (and thus sign and recompile) the app code. The framework itself is open source[3] and has been actively maintained since its creation in 2013. It is currently available as a stable release for Android 6 with an early testing release for Android 7 (the newest Android release at this time). Over 1,000 publicly-available Xposed modules are currently published at the official repository[4].

## 6.2 Parameter Reporter

In order to verify the information being passed to network sinks, I have implemented the Parameter Reporter[5] module as a runtime tool to "hijack" the Android apps at runtime and report the input parameters to the sinks involved in the suspected leaks. This practice allows for both the detection of calls to sinks and the detection of the specific data sent to those sinks. For example, the `org.apache.http.client.HttpGet()` sink is commonly used to transmit information in the form of GET variables to a remote server. In its simplest use, Parameter Reporter can detect invocations to the `HttpGet()` method and report any information, such as GET variables, that are passed to it in real time. In this way, Parameter Reporter provides a foundation for many use cases.

The method for instrumenting the code for a specific app and specific sinks is depicted in Figure 6.1. First, the module must be tailored for the sinks it is detecting for a spe-

[2]https://play.google.com/store/apps/details?id=eu.chainfire.supersu
[3]https://github.com/rovo89/Xposed
[4]http://repo.xposed.info/module-overview
[5]https://github.com/rslavin/Android-Parameter-Reporter

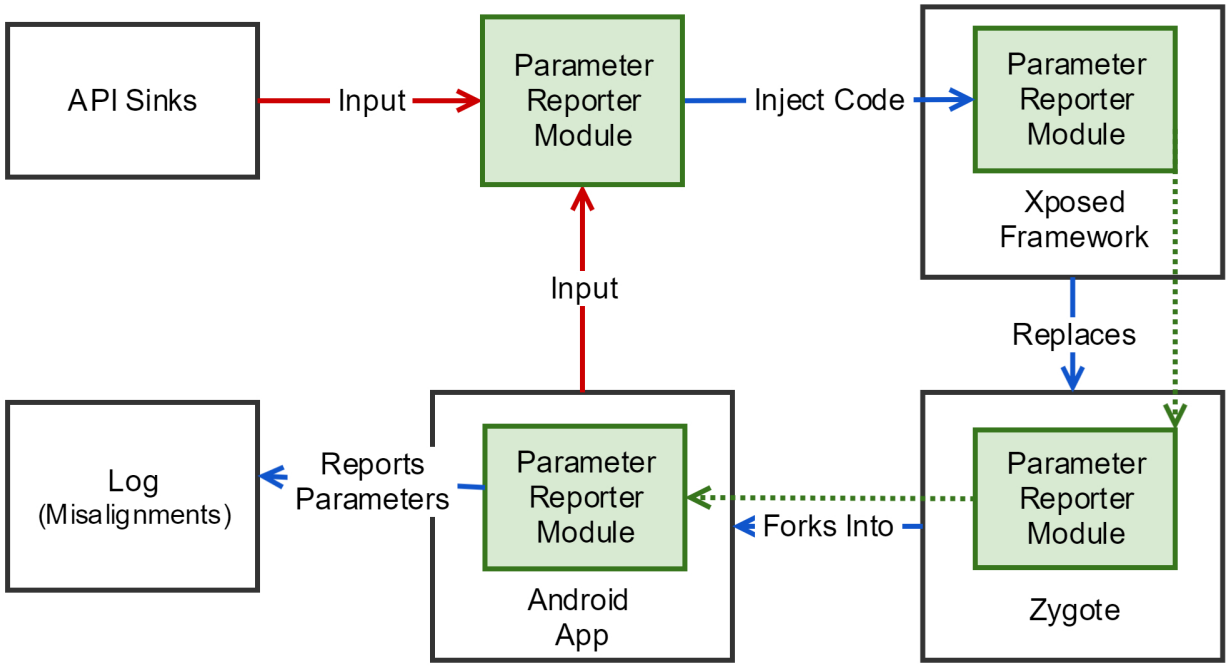



**Figure 6.1**: Xposed Parameter Reporter Module

cific application. As inputs, Parameter Reporter takes the signature of the API sink (e.g., `org.apache.http.client.HttpGet()`), the indices of the parameters for which the module will report their values, and optionally a list of the applications in which to scan. The module's implementation uses these inputs to inject detection code into the target apps. Once the module has been tailored for the sinks to detect, it can be loaded on the device using the Xposed Framework.

The integration of the module with the app at run time can be seen in the figure starting from the top right box. The module is loaded by the Xposed framework at boot time and thus is transferred into Zygote which is replaced by Xposed along with the custom Parameter Reporter code. When the app is started, the process (i.e., the app itself) is forked from Zygote and the module persists within the app's bytecode along with the hooks included for detection of the sinks.

The module, which now resides within the app on the device, includes instructions to report all input parameters passed to the sinks to a log file on the device. In this way, Parameter Reporter does not require external tooling or hardware allowing the device to be used in an everyday manner.

When an app is loaded, the module checks to see if it matches an input app given to Parameter Reporter. If so, it instruments all specified sinks called in the app with the code in Listing 6.1.

The method, `reportCall()`, is called immediately after the sink in question is called using the `afterHookedMethod()` Xposed method. The instrumentation executes after the method in question in order to guarantee that it did in fact complete. As seen in the listing, the instrumentation prints out diagnostic information on the call including the parameters in question in $O(n)$ time based on the number of parameters. As discussed in Section 9.2, the instrumentation results in an overhead of approximately 0.6ms per sink call.

Listing 6.1: Sink Instrumentation

```
1  private static void reportCall(XC_MethodHook.MethodHookParam param, String
       methodName, String packageName, List<Integer> paramIndices) {
2       // print package and method name
3         XposedBridge.log(">>> FOUND API CALL <<<\n\tApp:\t\t" + packageName +
              "\n\tMethod called:\t" + methodName);
4         if (param.args != null && param.args.length > 0){
5            for (int i = 0; i < param.args.length; i++){
6            // print parameter
7               if (paramIndices.indexOf(i) >= 0)
8                  XposedBridge.log("\tParameter[" + i + "]:\t" + (param.args[i] ==
                       null ? "null" : param.args[i].toString()));
9          }
10      }
11        XposedBridge.log("\n");
12 }
```

## 6.3 User Input Detection Validation

As described in Chapter 5, a major contribution of this work is a method for detecting policy misalignments using static analysis techniques. These techniques rely on the API-based mapping (Section 4.1) to bridge the gap between policy and code. In a related work [86], we expanded upon this methodology to include user input for native code (i.e., non-API collection). Here, I outline our methodology for user input-based misalignment detection and describe Parameter Reporter's usage as a validation tool.

### 6.3.1 User Input Detection

The approach associates each graphical user interface (GUI) input view in an Android app to terminology in privacy policies and then performs static information flow analysis to detect information flows in the app's code which misalign with relevant policy statements. This is similar to the approach in Section 5 except the inputs to the data flows are not bound to the Android API. This is due to two issues with dealing with GUI-based input. **1:** The types of information collected through GUI forms are vague and unbounded since developers can design novel GUIs that take as input potentially any kind of information or data type. **2:** GUIs can be implemented using varying techniques which cannot be anticipated. To address these issues, we used various phrase similarity measurements to map GUI labels together with context to known privacy phrases in a similar ontology as described in Section 4.2. To address the issue of varying implementation techniques, we developed a GUI string analysis technique to estimate the structure of programmatically-generated UIs and collect all UI labels in the context of a given input view. Our analysis is based on GATOR [68], an existing GUI analysis framework.

Listing 6.2: GUI XML Code Snippet

```
1  <LinearLayout android:gravity="center_horizontal" ...>
2    <cc.pacer.androidapp.ui.common.fonts.TypefaceTextView ...
3    android:id="@id/title" ... />
4    <cc.pacer.androidapp.ui.common.fonts.TypefacedEditText ...
         android:id="@id/et_content" ... />
5    <LinearLayout ...>
6      <Button ...
7        android:id="@id/btnLeft" ...
8        android:text="@string/btn_cancel" ... />
9      <Button ...
10       android:id="@id/btnRight" ...
11       android:text="@string/yes" ... />
12   </LinearLayout>
13 </LinearLayout>
```

Unlike API methods which have explicit purposes, the purpose of user input GUIs are implicit and can be understood only from the context of the GUI. Code constructs, such as view IDs sometimes provide relevant information, but are not a reliable source of purpose. For example, in Listing 6.2, the view id of the input box is "et_content", which does not have an obvious meaning, thus making it difficult to accurately map to a policy phrase. Furthermore, since the concepts in the user input are unbound, it is not possible to exhaust all privacy-related phrases and put them in an ontology. Therefore, the misalignment detection approach in Chapter 5 is not applicable.

To address this issue, we introduced a novel misalignment detection strategy which takes advantage of the context GUI labels. The idea is that, similarly to ontologies, the GUI hierarchies convey information about hypernymy relationships. For example, an activity with title "Transaction Information" may contain multiple user input boxes about transaction time, source account, etc, which all are sub-concepts of transaction information. Therefore, when mapping user input views to ontology phrases, we use not only the id or label of the view itself, but also its ancestor ids and labels in the view hierarchy.

The mapping approach is illustrated in Figure 6.2. First, we collect the labels and ids of all the ancestor views a given input view (light blue views). Labels and ids of sibling views are then collected immediately before any collected ancestor views (dark blue views) since such sibling views often hold text labels of the input views. If an input view's id or label cannot be directly mapped to any ontology nodes, we further map its ancestor labels to the ontology. Information flow analysis using FlowDroid and GATOR can then be applied to the resulting mapped GUI labels and ids as sources to relevant sinks based on what the privacy policy allows as described in Section 5.3.

To validate the approach, we focused on two app domains in the Google Play Store: Health and Finance. These two categories are are relevant to the problem of policy misalignment because they access sensitive personal information (e.g., bank account information, diet, etc). Furthermore, these two domains are potentially regulated by the Gramm-Leach-Bliley Act (GLBA) [6], Right to Financial Privacy Act (RFPA) [34], Health Insurance Portability and Accountability Act (HIPAA) [5], and Payment Card Industry Data Security Standard (PCI DSS) [22]. In our exper-

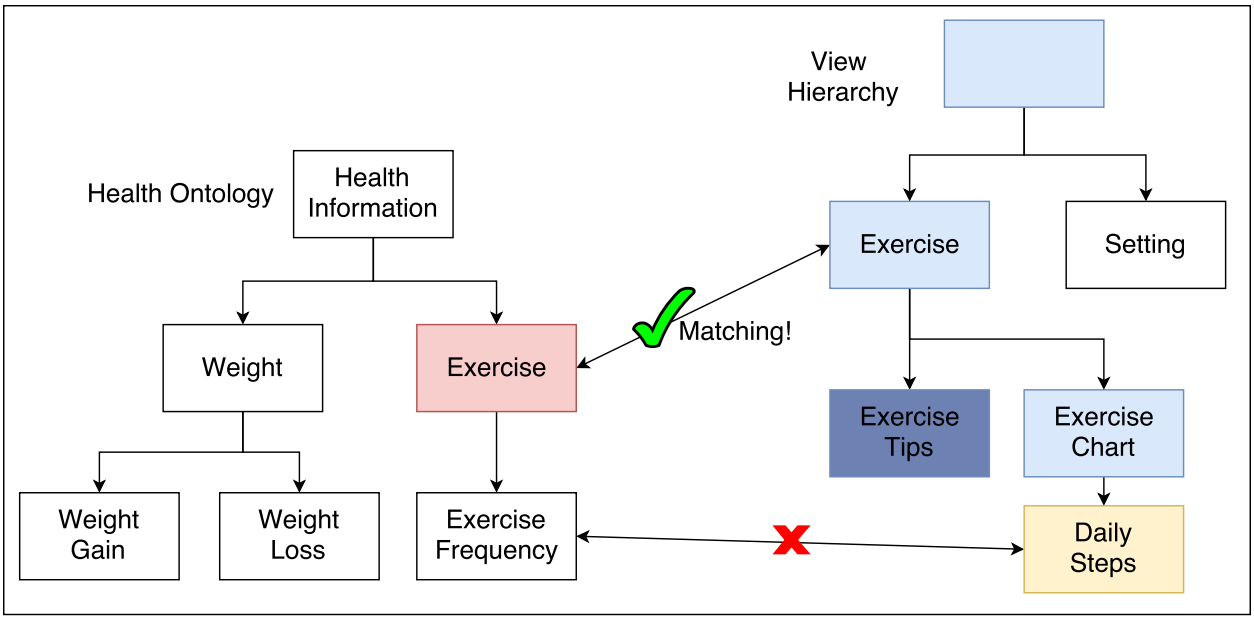



**Figure 6.2**: Hierarchical Mapping

iment, we collected 100 of the most popular apps and their privacy policies (50 apps for each category), and we used 20% of the apps (10 apps from each category) as a training set. Using the privacy policies from the training apps, we constructed an ontology for each of the two domains. Then, we applied our approach to the remaining 80 apps, and detected 30 misalignments, which we manually confirmed by recording the runtime network requests with the Parameter Reporter module.

### 6.3.2 Validation

Ultimately, the above approach allows for tracing input information from GUIs to sinks. Whereas the misalignment detection in Chapter 5 explicitly detects information flows from known sources to known sinks, GUI-based detection requires validation that the source (i.e., a GUI form) is correctly identified using the hierarchical mapping approach. There exists no ground truth to validate such flows, therefore, manual validation is necessary to measure precision. To this end, we utilized Parameter Detector to monitor the known sinks and detect data sent to them.

In order to understand the merit of the approach and the necessity for runtime validation, it is important to note that it is possible for an app to utilize data as an input without actually collect-

ing it. For example, a health-related app may require weight to locally calculate metrics such as body mass index. Merely utilizing such information would not constitute a policy misalignment. However, if the information was sent outward through the network, this indicates a situation where the data practice should be disclosed in the privacy policy. Therefore, simply identifying instance where sensitive data is entered into the app via a GUI is not sufficient for validation.

Validation was done by entering a uniquely identifiable string (e.g. "LEAKED DATA") to data sources for which a possible misalignment was suspected. Then, by manually interacting with the app to trigger the data leak, Parameter Reporter would log the identification string for every sink that took it as input. Using the method described in Section 6.3.1, 20 strong misalignments and 10 weak misalignments were detected in eight apps from an initial set of 100 apps from the health and finance categories in Google Play. In each case of a suspected leak, Parameter Detector reported data validating both the hierarchical mapping approach and the Parameter Detector module itself.

## 6.4 API Data Validation

The above approach can be extended to detect API calls similarly to as described in Chapter 5. However, unlike data collected by a GUI, data types sent to sinks may not always be easily-taggable strings. Instead, similar validation would involve the identification of sensitive object types produced by API methods. For example, the `android.net.wifi.WifiInfo` object containing sensitive connection information is returned by the `android.net.wifi.WifiManager.getConnectionInfo()` method. For validation purposes, the object type can be returned using standard methods such as `Object.getClass()`. Furthermore, if the class is known beforehand, data from its instantiation can be reported as well through standard method calls. Listing 6.3 demonstrates a simple example of how the type can be reported instead of the value itself.

Since any method with a known signature can be hooked using the Xposed framework, information-producing API methods (i.e., sources) can also be monitored allowing for input parameters to those sources to be reported in the same way as in Listing 6.1. This would require

only the method signatures of the source methods which are readily available in the Android SDK documentation.

Listing 6.3: Class Reporter

```
1 private static void reportClass(XC_MethodHook.MethodHookParam param, String
      methodName, String packageName, List<Integer> paramIndices) {
2      // log package and method name
3        XposedBridge.log(">>> FOUND API CALL <<<\n\tApp:\t\t" + packageName +
             "\n\tMethod called:\t" + methodName);
4      // log object type
5        XposedBridge.log("\tParameter[" + i + "] type:\t" + (param.args[0] == null ?
             "null" : param.args[0].getClass()));
6 }
```

This approach is useful for verifying static analysis techniques such as those described in Chapter 5. Specifically, a potential leak can be activated in real time through the app and it can be determined if the suspected data leak occurs by logging the object type and/or the input parameters to the source method.

## 6.5 Potential End User Applications

Parameter Detector has much potential for the end user. Ideally, runtime analysis would be most beneficial if it were to notify and/or block actions which may misalign with an app's privacy policy. Since Parameter Reporter can be installed similarly to how an app is installed, the resulting app would be more accessible than the static analysis tools described in Chapter 7. The tradeoff, however, would be limited robustness with regard to NLP.

By its nature, parameter detector can already notify the user of policy misalignments. To fully automate misalignment detection, the module would need to automatically retrieve the privacy policy of whichever arbitrary app it would monitor. In Section 4.1.2, the automated method for locating and downloading the policy for a specific app using a Python script was described. The same method can be applied to Parameter Reporter as the app's package names are available via its

manifest file. This would incur added overhead based on the NLP performance and downloading the policy. However, the overhead would only be experienced once per app and thus normal use of the app should not be affected.

Listing 6.4: Replacing a Hooked Method

```
1  // called before the method in question
2  protected final void beforeHookedMethod(MethodHookParam param) throws Throwable {
3     try {
4        Object result = replaceHookedMethod(param);
5       // replace method's return value
6        param.setResult(result);
7     } catch (Throwable t) {
8        param.setThrowable(t);
9     }
10 }
```

*Blocking* such actions would be a matter of modifying the instrumenting code to simply return at the beginning of the call, thus denying the sink's code body from executing. Listing 6.4 demonstrates how Xposed's `replaceHookedMethod()` method can be called prior to the actual method call in order to change the hooked method's return value and effectively prevent the original method from being called. Similar to `afterHookedMethod()`, the `beforeHookedMethod()` code is added to the hooked method, but called before it executes. Line 4 replaces the hooked method with the overridden `replaceHookedMethod()` body in Listing 6.5 which simply returns null.

Listing 6.5: MethodBlocker

```
1  public class MethodBlocker extends XC_MethodReplacement implements Constants {
2    @Override
3    protected Object replaceHookedMethod(final MethodHookParam param) throws
         Throwable {
4      return null;
5    }
6  }
```

Such a technique as returning null in place of the code body could produce undesirable side effects depending on the blocked methods since code would be prevented from executing. A less destructive solution would be to nullify the outward-bound data to prevent the leak without preventing code from executing. Listing 6.6 demonstrates how this could be accomplished by modifying `param.args`. As apparent in Line 4, the index parameter in question would need to be known in advance. This would require further annotation of API methods, which it out of the scope of this work.

Listing 6.6: Nullifying a Hooked Method's Parameters

```
1 // called before the method in question
2 protected final void beforeHookedMethod(MethodHookParam param) throws Throwable {
3   // replace parameters
4   param.args[0] = null;
5 }
```

# CHAPTER 7: POLIDROID TOOL SUITE

As efficacy for the practicality of the framework, the POLIDROID tool suite includes various tools for use by developers, auditors, policy writers, and end users. PoliDroid is freely available online [1] and features analysis tools for both byte and source code. The tool suite's web site includes tutorials that enable users to more easily access the methods used in this research. The suite also includes an Android Studio version specifically for developers (Section 7.3).

## 7.1 PVDetector

The bytecode analysis tool, PVDetector[2] [80], is a front-end for a server-side process for detecting potential misalignments between a compiled Android APK file and a natural language privacy policy.

As shown in Figure 7.1, PVDetector presents the user with a webform which takes email address, APK file, and textual privacy policy as input. Upon submitting the form, the server-side implementation uses FlowDroid [9] to perform a static analysis of the app's bytecode and detect data flows from sources of potentially private information to network sinks [63]. A natural language analysis of the textual privacy policy cross referenced with the mappings described in Section 4.1 produces a set of omitted APIs ($A_{omitted}$) as described in Section 5.2. Data flows originating from API methods within $A_{omitted}$ are flagged as potential misalignments and strong/weak identification is determined as described in Section 4.

Upon completion, the user is emailed with results and a list of suggestions for resolving any detected misalignments. The email step is necessary due to the possibility for long analysis times.

[1] http://polidroid.org
[2] http://polidroid.org/pvdetector


**Figure 7.1**: PVDetector Web Interface

## 7.2 Source Code Analyzer

The source code analysis tool [3] is a client-side application for detecting potential misalignments between Android app source code and natural language privacy policies. The application works similarly to POLIDROID's Android Studio plugin, but only for one class at a time and does not take into account weak misalignments. For a more robust analysis, the plugin is recommended.

As shown in Figure 7.2, the tool itself presents the user with a text area formatted to identify the syntax of Java source code and a plain text area to provide a corresponding privacy policy. After pasting both inputs in, the user can then click the "Analyze" button to detect potential misalignments. The detection process is done on the client using JavaScript to identify API invocations that are not represented by the privacy policy. Such potential misalignments are enumerated on the page along with a list of suggested phrases to resolve the misalignments. Like the Android Studio plugin, it does not analyze the information flows within the code. For this feature, it is

[3] http://polidroid.org./source-analyzer

recommended to use PVDetector.

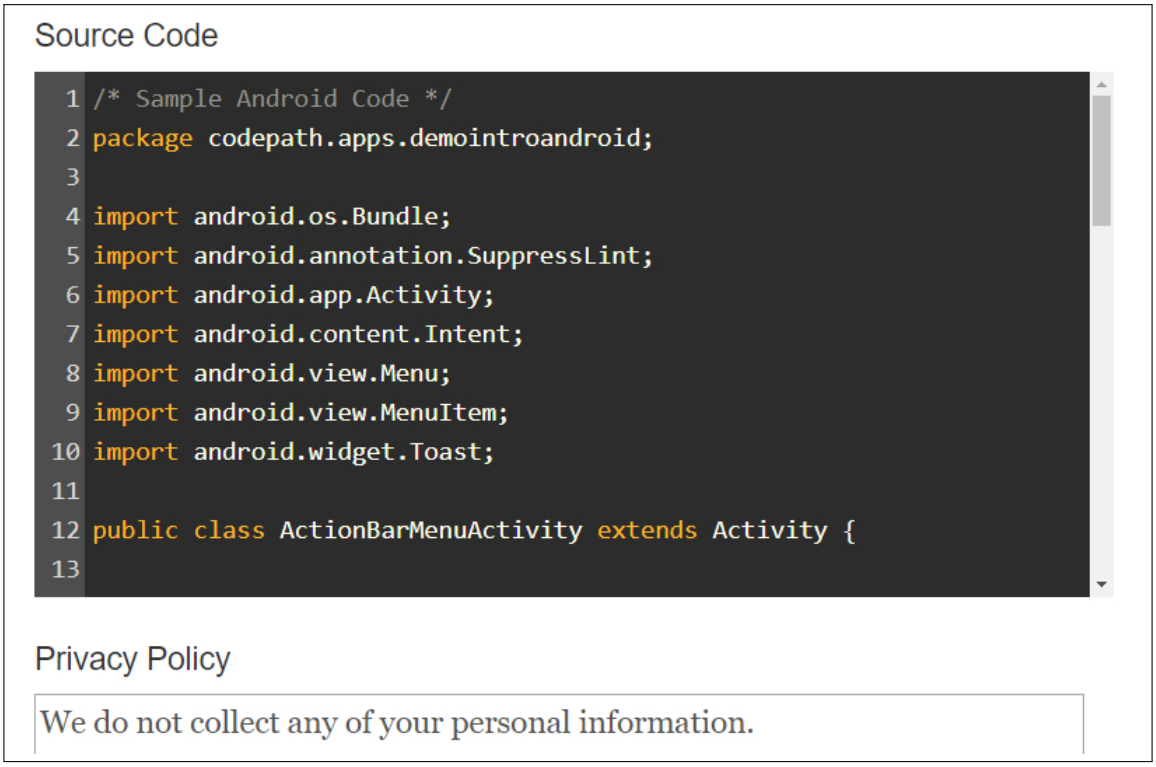



**Figure 7.2**: Source Code Analyzer Web Interface

## 7.3 Android Studio Plugin

Mobile applications frequently access personal information to meet user or business requirements. Developers are, in turn, often required to align their code with a privacy policy or to create the privacy requirements to specify the collection and use of personal information. However, it is challenging for a regular programmer to create code complying with a privacy policy. To aid app developers in such tasks, I have created *PoliDroid-AS*, an Android Studio plugin for the detection of code-policy misalignments and the generation of privacy specifications [79]. PoliDroid-AS is considered to be a portion of the POLIDROID tool suite. Due to its various features, I present it in its own section.

### 7.3.1 Components and Architecture

As seen in Figure 7.3, PoliDroid-AS is built upon a collection of tools that allow it to utilize our privacy policy misalignment detection framework within Android Studio, the predominant IDE for Android development.

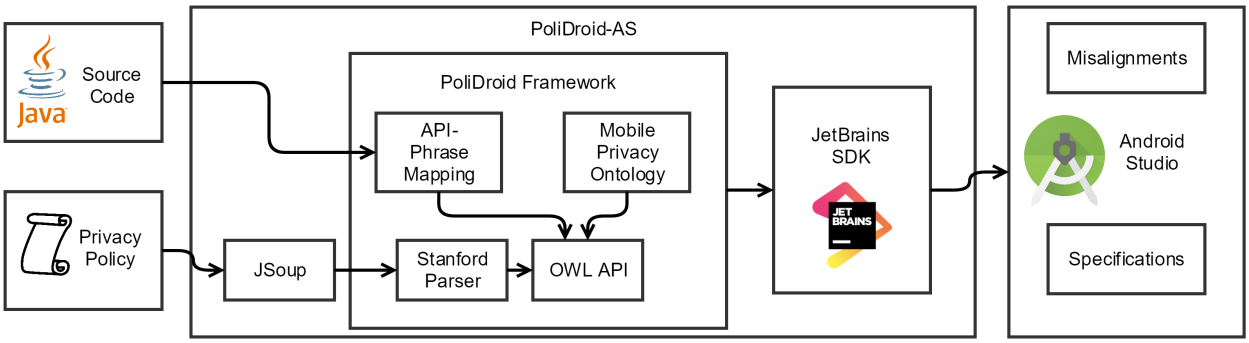



**Figure 7.3**: PoliDroid-AS Architecture

## 7.4 User Input and Supporting Artifacts

PoliDroid-AS relies on multiple inputs from the user as well as artifacts from the IDE's SDK and the framework described in Chapter 4.1. These inputs are described here.

### 7.4.1 Android Application

PoliDroid-AS operates on one application at a time and thus takes as inputs the application's privacy policy and its source code.

Since privacy policies are generally included along with other legal documents (e.g., terms of service) often displayed on the application's web page or Google Play, PoliDroid-AS is designed to take as input either a plaintext document or an HTML file including the policy. The relevant privacy policy sections are automatically extracted from the entire document (Section 7.4.3). US regulators require that privacy policies be in English, so the tool also requires an English document. Listing 7.1 is an abbreviated example of an HTML input that includes both privacy policy text and irrelevant terms of service text. Information relevant to the running example is in bold.

Listing 7.1: HTML Privacy Policy

```
<body>
        <h3>Information We Collect</h3>
        <p>
        When you use our service, we may collect
        information such as your mac address and address.
        </p>

        <h3>Terms</h3>
        <p>
        By using our app, you agree to the following
        ...
        </p>
</body>
```

PoliDroid-AS allows for real-time consistency verification, thus the app's source code can be scanned "on-the-fly" as the app is being developed. The tool expects Java source code as the relevant code used for accessing private information.

### 7.4.2 Framework Integration

.

**Table 7.1**: Example API-Phrase Mappings

| Phrase | API Method |
|---|---|
| "ip address" | getIP() |
| "unique identifier" | getIP() |
| "unique identifier" | getMac() |
| "mac address" | getMac() |

PoliDroid-AS uses the mapping and ontology described in Sections 4.1 and 4.2 respectively to detect weak and strong misalignments in source code. As an example mapping set, consider Table 7.1. The data from the mapping is taken as a comma-separated values (CSV) file as input

with policy phrases in the first column and API methods in the second column. The mobile privacy ontology is taken as a Web Ontology Language (OWL) file as exported from the Protege ontology editor [53]. The most recent versions of the mappings and ontology based in this research can be found at the PoliDroid website.

### 7.4.3 Privacy Policy Processing

Before code analysis, an app's privacy policy must be preprocessed to extract the relevant information.

If the input is in HTML form, PoliDroid-AS calls upon an existing HTML parsing Java library, Jsoup[4], to strip the HTML tags and produce a plaintext version of the input. The plugin then filters the text down to only privacy-relevant data collection paragraphs. Next, all sentences in the resulting plain text are converted to relevant parse trees using Stanford CoreNLP[5] parser. For each parse tree, the verb phrases (VPs) are extracted and if the lemmatized verbs of a VP contains a collection verb (e.g., "store", "collect", "record", etc), the VP is selected for more analysis. Sentiment analysis is also applied to remove negatively-polarized phrases (e.g., "we do *not* collect…") by tagging phrases with sentiment labels based on the Sentient Treebank described in Chapter 8. The reduced constituents are compared with the mobile privacy ontology concepts to find a match using the Web-Ontology Language (OWL) API. PoliDroid-AS then searches for phrases in the resulting paragraphs for which API methods are mapped.

### 7.4.4 JetBrains IDE SDK

The Android Studio IDE is based on the IntelliJ IDEA Java IDE by JetBrains. PoliDroid-AS was built using the JetBrains IDE Plugin SDK[6] so that it can directly interface with IntelliJ-based IDEs. PoliDroid-AS's core component is an extension of IntelliJ's inspection tool which allows PoliDroid-AS to visit all API method calls within the source code being developed within the IDE

[4]http://jsoup.org
[5]http://nlp.stanford.edu/software/corenlp.sHTML
[6]http://www.jetbrains.org/intellij/sdk/docs

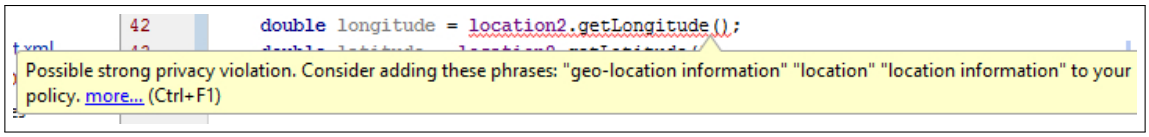



**Figure 7.4**: Misalignment Tooltip

and verify that the API method call is represented by the app's privacy policy.

## 7.5 Plugin Functionality

After providing the tool with the API-phrase mappings, ontology, and privacy policy, the PoliDroid-AS Android Studio plugin works seamlessly with Android Studio without further configuration and minimal interaction.

### 7.5.1 Set Up

Once the app developer has started Android Studio, they will be presented with a "PoliDroid" menu in the IDE's menu bar. Clicking this menu will result in four options: "Select Mappings File", "Select Ontology File", "Select Policy File", and "Generate Specifications". The first three of these selections prompts the user to load the corresponding file. The mappings file is a simple comma-separated values (CSV) file including the many-to-many mappings of APIs to privacy information collection phrases. This mapping is included with the plugin. The ontology file is an OWL formatted document specifying the subsumption, equivalence, and sibling relationships used for identifying weak and strong misalignments. Like the mappings file, a pre-defined ontology is included with the tool. We provide the option to specify custom files for the sake of flexibility.

Once the framework files have been loaded, the app's privacy policy can be specified from the same menu. PoliDroid-AS immediately strips the policy of any html markup and extracts the relevant privacy policy paragraphs (as described in Section 7.4.3) immediately upon loading the file. Next, PoliDroid-AS generates a lookup table of API methods represented in the privacy policy through the mapping ($A_{represented}$) by searching through the policy text for all mapped phrases. A set of ommitted API methods is then generated as $A_{omitted} = A_{mapped} \setminus A_{represented}$ where $A_{mapped}$

represents the set of all methods for which the mapping file has phrases mapped to. $A_{mapped}$ must be used as opposed to the set of *all* API methods because the tool cannot make assumptions on methods for which it has no data.

Continuing with our running example, the phrases "mac address" and "address" existing in the source along with the mappings from Table 7.1 would result in $A_{omitted} = \{getIP()\}$ since the other method represented in $A_{mapped}$, getMac() is represented by the phrase "mac address" in the privacy policy.

### 7.5.2 Detection and Reporting

After the generation of $A_{omitted}$, PoliDroid-AS can then scan for API method calls not represented in the policy by checking for any occurrences of methods in $A_{omitted}$.

PoliDroid-AS utilizes the IntelliJ LocalInspectionTool class to inspect all method calls within the development code. The implementation is able to match API methods by comparing a method's signature, including the method name, class, and package. If, upon inspecting a method, PoliDroid-AS detects a misalignment $\omega \in A_{omitted}$, it registers the violating method call and highlights the offending method call directly in the IDE as seen in Figure 7.4 (the annotation style can be modified to the user's preference within Android Studio's settings menu). In our example, a call to getIP() would be a member of $A_{omitted}$ (i.e., not be represented in the privacy policy) and thus be flagged as a potential misalignment.

To determine information about the misalignment, the offending API method, $\omega$, is cross-referenced with $A_{mapped}$ to determine all phrases to which it is mapped, $\Phi_\omega$. The mobile privacy ontology is then used to determine the set of hierarchical ancestors, $\Psi_\omega$, of all $\phi \in \Phi_\omega$. These phrases semantically subsume at least one member of $\Phi_\omega$ so that $\Psi_\omega = \{\psi \mid \phi \in \Phi_\omega \wedge \phi \sqsubseteq \psi\}$ according to the ontology. The members of $\Psi_\omega$ represent indirect, or transitive, representations of $\omega$ since they are not directly mapped to the $\omega$, but are related through a subsumption relationship. For this reason, $\psi \in \Psi_\omega$ can be checked for matches in the privacy policy to determine if the misalignment is considered weak or strong. If any phrase in $\Psi_\omega$ is found in the policy, the mis-

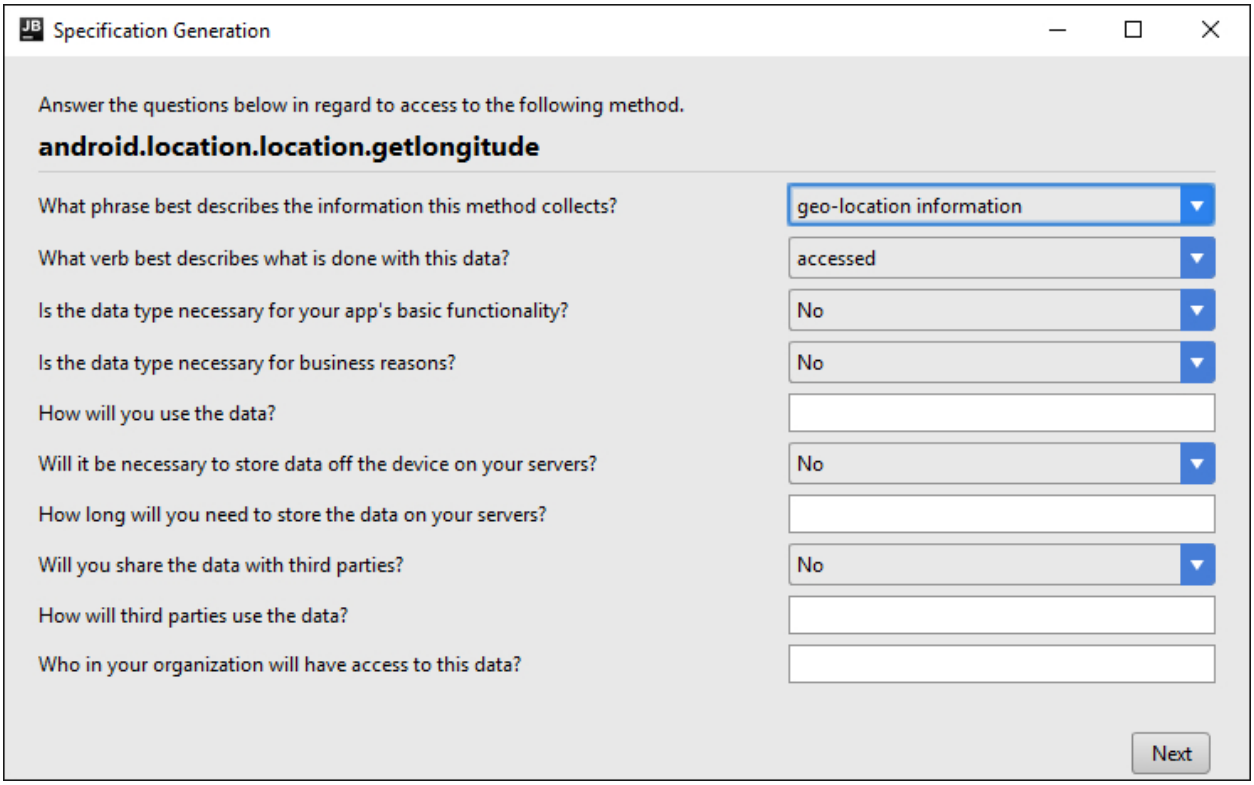



**Figure 7.5**: Specification Generator Wizard

alignment is flagged as weak (i.e., it does not have a phrase directly mapped to an API method, but the method is represented through a transitive relationship in the ontology), otherwise it is flagged as a strong misalignment (i.e., there is no relationship between any phrase in the ontology with the offending method call).

For our running example, $\omega$ = getIP(), $\Phi_\omega$ = {"ip address", "unique identifier"}. Based on the sample ontology in Figure 4.3, $\Psi_\omega$ = {"identifiable information", "address", "technical information"}. Since the phrase "address" appears in the privacy policy, $\omega$ can be reported as a weak misalignment even though its mapped phrases, $\Phi_\omega$ do not appear in the policy. This ability for the ontology to allow transitively-related phrases to represent methods helps to reduce false-positives by accounting for phrases with broader meanings.

Hovering over the highlighted code will reveal a tooltip including information about the misalignment (see Figure 7.4). Sample phrases $\Phi_\omega$ are suggested for insertion into the privacy policy in order to address the misalignment along with an indication of a weak or strong misalignment.

### 7.5.3 Specification Generation

PoliDroid-AS is also equipped with a privacy specification generator which reverse engineers a set of privacy specifications based on API invocations in the app's source code. The generator works by presenting the developer with a wizard that iterates through each invocation to an API method which has privacy phrase mappings. For each method, the developer is given a menu (Figure 7.5) that allows them to select or describe various details regarding the reasoning for using the privacy-sensitive method. Details include purpose, action verb (e.g. collection, sharing, etc), information type, if the data will be shared with third parties, and purpose of sharing. Once the questions have been completed, they are compiled into a textual representation of the privacy requirements (see Figure 7.6). In turn, these requirements can be used to construct a privacy policy for the app.

For each sensitive API method accessed for which a mapping exists (i.e., $\alpha \in A_{mapped}$), a set of specifications are generated as seen in Listing 7.2.

Listing 7.2: Specification Example

```
gps is COLLECTED with the following specifications:
# The gps data used is for the app's basic functionality.
# The gps data used is for business reasons.
# The gps data will be used for: to determine where you are
# The gps data will be stored off the device on servers.
# The gps data will be stored for 1 week on such servers.
# The gps data will NOT be shared with third parties
# Such third parties will use the gps data for: NOT SPECIFIED
# The gps data will be accessed by NOT SPECIFIED within the organization.
```

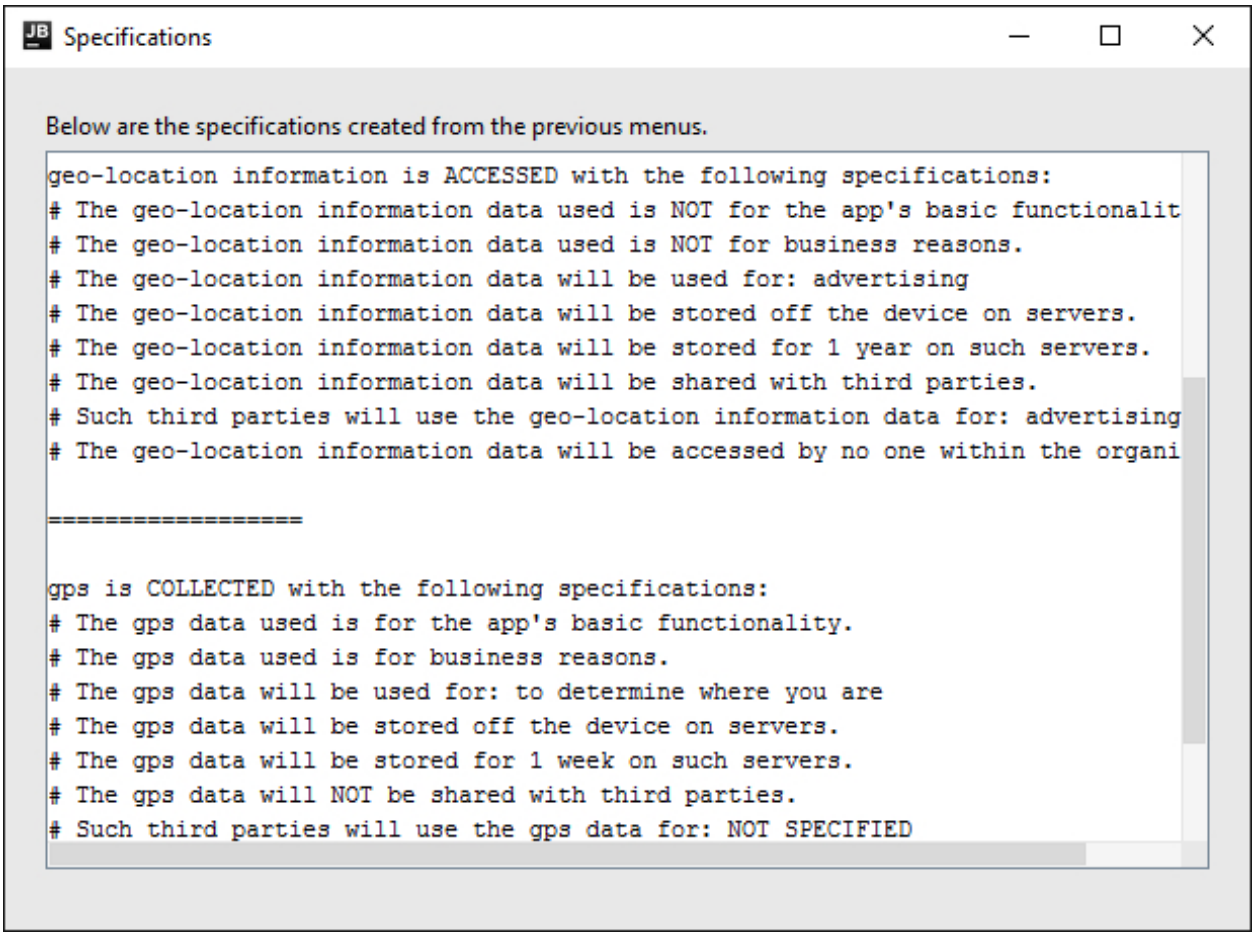



**Figure 7.6**: Generated Specifications

# CHAPTER 8: CONTEXTUAL POLARITY

Privacy policies often enumerate privacy information practices in simple sentences often of the form "we collect…". However, negative statements are also commonly used to declare the information not collected (e.g., "We do *not* collect location information."). By simply scanning a privacy policy for relevant terms, statements such as this can produce false negatives (which are arguably more dangerous that false positives when checking for privacy misalignment). For example, the phrase "location information" is present in the above statement. By ignoring the negation in the sentence, the algorithm described in Section 5.2 recognizes "location information" as information that may be collected by the application. The corresponding API methods would be included in $A_{represented}$, thus not being detected if invoked in the code.

A naive approach would be to simply negate any sentences with negative words. In the example above, the presence of the word "not" would simply have the opposite effect on the phrase (i.e., "location information" would be ignored when generating $\Phi$). Due to the nature of the English language, this approach is not feasible. For example, the phrase "We collect information including, but *not* limited to location information." states that "location information" is among the data that is collected. Negating sentences due to the inclusion of the word "not" would potentially result in a false negative by ignoring the phrase. Examples such as these are numerous in the English language and sentiment analysis is a difficult problem with much existing research [54, 58, 61, 66, 81].

## 8.1 Approach

The Sentiment Treebank created by Socher et al. includes over 215,000 sentiment labels for phrases in over 11,000 sentences [82]. These labels can be applied to existing machine learning techniques to train them for sentiment analysis. By definition, sentiment analysis classifies the sentiment of a sentence as positive, neutral, or negative. Since we are interested in strictly positive or strictly negative sentiment (i.e., ignoring conditions, an app either will or will not collect information), I propose the application of binary classification to single sentences for use in identifying

contextual polarity within privacy-related statements. When applied to the Recursive Neural Tensor Network (RNTN), sentiment detection accuracy for simple binary phrases is shown to reach 85.4% accuracy.

The Sentiment Treebank[1] and RNTN[2] (as included with the Stanford CoreNLP parse) are open source and available for modification and integration to existing tools. By modifying the parsers in PVDetector and PoliDroid-AS to integrate the RNTN, sentiment analysis can be directly integrated to source code and byte code policy misalignment detection. As described in Section 9.1.2, it is virtually impossible to calculate recall, and thus false positives, in the evaluation study on misalignment detection. For this reason, it is not until after the implementation of sentiment analysis that improvement in accuracy can be evaluated. With the inclusion of sentiment analysis, improvement would be shown by an decrease in false positives over the data set described in Section 9.1 by reducing the indications of collection where negative sentiment was ignored.

## 8.2 Implementation

The PoliDroid-AS plugin utilizes the Stanford Parser for identifying relevant collection sentences in privacy policies making it an ideal candidate for testing proof of concept. By applying sentiment labeling to PoliDroid-AS's policy parser, privacy-relevant phrases can be further filtered to only include those which state information collection. As described in Section 5.2, $\Phi$ is based on such phrases. Stanford's Deep Learning tool for sentiment analysis is included in the CoreNLP library allowing for the annotation of sentiment labels on parsed sentences. Since the plugin utilizes the same CoreNLP library as other tools in the tool suite, the resulting improvements can be applied to the other techniques.

The CoreNLP library parses phrases by tokenizing them into parts of speech and then annotating the phrase with linguistic information. By applying the sentiment property to the NLP annotator pipeline, PoliDroid-AS is able to determine a sentiment score for the sentence based on the Treebank model. Table 8.1 shows resulting scores for sentences similar to what can be found in

[1] http://nlp.stanford.edu/sentiment/treebank.html
[2] http://stanfordnlp.github.io/CoreNLP/

**Table 8.1**: Sentiment Scoring

| Phrase | Polarity | Score |
|---|---|---|
| We may not collect your address. | Positive | 0.2491 |
| We collect information including but not limited to your address. | Positive | 0.2549 |
| We never collect your address unless you allow us to. | Positive | 0.2663 |
| We may collect your address. | Positive | 0.3703 |
| We sometimes collect your address. | Positive | 0.3762 |
| We collect your address. | Positive | 0.4077 |
| We collect your address when you ask us to. | Positive | 0.4226 |
| We always collect your address. | Positive | 0.5475 |
| We don't collect information such as your address. | Negative | 0.2536 |
| We do not collect your address. | Negative | 0.2640 |
| We won't collect your address. | Negative | 0.2661 |
| We don't collect your address. | Negative | 0.2683 |
| We can't collect your address. | Negative | 0.2730 |
| We do not plan to collect your address. | Negative | 0.2854 |
| We never collect your address. | Negative | 0.2983 |
| We collect everything except your address. | Negative | 0.3310 |

a privacy policy. For each sentence, a ground truth of "Positive" or "Negative" is decided based on the sentence's meaning with regard to whether or not the data can be collected. For example, "We may collect your address." is considered Positive because it allows for the possibility of collection. The "Score" column reports the sentiment scoring from the CoreNLP library as a range from 0 to 1 with 0 being negative sentiment and 1 being positive.

Based on the data in Table 8.1, a threshold of 0.34 effectively removes all negative phrases from consideration at the cost of three false negatives (Figure 8.1). The false negatives all exhibit language commonly used in privacy policies as conditions or qualifiers (e.g., "including but not limited to", "unless", "may"). It is likely that these false positives are due to the Sentiment Treebank's creation from movie reviews, which do not generally use such qualifiers or conditions (i.e., reviewers do not often say they liked or disliked a movie only on certain conditions).

### 8.2.1 Improving Contextual Polarity Detection through Domain Knowledge

The Sentiment Treebank Model is based upon sentiment labels on sentences regarding movie reviews. The corpus [57] was labeled using Amazon Mechanical Turk as a platform for crowd sourc-

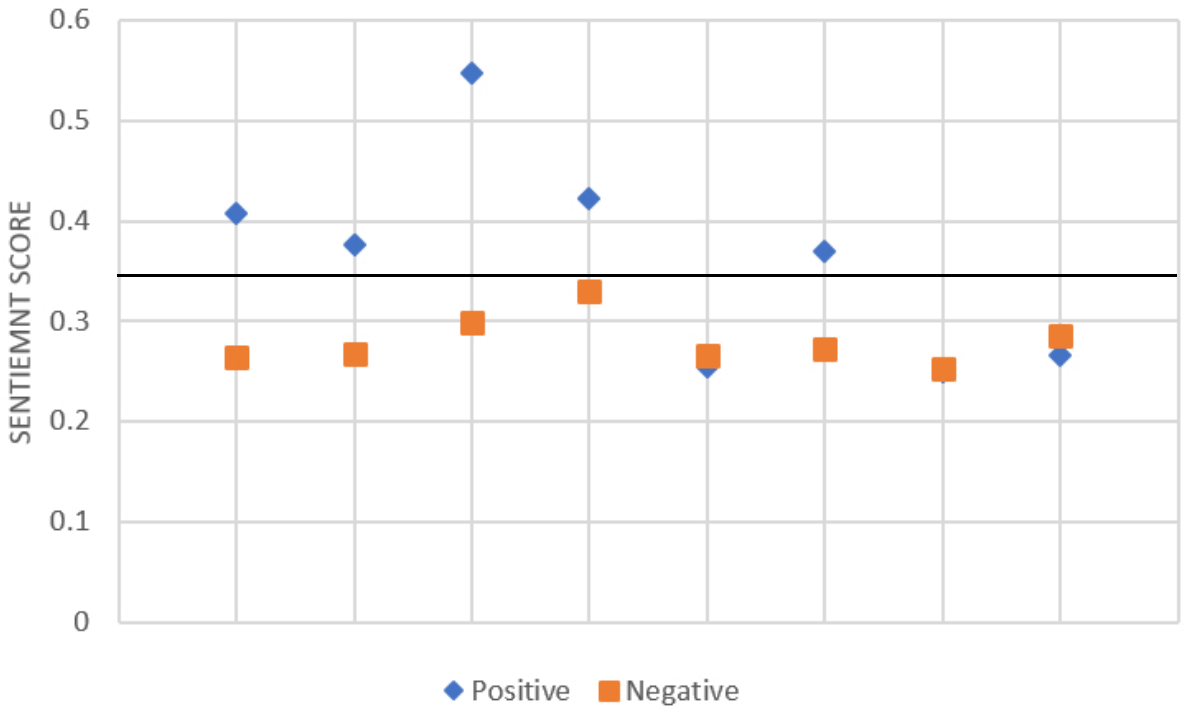



**Figure 8.1**: Sentiment Scoring

ing. Phrases from each sentence were presented to participants asking them to rate its sentiment on a 7-point scale rated "Very negative", "Negative", "Somewhat negative", "Neutral", "Somewhat positive", "Positive", and "Very positive" [82]. By combining the resulting labels for each sentence in a particular review, the overall rating can be interpreted.

When applied to privacy policies, this combination step is not necessary since polarity needs to be assessed at the phrase level. For example, "We do not collect your location" is a phrase of ideal granularity at which polarity should be determined. In applying sentiment analysis to PoliDroid-AS, labels were not combined to produce an overall sentiment of a privacy policy document. Instead, scores are determined at the phrase level in order to filter negatively-polarized collection phrases.

The method used to create the Sentiment Treebank can be applied to privacy policy phrases using the same corpus as described in Section 4.1.2. The resulting sentiment labels could then be applied in either in lieu of or supplemental to the Sentiment Treebank in order to incorporate phrases relevant to data collection.

# CHAPTER 9: EVALUATION

The evaluation of this work is split into three parts. First, the framework's viability in static analysis as described in Chapter 5 is demonstrated through an empirical evaluation over the top 300 free apps in the Google Play store. Second, the same methodology is applied to the runtime analysis techniques described in Chapter 6 to demonstrate its viability as both a validation technique for static detection as well as end user applications due to it's minimal overhead. Finally, the PoliDroid-AS IDE plugin described in Section 7.3 is evaluated through a user study to demonstrate the usefulness of the approach when applied to real-world applications.

## 9.1 Static Misalignment Detection

The static analysis techniques described in Chapter 5 were evaluated over the top apps on the Google Play store [78]. Google Play does not provide a list of known privacy policy misalignments (since such misalignments would likely result in a removal of the app from the repository). Thus, a ground truth of any existing misalignments does not exist so we designed our study with this in mind.

### 9.1.1 Study Setup

In this subsection, I introduce how we construct the data set for empirical evaluation, the metrics used, and the compared variants of our framework.

#### Data collection

The first step in our evaluation is to construct a data set of Android apps with their corresponding privacy policies. In particular, from the official Google Play market, we downloaded the top 300 free apps[1], as well as the top 20 free apps for each app category[2]. We combined all the down-

[1]The ranking of top 300 free apps is available at the Google Play website and was fetched on May 19th 2015.

[2]The list of app categories is available at the Google Play website, and the top 20 apps for each category were fetched on May 19th 2015.

loaded apps and acquired an app data set of 1,096 apps. Note that, although most Android apps have privacy policies, the app owners may put the policy at different places, such as their portal site at Google Play market, or a link in the main page of their company/organization. The privacy policy can also be in different formats, such as HTML, PDF, or Windows Word Document. Based on our observation, a large proportion of apps place their corresponding privacy policy at their portal websites at the Google Play market. Therefore, we crawled these websites and tried to automatically download the privacy policies of these apps. Furthermore, we considered only HTML privacy policies (the most popular format of privacy policies) in our evaluation for simplicity and avoiding potential noise in text extraction from various file formats. Note that, with proper text extraction tools, our framework can be applied to any format of privacy policies. Based on the automatic downloading and file format filtering, we collected privacy policies for 477 of the 1,096 apps, and thus generated a data set with 477 apps and their corresponding privacy policies[3].

**Evaluation metrics**

In our study, we measured the effectiveness of our framework by the number of misalignment detected, the number of true positives, false positives, and misalignments whose types (strong misalignments or weak misalignments) are mis-identified. It should be noted that, the ground-truth number of true misalignments in the entire data set is unknown, and thus the number of false negatives can not be calculated. In particular, we determined whether a detected misalignment is a true misalignment by manual inspection, and each misalignment was assigned to and inspected by two of the authors. For disagreements, a third author was assigned for reconciliation. When counting misalignments, we considered all invocations of the same method in an app as one misalignment.

**Evaluated techniques**

To measure the effectiveness of the misalignment detection technique, we considered two variants and compared them with our default technique.

[3]The data set is available at our project website: http://sefm.cs.utsa.edu/android_policy.

In our API mapping, we leveraged the knowledge from Android official documentation and crowd sourcing techniques to generate a fine-grained mapping between API methods and phrases in privacy policies. As a means to evaluate this fine-grained mapping, the first evaluation variant (referred to as "SUSI-only") used the coarse-grained SUSI API categorizations to map API methods to phrases. Specifically, we assigned phrases in the ontology to their most relevant SUSI categories (e.g., all descendants of the phrase "unique identifiers" in the ontology are assigned to the category of "unique identifiers"). Note that, since we focus on the collection of platform information only, the API methods in our mapping fall into one of the following 5 categories in SUSI: Unique Identifiers, Location Information, Network Information, Bluetooth Information, and No-Category (a SUSI category for API methods that are difficult to categorize), so we also assigned our policy phrases to these 5 categories. After the assignment of phrases, we mapped all phrases in a category in SUSI categories to all the API methods in the same category, and thus generated a new coarse-grained mapping based on SUSI only.

A second variant (referred to as "Keyword Search") was used to study the effectiveness of using light-weight NLP techniques, such as lemmatization, to filter out irrelevant paragraphs in privacy policies (described in Section 5.2). Specifically, this method did not use any filtering techniques and instead used a simple keyword-search-based strategy to extract phrases, $\Phi$, from the privacy policies.

### 9.1.2 Study Results

#### Overall results

We applied our default technique to the 477 pairs of apps and privacy policies in our data set. For the 477 apps, with a 30 minute time-out limit for each app, FlowDroid successfully processed 375 of them[4]. From these 375 apps, our default technique detected 402 misalignments in total, including 58 strong misalignments and 344 weak misalignments. Our manual inspection revealed that, among these detected misalignments, 341 were true misalignments, including 74 strong mis-

[4]Processing of the rest 102 failed due to time-out, heap overflow, or other exceptions.

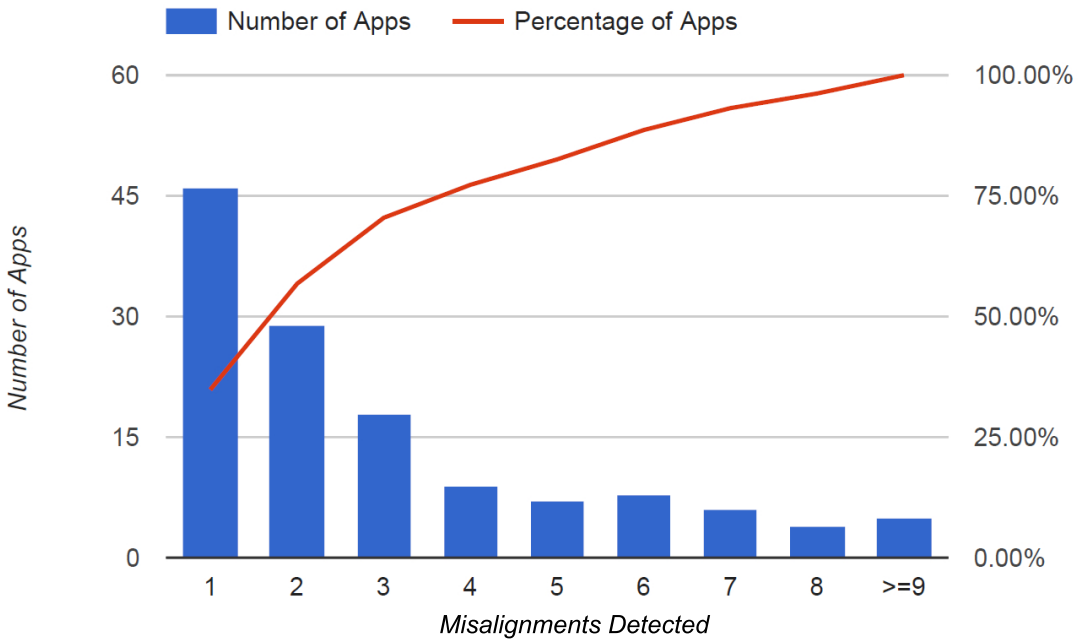



**Figure 9.1**: Distribution of Apps on the Number of Detected Misalignments

alignments and 267 weak misalignments (note that our framework mistakenly classified 19 strong misalignments as weak misalignments). The detection of 267 true weak misalignments shows that our privacy-phrase ontology is helpful on differentiate weak misalignments from strong misalignments.

We further studied how these misalignments were distributed among the apps, and the result is shown in the Pareto chart in Figure 9.1. In the figure, the blue bars represent the number of apps that have a certain number of misalignments, and the red line represents the proportion of apps that have misalignments smaller or equal to a certain number. From the figure, we can observe that, the misalignments are detected in 132 apps, and the majority of the apps had 3 or fewer misalignments. Specifically, the 74 true strong misalignments are from 31 apps, and the 267 true weak misalignments are from 101 apps.

### Study on detection errors

Our default technique generated three false positives for strong misalignments and 58 false positives for weak misalignments. From our manual inspection, the cause for the majority of the false

positives was that the privacy policies used phrases not in our ontology to describe the private information they collected. For example, a privacy policy used the phrase "carrier provider" to describe mobile network provider. Since the phrase is not in our ontology, our technique mistakenly determined that network information is not mentioned in the privacy policy and reported a false misalignment. Another relatively minor cause of false positives was that, in some cases, our NLP technique may have mistakenly filtered out paragraphs related to data collection. In such a scenario, the phrases in the removed paragraph would not be detected, so their corresponding API methods would not be added to $A_{represented}$ for the app and thus a misalignment would be raised.

Overall, our default technique was able to detect privacy-policy misalignments in a significant number of top Android apps and our false positive rate is relatively low, suggesting that developers do not need to waste much effort on inspecting false misalignments.

**Comparison of variant techniques**

To measure the effectiveness of our technique with regard to phrase and mapping granularity, we implemented the two variant techniques described in Section 9.1.1 and applied them to our data set. The three techniques detected 406 misalignments in total. Specifically, the Keyword-Search variant detected a proper subset of the misalignments detected by our default technique, and the SUSI-Only technique detected 3 weak misalignments and 1 strong misalignment that our default technique did not find. Our manual inspection found that 2 of the weak misalignments and the 1 strong misalignment were true positives. In Table 9.1, columns 3-6 presents the number of detected misalignments, the number of correctly-classified true positives, the number of mis-classified true positives, and the number of false positives, respectively.

From the table, we make the following observations. First, among the 3 techniques, our default technique was able to detect the most misalignments and achieved the highest type-classification accuracy (322/402 = 80.1%). Therefore, our default technique was more effective in general. Second, the SUSI-Only variant was only able to detect 97 misalignments, with 31 strong misalignments misclassified as weak misalignments. Inspection of the missed misalignments showed that

**Table 9.1**: Evaluation of Detected Misalignments

| Approach | Type | Detected | True Positives | Mis-classified | False Positive |
|---|---|---|---|---|---|
| Default Technique | Strong | 58 | 55 | 0 | 3 |
| | Weak | 344 | 267 | 19 | 58 |
| | Total | 402 | 322 | 19 | 61 |
| SUSI Only | Strong | 15 | 12 | 0 | 3 |
| | Weak | 82 | 44 | 31 | 6 |
| | Total | 97 | 56 | 31 | 9 |
| Keyword Search | Strong | 1 | 0 | 0 | 1 |
| | Weak | 389 | 261 | 74 | 54 |
| | Total | 390 | 262 | 74 | 55 |

the major reason for missed misalignments was that the SUSI categories are simply too coarsely grained. This was apparent particularly for the category of network information where an API method may be mapped to a phrase that has not much relation with it. For example, the method `getNetworkOperatorName()` should be mapped to "carrier network", but under the umbrella of network information, it is also mapped to "Wifi Access Points", "MAC Address", etc. Therefore, an app that sends carrier network information through this API method may be interpreted as not having a related misalignment because of the mistakenly mapped phrases are in the privacy policy. Third, the keyword-search technique is able to detect slightly fewer misalignments, but it cannot classify strong misalignments from weak misalignments. The reason is that, the more abstract a phrase is, the more likely that it appears in a paragraph that is not related to data collection. For example, the phrase "MAC Address" is almost always used to describe data collected, while the phrase "network" may be used for many different purposes (e.g., "social network"). Since the keyword-search variant cannot filter out paragraphs that are not data-collection related, it can mistakenly extract many abstract phrases from irrelevant sections. Under the umbrella of these abstract phrases, an API methods can easily find a map, and thus a strong misalignment is mistakenly identified as weak misalignment.

**Top privacy information types in detected misalignments**

To understand what types of private information are silently collected in privacy-policy misalignments, we studied the 10 API methods and phrases that were associated with most detected true misalignments. The results are presented in Figure 9.2 and Figure 9.3, respectively. In the two figures, the y-axis shows the name of the API or phrase, and the x-axis shows the number of strong misalignments (shown in gray) and weak misalignments (shown in black) associated with the API method or phrase. For brevity, we present only the short names of methods in Figure 9.2.

From Figure 9.2, we can observe that the top API methods associated with detected misalignments fall into four major categories. The API methods ranked 1st, 3rd, 8th, 9th, 10th in the list are all regarding mobile network information such as carrier network and mobile country code. The API methods ranked 2nd, 4th, and 5th are about GPS location information. The other two relatively smaller categories are `getUserAgentString()` (browser information) ranked the 6th, and `getDeviceID()` ranked the 7th. Similarly, Figure 9.3 shows a similar trend in that mobile country code (mcc) and carrier network are the most common missing phrases. The category of GPS location is related to the phrases ranked the 3rd, 4th, and 5th. The following two categories are different ways to describe device identifiers such as IMEI, UDID, etc., and the browser type. It should be noted that location information is one of the most important types of information involved in detected misalignments since it is required to be explicitly stated in privacy policies [2].

## 9.2 Dynamic Misalignment Detection

As described in Chapter 6, the API-phrase mapping has multiple uses in dynamic analysis apparent in the Parameter Detector module. This section evaluates its efficacy in dynamic misalignment detection in terms of detection effectiveness and performance.

### 9.2.1 Static Detection Validation

Evaluating the effectiveness of a user input detection tool as described in Section 6.3.1 presents two major hurdles. First, since the input data is not necessarily produced from standard well-

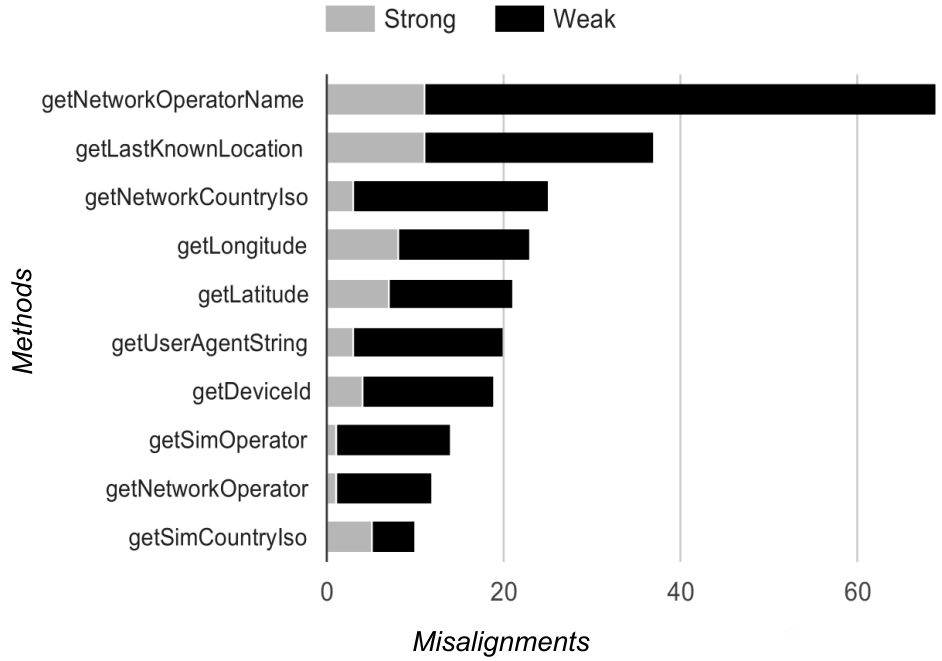



**Figure 9.2**: Methods with Most Detected Misalignments

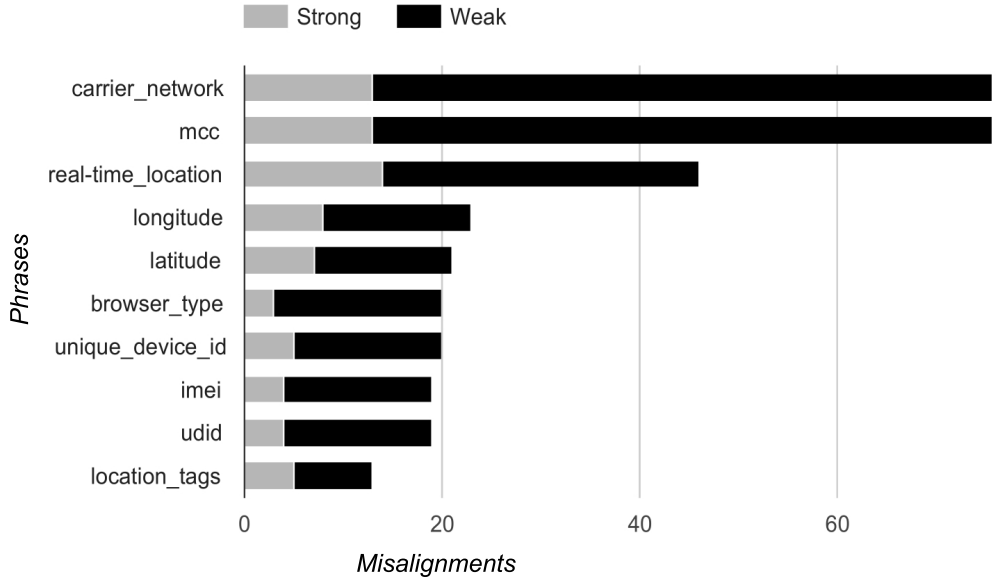



**Figure 9.3**: Terms with Most Detected Misalignments

known libraries (e.g., Android API), the source of the data cannot be anticipated. Although input fields are known, the methods for handling the input can vary based on how the developer chose to implement the app. Second, static information flow analysis is not sufficient for detecting leaks in flows originating in the UI. This is due to the nature of the interaction between the app source code and the activities that handle interaction with the UI. Because static information analysis is conservative when handling method-call relations, false positives are more easily generated (i.e., detection of information flows on infeasible program paths).

To address these issues, Parameter Reporter was used to verify the prediction of leaks of user-entered data through UI components of Android applications [86]. Due to the varying nature of implementation of UIs and the native code used to handle the data entered through the UI, identification of the data sent to known sinks is among the most reliable means to verify that data entered through the UI was in fact sent out through the leak. Parameter Reporter allows for this type of validation since any data sent to a sink will be passed as a parameter to that sink and thus logged.

For each validation detected using the techniques in Section 6.3.1, Parameter Reporter was configured based on the misalignment detected. The application was then manually operated in such a way that the misalignment was triggered, after which the Parameter Reporter's log files were checked to verify the misalignment. Using the technique described in Section 6.2, Parameter Reporter was able to verify the detection of 20 strong misalignments and 10 weak misalignments.

For this technique, Parameter Reporter was used to verify misalignments that were already suspect based on static analysis techniques. However, by providing Parameter Reporter with a sufficient list of potential sinks (e.g., SuSi), it can detect arbitrary misalignments at runtime without prior knowledge of the misalignment. The success of Parameter Reporter provides a basis for many uses of the tool such as real-time policy-based validation.

### 9.2.2 Performance

Part of Parameter Reporter's utility is its applicability to end users and auditors to detect misalignments in real time. For this reason, performance is a key aspect to the tool. By its nature, Parameter Reporter adds instructions to the calls in question. If the overhead incurred by these instructions is too high, the responsiveness of the app being monitored will be affected. In measuring both method and boot time overhead, the performance hit was virtually imperceptible.

Parameter Reporter requires not only extra instructions to execute for the logging of data leaks, but the inclusion of method hooks for the detection of specific API calls. To measure the overhead imposed by these mechanisms, I utilized Android SDK's Method Tracer[5] to measure the time for individual method calls as they are executed during runtime on the device. As seen in Figure 9.4, Method Tracer reports two metrics important to determining overhead: *Exclusive Time*, the time it takes for a method to execute not including any time spent executing children calls and *Inclusive Time*, the time spent executing the method including its children. Both of these measurements are in microseconds. Inclusive and exclusive overhead are calculated by measuring the average difference between method execution both with and without the module loaded.

The experiment was conducted on a Motorola Nexus 6 loaded with Android 5.0.1 The Nexus 6 was developed in collaboration with Google to serve as the launch device for Android 5. For this reason, the device is shipped with a stock Android OS not including any extra carrier software making it an ideal device for benchmarking Android software. As required by Parameter Reporter, Xposed version 86 was also installed on the device.

In order to measure method execution time in a controlled environment, a test app (Listing 9.1) consisting of a simple form which, when submitted via a "Send" button would call a known sink (`org.apache.http.client.methods.HttpPost()`) was loaded onto the device using Android Studio's debug tool. The tool accesses the device via Android Debug Bridge (ADB) to allow testing directly on the physical device as opposed to a virtual machine. As described in Section 6.2, Parameter Reporter was configured to monitor the app for invocations of

[5]https://developer.android.com/studio/profile/am-methodtrace.html

`org.apache.http.client.methods.HttpPost()`.

Listing 9.1: Test Code

```
1  public class InputForm extends AppCompatActivity {
2     private static final String POST_URL = "http://galadriel.cs.utsa.edu";
3
4     protected void onCreate(Bundle savedInstanceState) {
5        super.onCreate(savedInstanceState);
6        StrictMode.ThreadPolicy policy = new
             StrictMode.ThreadPolicy.Builder().permitAll().build();
7        StrictMode.setThreadPolicy(policy);
8        setContentView(R.layout.activity_input_form);
9     }
10
11    public void onButtonClick(View v) {
12       if (v.getId() == R.id.submit) {
13          TextView text = (TextView) findViewById(R.id.testData);
14          String data = text != null ? text.getText().toString() : "TEST DATA";
15          postData(data);
16       }
17    }
18
19    private void postData(String data) {
20       HttpClient client = new DefaultHttpClient();
21       HttpPost post = new HttpPost(POST_URL);
22       List<NameValuePair> nameValuePairs = new ArrayList<>();
23       nameValuePairs.add(new BasicNameValuePair("data", data));
24       try {
25          post.setEntity(new UrlEncodedFormEntity(nameValuePairs));
26       } catch (IOException e) {
27          e.printStackTrace();
28       }
29    }
30 }
```

To test execution time of `HttpPost()` with Parameter Reporter enabled including any over-

**Table 9.2**: Parameter Detector Benchmark Statistics

| **Measure** | **Inclusive Average ($\mu$s)** | **Inclusive Median ($\mu$s)** | **Exclusive Average ($\mu$s)** | **Exclusive Median ($\mu$s)** |
|---|---|---|---|---|
| With Parameter Detector | 4802.7 | 3699.0 | 27.8 | 24.0 |
| Without Parameter Detector | 4150.9 | 4081.5 | 23.1 | 23.0 |
| Difference | 651.8 | -382 | 4.7 | 1 |

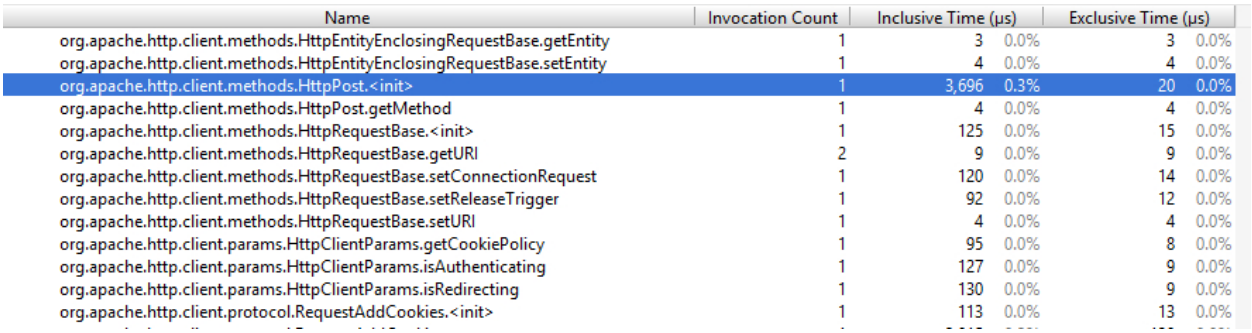



**Figure 9.4**: Android Method Tracer

head caused by Parameter Reporter, Parameter Reporter was enabled via Xposed on the device and the app was executed on the device through ADB in Android Studio. Method Tracer was then manually invoked from within Android Studio by starting a trace, pressing the "Send" button in the test app on the device, and ending the trace. A resulting trace, such as the one in Figure 9.4 would then display inclusive and exclusive time for all methods invoked during the trace. By locating `org.apache.http.client.methods.HttpPost()` in the "Name" column, the values could be recorded. This sequence was repeated for a total of 50 times with Parameter Reporter and Xposed enabled and 50 times without them enabled. The resulting data is seen in Figures 9.5 and 9.6 with averages and medians reported in Table 9.2.

As seen in Table 9.2, inclusive and exclusive times incurred an extra 651.8$\mu$s and 4.7 $\mu$s respectively. These times would not increase with method time as they represent only the time taken for the instrumented code to execute. For this reason, the percentage overhead will only decrease as the method execution time increases. Furthermore, the overhead is only incurred on methods for which the hook triggers (i.e., sinks reporting to logs). For other methods, or methods in apps not being monitored, Parameter Reporter's control flow does not allow for `reportCall()` (Listing 6.1) to be called.

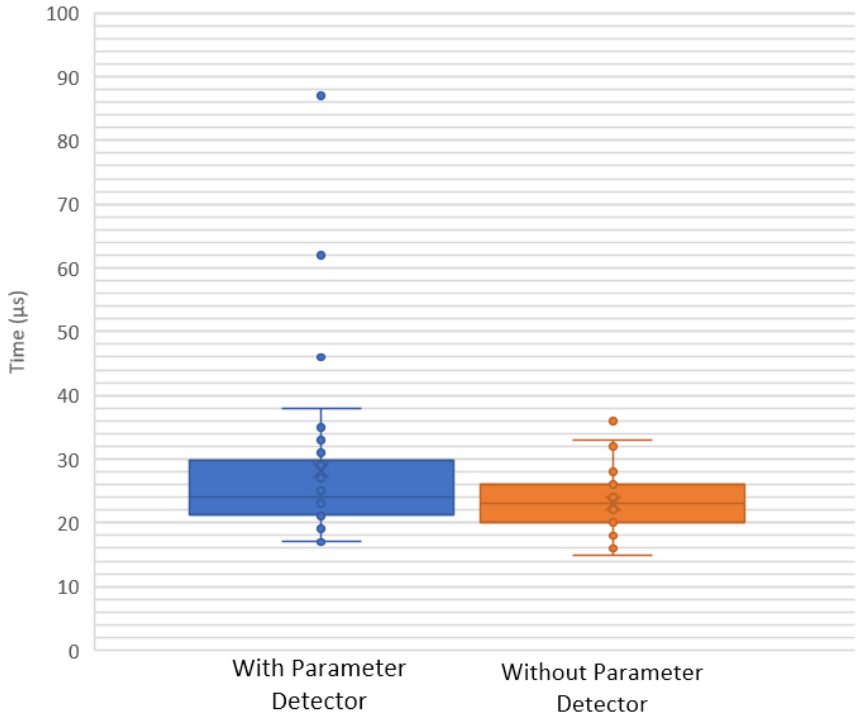



**Figure 9.5**: HttpPost() Exclusive Execution Times

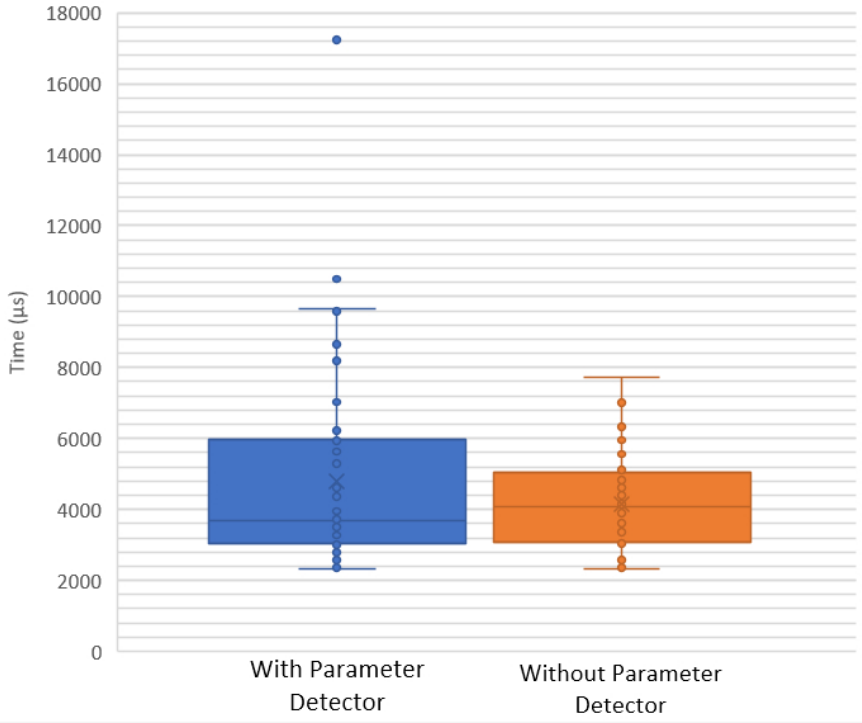



**Figure 9.6**: HttpPost() Inclusive Execution Times

**Table 9.3**: PoliDroid-AS Case Study Misalignments

| **Participant** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **Detected Misalignments** | 3 | 0 | 0 | 1 | 0 | 1 | 0 |

## 9.3 Real-World Application

The framework established by this research is meant for the benefit of real-world apps. The following user studies were conducted over Android developers with their own source code to demonstrate its viability in real-world scenarios.

### 9.3.1 PoliDroid-AS Study

The first study was done over the PoliDroid-AS plugin to verify its ability to utilize the artifacts produced by this research for misalignment detection in production apps. Seven developers working on their own pre-existing Android apps in Android Studio were recruited to conduct a case study on the usage of PoliDroid-AS's misalignment detection and specification generation. The participants included graduate and undergraduate computer science students and a computer science faculty member. All participants had experience with Android Studio.

Prior to using our tool, we had the participants underwent a training session for the creation of privacy policies based on the California attorney general's guidelines for privacy for the mobile ecosystem [33]. Participants were then asked to create a privacy policy for their apps.

Participants tested the misalignment detection feature of the plugin on computers pre-loaded with Android Studio and PoliDroid-AS. They were asked to import their code and privacy policies, open each class file, and report any misalignments detected by the plugin. Table 9.3 shows the results. Three of the participants found misalignments using the plugin. The misalignments all fell into the categories of location and network information. It interesting to note that each participant addressed their misalignments by updating their policy as opposed to the code.

Participants two and five reported that their apps simply did not collect sensitive information. We consider the data from these groups to still be relevant in that they did not exhibit false positives. Participant seven's code did not exhibit misalignments due to the fact that the plugin could not

**Table 9.4**: PoliDroid-AS Specification-Policy Comparison

| **Participant** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **Included by Policy** | 6 | 0 | 0 | 5 | 0 | 5 | 0 |
| **Excluded by Policy** | 4 | 0 | 0 | 4 | 0 | 5 | 0 |

detect invocations from within compiled libraries. We found that participant three's code called upon the `Build.MODEL` field, a field containing the device's model, which was not detected. This is a limitation of the framework, which only considers methods, and not the plugin itself.

Participants were also asked to use the privacy specification generation feature of the plugin to create specifications for their apps. They were asked to then annotate their original privacy policies noting if and where the information in the specifications was included. The results are seen in Table 9.4. As described above, groups two, three, five, and seven did not report any misalignments. For the remaining groups, an average of 55.2% of the information generated by the specification generator was not included in the participants' privacy policies. This shows that specifications generated by the tool would help in reducing omissions in policies.

During a post-survey discussion of the tool's usability, three participants reported issues with responsiveness of the plugin. When detecting sensitive API method invocations, the plugin would take approximately 15 seconds to begin reporting them. All participants who found misalignments in their code and thus updated their policies noted that after updating a privacy policy, the entire IDE would need to be reloaded to reflect changes. These errors were due to technical bugs in the implementation of the plugin and not the framework methodology itself. The reporting delay issue was resolved by triggering IntelliJ's code inspection immediately after loading a policy file and the policy loading issue was resolved by refreshing the VirtualFile object representing the policy upon reloading it.

### 9.3.2 PoliDroid-AS Followup Study

Based on the observations from the PoliDroid-AS study in Section 9.3.1, the plugin was improved to account for usability errors resulting from the technical bugs reported. Furthermore a script was developed to better record responses (Appendix A.2.1). The goal of the followup study was to

verify the resolution of the technical issues.

The faculty member recruited for the previous study was the subject of the followup study. The participant loaded the plugin onto their own machine after downloading the compiled version directly from Github[6]. This version included the two bug fixes described in the previous section. After installing the plugin, the subject was instructed to follow the directions described in Appendix A.2.1 via a web-based survey constructed using the Qualtrics [7] data collection and analysis tool.

The study results confirmed the resolution of the bugs present in the previous study. Furthermore, the privacy specifications generated by the plugin identified all invocations of private-information-producing API method calls. Also, while all invocations were identified in the source code, the participant's manually-written privacy policy (Question 5b) was well-aligned resulting in no misalignments (i.e., false positives) detected by the plugin.

[6]https://github.com/rslavin/PoliDroid-AS/releases/tag/v1.0.1
[7]https://qualtrics.com

# CHAPTER 10: RELATED CONTRIBUTIONS

I have been involved in other privacy-oriented research which has contributed to this work by way of both establishing motivation and addressing the assumption of secure programming practices necessary for automated policy alignment. This work is described here.

## 10.1 Sequence Diagram Aided Policy Specification

In an effort to verify consistency between policy and code, we have introduced the use of unified modeling language (UML) sequence diagrams to specify privacy policies in a formalized manner that can be translated to Linear Temporal Logical (LTL) [74]. Such a formal representation of a policy can be directly analyzed and compared to code logic. This work, however, demonstrated the difficulty involved in policy reconciliation.

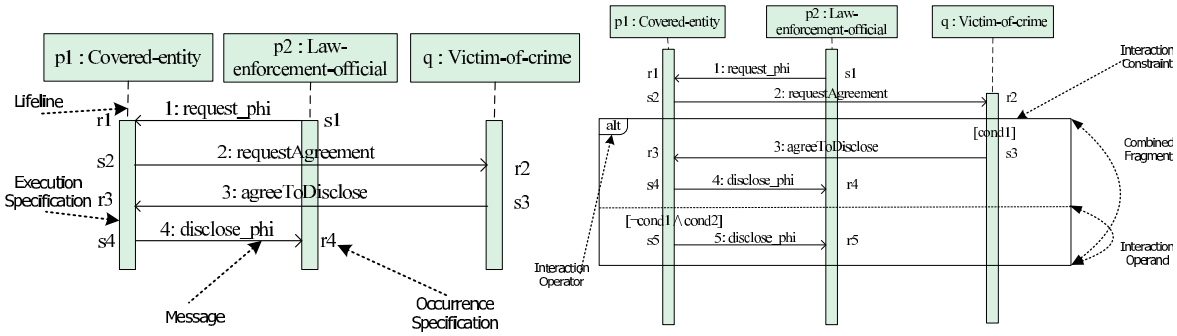
A graphical representation allows decision-makers such as application domain experts and security architects to easily verify and confirm the expected behavior. Once intuitively confirmed, an algorithmic approach can be used to formalize the semantics of Sequence Diagrams in terms of LTL templates.

The approach uses LTL to formalize the semantics of UML 2 Sequence Diagrams with all the Combined Fragments (CF), including nested Combined Fragments and Interaction Constraints (IC). These constructs allow different types of control flow for presenting complex and concurrent behaviors. In all the templates, different semantic aspects are expressed as separate, yet simple LTL formulas that can be composed to define the complex semantics of Sequence Diagrams. The formalization enables us to leverage the analytical powers of automated decision procedures for LTL formulas to determine if a collection of Sequence Diagrams is consistent, independent, etc. and also to verify if a system design conforms to the privacy policies leveraging existing automated procedures such as model checking for rigorous formal analysis. In this previous work regarding sequence digram aided privacy policy specification, we used this method to model Health Insurance Portability and Accountability Act (HIPAA) regulations and test various health information data

flows for conformance.

### 10.1.1 Encoding Semantics

The semantics of Sequence Diagrams are described in terms of traces by the Object Management Group [55]. However, it is not formally defined how traces are derived compared to their precise syntax descriptions [55]. As the first step of defining a Sequence Diagram using LTL formulas, we begin with presenting the semantics of the basic Sequence Diagram, then discussing the structured control constructs [75].




**a. HIPAA** $\S 164.512(f)(3)(i)$ **b. Combined Fragment (HIPAA** $\S 164.512(f)(3)$**)**

cond1 = coveredEntityObtainAgreement
cond2 = emergency/\needToDetermineCrime/\notUsedAgainstVictim/\activityAdverselyAffectedByWait/\
inBestInterest

**Figure 10.1**: Sequence Diagram Syntax

#### Basic Sequence Diagram

We refer to a Sequence Diagram without Combined Fragments as a basic Sequence Diagram (Figure 10.1a for an example with annotated syntactic constructs). A *Lifeline* is a vertical line representing a participating object. A horizontal line between Lifelines is a *Message*. Messages are of two types: asynchronous and synchronous. (A synchronous Message is depicted with a block arrow head and must have an explicit reply Message. Its formal semantics can be found in Section 5.6 of [73]) Each Message is sent from its source Lifeline to its target Lifeline and has two endpoints. Each endpoint is an intersection with a Lifeline and is called an *Occurrence Specification (OS)*, denoting a sending or receiving event occurrence within a certain context, i.e., a Sequence

Diagram. OSs can also be the beginning or end of an *Execution Specification* [55].

The semantics of a basic Sequence Diagram is defined by a set of traces. A trace is a sequence of OSs expressing Message exchange among multiple Lifelines. We identify four orthogonal semantic aspects, each of which is expressed in terms of the execution order of concerned OSs.

1. Each OS can execute only once, i.e., each OS is unique within a Sequence Diagram.
2. On each Lifeline, OSs execute in graphical order.
3. For a single Message, the sending OS must take place before the receiving OS does.
4. In a Sequence Diagram, only one object can execute an OS at a time, i.e., OSs on different Lifelines are interleaved.

**Interaction Operator**

Each Operator has its specific semantic implications regarding the execution of the OSs enclosed by the CF on the covered Lifelines. The Operators that are utilized in the article are summarized as follows. (We provide the complete list of Operators in [73]):

- **Alternatives**: one of the Operands whose Interaction Constraints evaluate to $True$ is nondeterministically chosen to execute.
- **Option**: its sole Operand executes if the Interaction Constraint is $True$.
- **Parallel**: the OSs on a Lifeline within different Operands may be interleaved, but the ordering imposed by each Operand must be maintained.
- **Loop**: its sole Operand will execute for at least the minimum count (lower bound) and no more than the maximum count (upper bound) as long as the Interaction Constraint is $True$.
- **Assertion**: OSs on a Lifeline must occur immediately after the preceding OSs.
- **Negative**: its Operand specifies forbidden traces.

- **Weak Sequencing**: *on a Lifeline*, the OSs and CFs within an Operand cannot execute until the OSs and CFs in the previous Operand complete.

**Sequence Diagram Deconstruction**

To facilitate codifying the semantics of Sequence Diagrams and nested CFs in LTL formulas, we show how to deconstruct a Sequence Diagram and CFs to obtain fine-grained syntactic constructs. Recall that OSs and CFs directly enclosed in the same Operand or Sequence Diagram are combined using Weak Sequencing, constraining their orders with respect to each individual Lifeline only [55]. We further deconstruct a Sequence Diagram into syntactic constructs on each Lifeline.

We project every CF $cf_m$ onto each of its covered Lifelines $l_i$ to obtain a *compositional execution unit (CEU)*, which is denoted by $cf_m{\uparrow_{l_i}}$ (e.g., OSs r3, s4, and s5 are grouped into a CEU (Figure 10.1b)). Every Operand $op_n$ of CF $cf_m$ is projected onto each of its covered Lifelines $l_i$ to obtain an *execution unit (EU)* while projecting $cf_m$ onto $l_i$, denoted by $op_n{\uparrow_{l_i}}$ (e.g., OSs r3 and s4 are in an EU (see Figure 10.1b)). If the projected Interaction Operand contains a nested Combined Fragment, a *hierarchical execution unit (HEU)* is obtained; otherwise a *basic execution unit (BEU)* is obtained, i.e., an EU is a BEU if it does not contain any other CEUs. Projecting a Sequence Diagram onto each enclosing Lifeline also obtains an EU. In an HEU, we also group the OSs between two adjacent CEUs or prior to the first CEU or after the last CEU on the same level into BEUs (e.g., OSs r1 and s2 are grouped into an BEU (see Figure 10.1b)).

The deconstruction enables us to focus on the order of constructs on each Lifeline, i.e., a composition of (nested) CEUs at different levels. On each Lifeline, the OSs and CEUs directly enclosed in the Sequence Diagram, which are considered at the highest-level, are ordered sequentially. The semantics of the CEUs are represented at a lower-level, where the order of the EUs directly enclosed in each CEU depends on its Interaction Operator. Similarly, the directly enclosed OSs and CEUs (if any) within each EU are ordered sequentially at the next level. A BEU is considered at the lowest-level. In this way, the semantics of the constructs on a Lifeline can be described recursively. The inter-Lifeline semantics is defined by the Messages and the interleaving seman-

tics, i.e., the receiving OS must happen after the sending OS of the same Message, and only one Lifeline can execute an OS at a time. The inter-Lifeline semantics rules are independent from the intra-Lifeline semantics rules, which make the semantics rules composable. Further details on the deconstruction are provided in [73].

### 10.1.2 Sequence Diagram Translation Tool

Exhaustive Sequence Diagram representations with large numbers of traces can be tedious to construct. For example, when multiple messages are allowed to be interleaved, the set of possible traces are less obvious. To address this issue, I have designed a tool, Sequence Diagram Translator (SDT)[1],to translate Sequence Diagrams into LTL formulas which can be used with a model checker to identify all traces and test for security properties.

First (Figure 10.2), SDT takes as input a Sequence Diagram created in Papyrus UML[2], a modelling plugin for the Eclipse™IDE. The Sequence Diagram is converted from .uml format into an XML representation (to support manual generation) by the PapyrusToXML component in SDT. Next, a Java-based parser reads the XML file and passes an internal representation to the program's LTL generator, which implements the LTL templates to generate a LTL formula representing the Sequence Diagram. The LTL generator is designed to take collections of Sequence Diagrams and generate the corresponding formulas, and the necessary model to be used in NuSMV [19] model checker. SDT implements all of the necessary template formulas to handle all CFs, nested CFs, and Constraints.

While SDT removes the burden of manually translating sequence diagrams to LTL, the construction of sequence diagrams is not trivial. For the case of privacy policies where the enumeration of privacy specifications are less complex than regulations, a more user-friendly notation may be viable. In such a case, SDT can be modified to accept the new input and potentially be integrated into POLIDROID. Such a goal is out of the scope of this research.

[1] http://github.com/rslavin/Sequence-Diagram-Translator
[2] https://eclipse.org/papyrus/

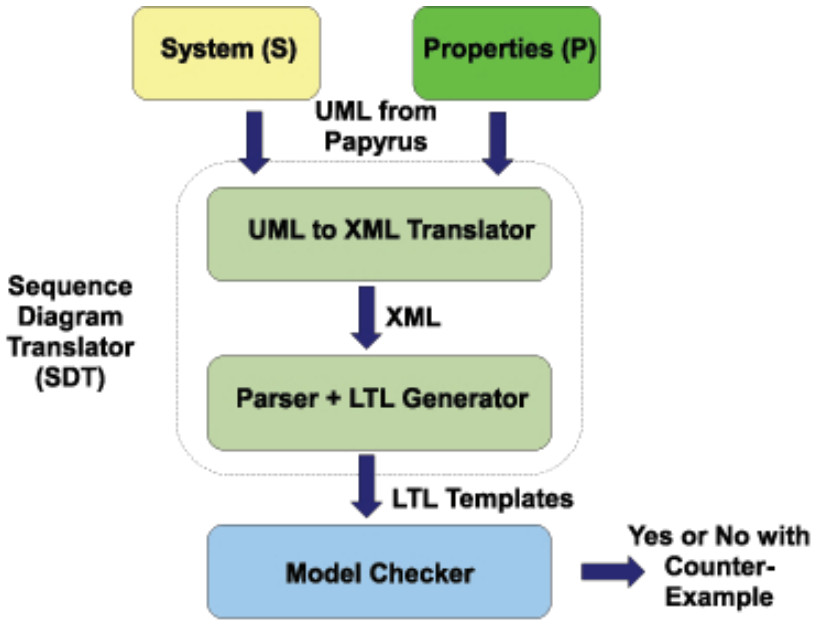



**Figure 10.2**: SDT Architecture

## 10.2 Security Requirements Patterns

Accurately-aligned privacy policies are based on the assumption that the corresponding code does not unintentionally leak private data. The basis for this assumption lies in best practices for security requirements In a prior work, I addressed this assumption for novice users by applying hierarchical data structures, feature diagrams, to known security requirements patterns as a measure for improving pattern selection practices [76]. In a study over seven industry and academia security experts and 34 novice students, we verified the efficacy of the approach as both improving novice selection and reducing selection time. Privacy policy alignment with application code can thus be improved through the use of such a technique.

A *requirements pattern* is a structure that engineers can use to generate one or more requirements for a recurring situation [45]. Based on design patterns [7], each requirements pattern describes a recurring problem as well as a core solution which can be used repeatedly, but not necessarily in the same way every time. Because situations vary, a requirements pattern must be parameterized to selectively control for those effects that vary across problem spaces. For example, an engineer may choose between single sign-on [49], where users need only one username

and password for logging into different systems, as means to maximize usability, where they prefer more complex role-based access control (RBAC) [71] requirements to maximize confidentiality by compartmentalizing access to information. In this example, the engineer perceives constraints differently and desires different levels of control: more control over different systems to manage logins, or more control over classification of resources into roles.

Requirements patterns incorporate engineering knowledge into the solution space of a pattern in order to provide a basis for requirements elicitation and generation. For example, patterns can itemize pre-conditions to indicate when a pattern applies to a given scenario, or questions whose answers direct the engineer from one solution to another (e.g., from single sign-on to RBAC). Such knowledge reuse allows software engineers to solve problems in a more effective manner. That is, patterns consist of tried-and-tested solutions that have been shown to be effective when applied in the correct context. Consequently, patterns serve as a common language for software engineers to document their design decisions [12].

Security requirements patterns are a special case of requirements that address security risks in a system. Historically, security is dealt with using a penetrate-and-patch approach [52], wherein security problems are identified and addressed in response to penetration testing of the fully-functional system, sometimes in a post-deployment situation. If the system failures are intrinsic to the design, then significant rework is required [15]. Alternatively, it is more cost effective to identify security flaws early in the requirements and design stages of development, which is the focus and intent of security requirements patterns [92].

In the wild, security requirements patterns appear mostly in isolation, either in small sets of related patterns or repositories and related only by a common topic or theme [41,89]. Combined with the lack of guidance for pattern selection [39], engineers lack the structure needed to holistically address security and balance complex forces or quality attributes (e.g., usability and confidentiality). As the number of security patterns continues to grow [72], engineers face an increased challenge in recognizing what patterns to select [88]. The contribution of our approach is as follows: (1) we demonstrate that security requirements patterns can be linked into a hierarchy to make

problem space trade-offs more salient and to connect related patterns to comprise holistic solution spaces; and (2) we evaluate this new format and hierarchy in a user study to measure speed and correctness in selecting patterns to apply to example scenarios.

We evaluate our approach in a user study consisting of graduate and undergraduate students at the University of Texas at San Antonio (UTSA). The study examined our method's ability to deduce the most appropriate set of patterns by having novice users with limited security knowledge and experience use a pattern hierarchy to select patterns for a scenario. We then compared the results with selections that experts chose for the same scenario. Based on our analysis, we found that novice users not only were more accurate in their pattern selections, but were able to select patterns more quickly than users without a hierarchy. From these results and further observations we assert that the use of pattern hierarchies can improve the usability of security requirements by providing a means for easier access and faster selection as well as better documentation of related patterns.

### 10.2.1 Pattern Template

The presentation of pattern content varies depending on the pattern author. In requirements engineering, formal and semi-formal methods, such as Tropos [18] and Problem Frames [38], structure requirements knowledge into pattern-like representations. However, many approaches that principally identify themselves as patterns use natural language templates (NLT) to illustrate patterns [26, 56]. These NLT approaches tend to base their template on the so-called Gang-of-Four (GoF) [29] book which outlines the following essential pattern elements:

- *Pattern Name* - A simple phrase to describe the problem
- *Problem* - A description of when to apply the pattern
- *Solution* - A general arrangement of the elements used to solve the problem
- *Consequence* - The results and trade-offs of applying the pattern

Our approach includes two additional important features: forces, which determine quality attributes that are impacted (maximized, minimized, etc.) by the pattern, and relations or links to other patterns using an inquiry-based approach. For example, the access control requirements pattern may contain such forces as generalizability, flexibility, and modifiability since they would be particularly relevant to its application and a question such as, "How are authorization privileges bound to actors and resources?" may lead to an authentication pattern.

By balancing forces using questions which invoke discussion and refinement of security requirements, we aim to minimize the negative consequences or liabilities of the pattern (e.g., access control may require many rules and increase the system's complexity) as well as capture the applicability of the pattern [93]. The use of an approach based on the ICM provides a way to connect existing security requirements patterns as well as refine system requirements themselves through questions that surface hidden assumptions about the system and its environment. The approach works on two levels: (Level 1) questions relevant to the problem can either draw out specific information from the situation which can be applied to generic requirements in the template in order to generate specific requirements, or (Level 2) they relate the pattern to other patterns relevant to the situation as a means to increase requirements coverage by identifying dependencies or to introduce alternative requirements that balance forces in alternative ways that may be better suited to a particular problem or stakeholder needs. Pattern hierarchies are constructed using the second level, which further supports eliminating unnecessary or unfitting requirements to select the most appropriate pattern.

To catalog patterns, we used a standardized pattern template consisting of six elements. The template we used is an elaboration of ideas devised at RePa'12 and our previous work on the Standardized Pattern Format for deriving characterizations and boundaries of patterns [77]. Using this pattern template is what enables us to construct a pattern hierarchy. To illustrate the different aspects of the template, we use access control as an example.

The template consists of the following elements:

- *Name* - a unique name that is limited to the scope of the pattern

- *Problem* - a statement of the problem to be addressed or a high-level goal to be achieved
- *Context* - domain assumptions that must be true in conjunction with the generated requirements
- *Forces* - the non-functional quality attributes that create trade-offs for the application of the pattern
- *Solution* - a set of questions which refine requirements and guide the pattern user to related patterns
- *Management* - additional information related to the pattern including examples and known uses

We now describe the use of each of the attributes using access control.

**Name** For our running example, we name the pattern "Access Control" to describe what the pattern covers. In addition, it can be uniquely incorporated into the pattern hierarchy.

**Problem** The problem shall be expressed either as the security objectives that need to be achieved or the threats that must be mitigated. For our example, we define the problem as "the confidentiality and integrity of resources shall be protected by regulating access to the resources based on different factors." Here, confidentiality and integrity, two fundamental security attributes [8], are addressed.

**Context** A pattern addresses a generic problem in a specific context [72] , which shall describe the nature of the situation. This includes any domain assumptions as well as expectations of the system and its environment. For access control we describe the context as "any computer-driven environment in which the access to the resources needs to be regulated" and we list the following assumptions:

- The administrator involved in implementing the authorization system shall be trusted.

- The actors involved in granting or denying authorization shall have the ability to restrict access to the resources being protected.
- Actors shall not assume the identity of other actors with different rights.

We also include one expectation:

- Actors will not circumvent the system to gain access to resources.

**Forces** Forces that must be balanced to make a decision are necessary for the creation of requirements in order to provide boundaries and limits. We believe it is useful to list these potentially conflicting or complementary goals as quality attributes in order to better balance the trade-offs of benefits and cost that are imposed by the requirements [48]. For access control we list three forces:

- Generalizability - The authorization structure must be independent of the type of resources.
- Flexibility - The authorization structure shall be flexible enough to accommodate different types of principals (users or subsystems), objects, and rights.
- Modifiability - It should be easy to modify the rights of a principal in response to changes in his or her or its duties or responsibilities.

**Solution** To solve the problem and balance the forces, the solution includes a series of ICM-inspired questions that help elicit responses from a user or a subject matter expert. A set of requirements templates with variable parts may be configured by the developer and may correspond directly to the questions. The answers to the questions may be pre-conditions to asking additional questions, pieces to fill in the requirements templates, or guides to other patterns. The questions for our example are:

- Which entities (principals) exist in the system and what resources do they need to access?
- Can two entities access the same resource at the same time?

- How are the resources intended to be accessible?
- Can users of the system be categorized into roles that will have different access privileges? *See Role-Based Access Control pattern*.
- Can resources be assigned labels by the system so that users are given clearance to access resources based on levels of clearance? *See Mandatory Access Control pattern*.
- Can access to resources be regulated by the owner of the resource? *See Discretionary Access Control pattern*.
- How are entities to be authenticated in order to gain access to the system? *See Authentication pattern*.

The requirements template for our example consists of:

- <list of entities> shall <be permitted | not be permitted> to access <resource> simultaneously.
- <list of entities> shall access <resource> through the use of <action>.
- An entity shall be granted or denied privileges to <resource> based on <authorization criteria>.

Some questions are accompanied by italicized text. This text refers a pattern-user to a corresponding pattern [72, 89] for the question (as seen in Figure 10.3). The requirements template also lists generic requirements which can be refined based on the answers to some of the questions. For these requirements, words surrounded by "<" and ">" are mandatory parts of the requirements which must be filled in and words surrounded by "[" and "]" are optional. In some cases, a choice of possible answers are provided separated by the "|" character.

**Management** Finally, a pattern should include any information regarding the source of the pattern, the version, known uses, or any information relating to how the pattern was derived. For this pattern we note that it was adapted from the security design pattern from [72] and modified for requirements based on Withall's work on access control requirements [89].

### 10.2.2 Feature Diagram Hierarchies

Pattern relations are managed with the use of feature diagrams, which connect patterns in places where a question might lead a user to another pattern. The resulting hierarchy partitions a larger problem or security area into smaller problems (i.e. patterns). We believe such a hierarchical structure would reduce the time and difficulty needed to select an appropriate pattern and security requirements for a complex situation. Furthermore, the hierarchy enables users to easily discover related patterns which could be implemented to enhance their system security. Feature diagrams are not necessary for the end use of pattern hierarchies, as apparent in our user study. The diagrams serve as a means for visualizing and managing the structure. When knowledge of the feature diagram notation is not present, users can simply be guided with the pattern questions as long as the hierarchy has been constructed correctly.

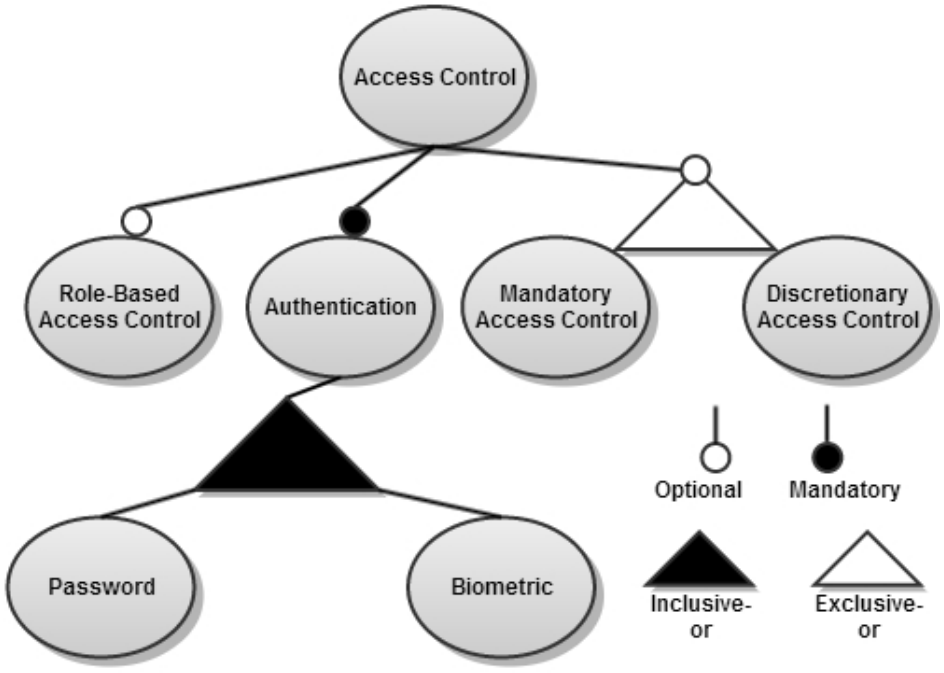



**Figure 10.3**: Feature Diagram Notation

Based on the solution space of the template, a pattern hierarchy can be derived as seen in Figure 10.3. The hierarchical relations between patterns appear within each pattern template as the questions that help the pattern user make decisions on which pattern to include for their situation.

For the Access Control pattern in Section 10.2.1, the question, "how are entities to be authenticated in order to gain access to the system?" is associated with the Authentication pattern. Figure 10.3 shows this connection with a link between the Access Control and Authentication patterns. Our approach is inspired by the Inquiry Cycle model (ICM) which uses inquiry-based discussion between stakeholders in order to revise previous iterations of requirements and introduce relevant patterns [62]. Our approach begins by highlighting existing challenges for the problem (i.e. the forces) and uses questions to elicit security requirements as responses to those challenges.

We continue to use access control as an example to illustrate a hierarchy due to its large amount of related work including various aspects and specialized models [23, 35, 44, 46]. Not only is it a commonly understood aspect of security, but it also includes many facets which allow us to demonstrate the different benefits of using a feature diagram to illustrate the pattern. In Section 10.2.1 we have described a single Access Control pattern from which our hierarchy extends. In this example, the access control pattern is placed as the root of the hierarchy; hence, we name the entire hierarchy after it. In the following subsections, we describe the use of each notation for our hierarchy design using the same example from Figure 10.3.

**Optional**

The notation for optional components is denoted by an unfilled circle on the end of the connection closest to the optional pattern. Such components are directly related to the inquiry-based approach. In Section III-E, our example included the question, "Can users of the system be categorized into roles that will have different access privileges?" which related the Access Control pattern to the Role-Based Access Control pattern. For such an instance where the answer to the question determines whether or not the related pattern is relevant, an optional connection would be made.

**Mandatory**

Mandatory components are denoted by a filled circle on the end of a connection closest to the mandatory pattern. In Figure 10.3, the Access Control pattern includes the Authentication pattern

which is commonly viewed as a necessary component of access control [89] as a mandatory feature. Mandatory components do not stem from "yes" or "no" questions as such questions would imply a non-compulsory relation. Instead, such relations are made through "how" questions. For instance, we asked, "How are actors to be authenticated in order to gain access to the system?" Here, the word "how" poses the question so that it requires an answer. On the contrary, if the question were posed, "Are actors to be authenticated in order to gain access to the system?", one could simply answer "no", thus avoiding the mandatory component. In this example, the phrasing requires that there must be a way for the actors to be authenticated. This question included a reference to the Authentication pattern and is connected within the diagram through the use of a mandatory connection.

**Inclusive Or**

Some questions may require a child pattern to be included, but allow for multiple children to be included simultaneously. Our example includes two forms of credential verification patterns: Password and Biometric. These patterns are included as an Inclusive Or decision since it is acceptable to use multiple credential verification mechanisms and at least one is required. These relations have a filled triangle spanning the edges between the nodes in the decision. Cardinality is represented with brackets in the form [n, m] where n is the lower bound and m is the upper bound in the Inclusive Or relation. For instance, "[2,5]" would indicate that between 2 and 5, inclusive, of the patterns across the relation must be chosen.

**Exclusive Or**

Similarly to the inclusive-or case, the exclusive-or relation allows for decisions between different patterns. However, this relation is used when only one choice may be made. These relations have an unfilled triangle spanning the edges between the nodes in the decision. The Access Control node illustrates the use of an Exclusive Or decision between the MAC and DAC patterns. An unfilled triangle joins their edges to signify that only one can be chosen if this optional branch is selected.

### 10.2.3 User Study

We believe that pattern hierarchies combined with the ICM will improve users' ability to select appropriate patterns and requirements faster and more effectively than unassisted pattern selection. Furthermore, the use of pattern hierarchies should allow novice software engineers with limited experience to benefit from expert knowledge embedded in such patterns and hierarchies.

To evaluate our approach we conducted a study to compare the pattern selection of two groups: one with a pattern hierarchy and one without a pattern hierarchy. The first step was to gather existing security patterns into a repository[3]. This assessment included design, architectural, and requirements patterns so we could better understand how different security concerns were addressed at different development stages among the pattern landscape. Sources included a literature review of textbooks and scientific publications on security patterns. A total of 143 security patterns, 30 of which we classified as security requirement patterns, were collected. We then gathered a set of patterns relating to access control from the repository to create the initial draft of a pattern hierarchy. We mapped these existing patterns to match our pattern template described in Section III in order to build the pattern hierarchy.

Next, we reached out to security experts from the local security community with the intention of revising our hierarchy and preparing for the evaluation of the hierarchy with novice users. Two banking scenarios regarding access control were presented to these experts and used as a means to gather more information and possible patterns for our existing hierarchy example. From transcriptions of expert interviews conducted using the ICM and feedback directly from the experts, we further refined the Access Control hierarchy. For each of the two scenarios we were also able to assemble a subset of patterns in the hierarchy which were most applicable. These subsets were used later in evaluation with novice users.

Finally, we evaluated the Access Control hierarchy comprised of patterns chosen by the experts by observing what patterns novices selected from it. This evaluation consisted of a comparison of novices both with and without the use of the hierarchy so as to test our hypothesis that the hierarchy

[3]http://sefm.cs.utsa.edu/repository/patterns

would improve users' ability over unassisted pattern selection.

### Scenarios

The two scenarios used in our study involved access control in some way due to the common use of access control as an example for security [35]. This allowed for less time to be taken in training since it would be more likely for both experts and novices alike to have some background knowledge in the domain. The scenarios involved a fictional credit union and bank which were partnering with each other. This partnership required new software to be implemented to accommodate shared data. Our description of the entities described the number of employees, their jobs, and features from each of the entities' existing software infrastructures (e.g., instant messaging system and administrative tools).

Scenario 1 described the ATM system currently in place for both institutions and gave the new requirements for the partnership. These requirements outlined the location, hours of operation, and usage fees for customers of each institution. A general set of auditing requirements were provided as well as the stipulation that only a subset of the bank's ATMs would be accessible to credit union customers.

Scenario 2 described a computerized banking system with online banking. The system would need to be implemented only for the credit union, but with possibility for relevant data to be transferred to the bank. Specifications pertaining to user access, financial transaction services, customer services, and browser support were described.

These scenario descriptions were intended to include only high-level descriptions of the institutions' requirements so as to elicit questions and discussion from the experts. This would allow us to infer more about the thought process of the experts interviewed. For situations where clarification on the scenarios was requested, a third-party trained as a domain expert representing the financial institutions was provided. The domain expert explained the different features required for employees based on their roles. For any other questions, we only required that the domain expert be consistent with his responses so as to provide a constant experience for each expert.

### Sampling

We gathered expert participants with experience relating to computer security from academia and industry. We were able to recruit two members of a local security forum in San Antonio. Members of this forum were industry security specialists in the financial, utility, and local government sectors. We also enlisted participation from a member of a local application security consulting firm, a researcher and professor in the area of access control UTSA, a senior security staff member at a multinational computer technology corporation, and two post-doctoral researchers in the areas of privacy and security policies. In total, we interviewed seven industry and academic experts. Our goal was not to provide a complete representation of the security community's opinion, but instead to create an example to test the use of a hierarchy. For this reason, the representation given by our small sampling of security experts would be sufficient.

The second group consisted of university students with computer science backgrounds. We recruited participants in three undergraduate (junior and senior level) and one graduate-level computer science course. For each class, we gave a three minute explanation of what participants would be asked to do for our study and offered a $15 gift card as compensation for time. We were able to recruit 34 participants in this way. Due to the course requirements of the department, these students also had some knowledge of computer security. We chose students as subjects due to their fresh entry-level knowledge in access control with little industry experience. This would allow us to see whether pattern hierarchies are useful in relaying expert experience in order to make better and faster decisions.

### Expert Interviews

We interviewed each expert with an existing hierarchy already created from patterns gathered from textbooks [22, 42, 72, 89] and our pattern repository. Our goal was to see where our hierarchy needed expansion or revision as well as to find a unique subset of the patterns in the hierarchy that would apply to each scenario based on expert opinion. Such patterns subsets would be treated as the gold standard or "correct" set of patterns for evaluating novice performance. Furthermore, by

interviewing experts, we sought to better understand how experts decide what security concepts are relevant to a particular system context. This information is fundamental to refining our pattern hierarchy.

The two banking scenarios were presented to each expert at the beginning of the interviews. We asked the participant to assume the role of a software analyst with the task of creating security requirements for the system. The actor playing the domain expert was introduced and the facilitator explained that he would be available for any questions or clarification that may be needed. The same domain expert was present during each interview to any questions the expert participant may have had.

Interview questions were planned out according to a script (Appendix A.1.1) based on the ICM. We used this model as a means to encourage the expert to ask questions that could be used as the pattern-linking questions fundamental to the hierarchy. For example, when expert participants began to consider an access control policy they would ask the domain expert questions such as, "What kinds of users are there?" or "When do you want users to be able to access the system?". We looked for these kinds of cues to construct a hierarchy using the method described in Section 10.2.2.

After the interview was completed, we asked the participant to go through the existing list of security requirements patterns in the hierarchy and affirm whether or not the pattern should be regarded for the scenario along with an explanation for inclusion or exclusion. Experts were also encouraged to add anything else they thought was relevant. Transcriptions of the interviews were analyzed along with any notes the experts provided in order to create a final Access control hierarchy as seen in Figure10.6. This resulted in the Authorization pattern [89] being placed at the root of the hierarchy above Access Control. We continue to refer to the hierarchy as an Access Control hierarchy due to our scenarios being focused on access control as well as consistency.

A subset of the patterns in the final Access Control hierarchy was also compiled for each scenario based on expert responses to be used in the novice interviews. To do this, we tallied the selections made by the experts resulting in Figures 10.4 and 10.5. If at least five of the seven agreed

on a pattern, we included it in the subset of correct patterns for the scenario. This resulted in 12 of 17 patterns for scenario 1 and 13 of 17 for Scenario 2.

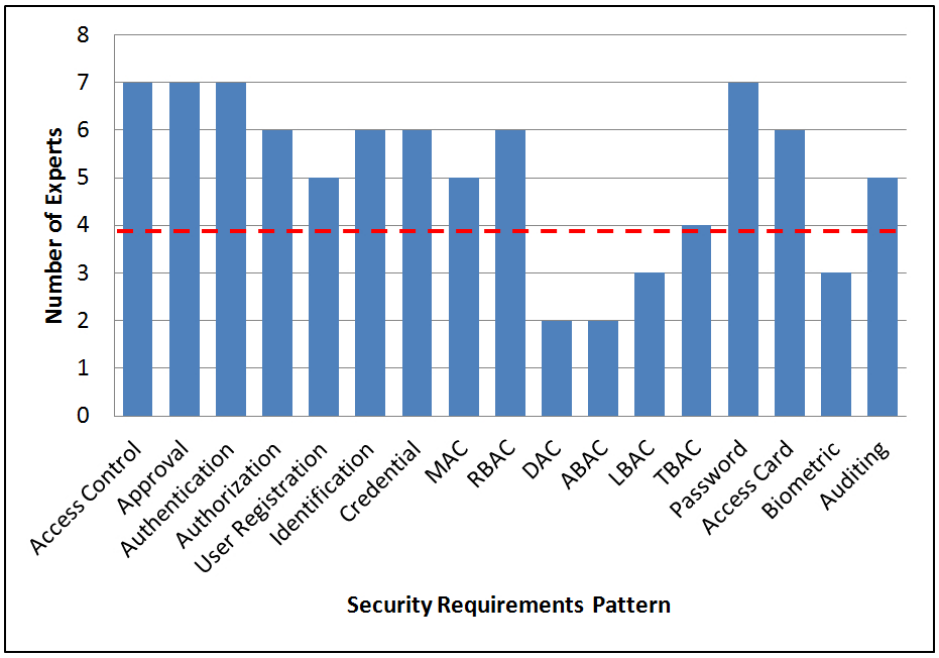



**Figure 10.4**: Scenario 1 Expert Pattern Selection

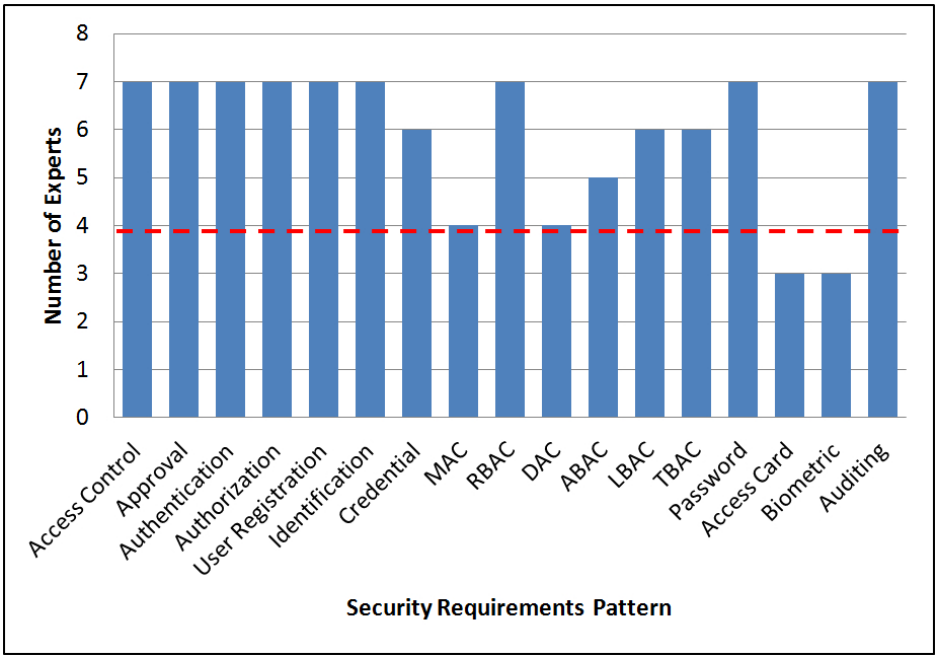



**Figure 10.5**: Scenario 2 Expert Pattern Selection

### Novice Interviews

Novice participants were interviewed individually and randomly placed into either a control or an intervention group. Members of both groups were asked to consider the same two scenarios given to the experts and were provided with the same domain expert present during the expert interviews. Participants for both groups were placed into the role of a software analyst and asked to select the most relevant patterns for each scenario from a list of the patterns from the Access Control hierarchy. Participants were informed that any information they provided would be made anonymous so as to encourage them to proceed naturally. The time to complete both scenarios was also recorded.

For the control group, only the list was given with no relations between the patterns as in the hierarchy. This group was provided with access to the Internet and asked to make the selections to the best of their ability. This gave us a baseline description of how well novices could do in pattern selection on their own.

The intervention group was presented with a questionnaire (Appendix A.1.2) representing the hierarchy. A questionnaire was used in order to forgo any confusion involving training novices in the use of feature diagrams. The questionnaire was created by organizing the pattern hierarchy questions so that they could correspond to the same checklist of patterns given to the control group. Instructions placed after each question both directed the participants to appropriate consecutive questions and had the participant check off appropriate patterns for the scenario based on the hierarchy.

### Threats to Validity

In this section we discuss threats to validity, including external and internal validity [91].

In order to make statistical comparisons between the two groups, we had to provide the control group with the same set of patterns to choose from as the intervention group. In the wild, it would be up to the user to select from all available patterns. By providing the control group with a list, we were forced to provide them with much of the work that would have been done by the hierarchy.

Regarding external validity, we believe that the control group would have performed more poorly in both selection and speed without the provided list. This would actually further validate the use of pattern hierarchies.

We chose not to use the feature diagram representation of the hierarchy for our study due to the time constraints involved for training. This presents a risk to internal validity. Even with the feature diagram notation, the same questions must be answered as in the questionnaire. The feature diagram itself is useful for visualizing the flow between patterns; however the user must be familiar with the notation. We felt that the questions alone were enough for the user to gain the guidance provided by the hierarchy, and a properly-trained user would have only performed better. By providing the feature diagram representation to participants unfamiliar with it, we would have risked user error due to misunderstanding of the notation.

### 10.2.4 Results and Observations

Here, we discuss the knowledge we were able to extract from the experts as well as how our hierarchy affected novice decisions.

#### Expert Knowledge Goes Beyond Patterns

An important contribution of patterns is the reuse of expert knowledge. Our research asks if expert knowledge could go beyond individual patterns and expand to pattern organization and selection through the use of pattern hierarchies. Based on our literature review, interaction with experts, and what we were able to produce with that knowledge, we found that expert knowledge can be applied in a larger scale through our hierarchies.

We used 14 patterns from textbooks and research papers prior to our interaction with experts [72, 89]. From the seven interviews we were able to expand that number to 17 from the additions the experts included to our checklist of patterns. The additions (auditing, time-based access control (TBAC), location-based access control (LBAC), and security question) were not in the form of explicit patterns, but fit into the hierarchy as supplemental security topics (Figure 10.6

related to the existing patterns. We assert that this implies the potential for yet-to-be-created patterns.

The links between patterns which formed the pattern hierarchy were produced by integrating information already present in patterns (e.g., "Related Patterns" sections) and information gained from expert interviews. Generally, as experts described what kind of security requirements were important for the scenarios they would ask the domain expert questions for clarification. For example, when discussing attribute-based access control (ABAC) [85], the ICM urged us to pose the follow-on question: "when should users have access to the system?" Experts would often respond by asking the domain expert if there would be times when employees would not be at work and if they should have access during those times. Depending on the answers to these questions, a need for a corresponding requirements pattern would be necessary. For this example, the domain expert indicated that there are parts of the day when no employees are at work and thus should not be able to access the system. This triggered the expert to begin considering requirements that would be part of a TBAC pattern. The resulting relation between ABAC and TBAC could now be represented with the question: "are there times when users should not have access to the system?" This is represented in Figure 10.6 by the optional connection between these two patterns. The question itself would become part of the solution space of the ABAC pattern.

The experts would often give stipulations for some security requirements. For example, biometrics should only be used with consent of the user. These would be handled similarly. Whether or not the stipulation was viable would be posed in the form of a question and applied to the hierarchy. Furthermore, the question of viability could have implications on the relevant quality attributes to be represented in the Forces portion of the resulting pattern. For this example, usability would be a contributing factor.

**Pattern Hierarchies are Efficient and Usable**

Novice results were measured by correct pattern choice and time to completion. In both cases, the intervention group performed better than the control group.

| | Average Success Rate (%) | | Increase | Standard Deviation (s) | | Sample Size (n) | | Significance | Effect Size |
|---|---|---|---|---|---|---|---|---|---|
| | Control ($\bar{X}_C$) | Intervention ($\bar{X}_I$) | ($\bar{X}_I$ - $\bar{X}_C$) | Control | Intervention | Control | Intervention | (p-value) | (Cohen's d) |
| Scenario 1 | 60.78 | 82.92 | 22.13 | 14.70 | 12.41 | 17 | 16 | 0.00003 | 1.63 |
| Scenario 2 | 72.08 | 82.75 | 10.66 | 11.21 | 7.09 | 16 | 17 | 0.002 | 1.16 |

**Table 10.1**: Statistical Results

Regarding correct pattern choice, novices were graded depending on if they made the same decision to include or exclude a particular pattern as defined by the gold standard pattern sets derived from expert decisions described in Section V-C. For example, Scenario 1 DAC was not included by experts, so if a novice user chose to include the pattern, then it was marked as incorrect. Points were not deducted for incorrect answers. A majority of the experts expressed the need for two-factor authentication [24] for both scenarios. To accommodate this, a correct choice of authentication patterns (password, biometric, access card, and security question) consisted of any two. If a participant chose only one of those patterns, they were not awarded points. This selection scheme is apparent in Figure 10.6 with the range of necessary patterns denoted within the exclusive-or arc with "[2]".

Before computing the results of our study, we removed outliers by using Iglewicz and Hoaglin's modified Z-score with median absolute deviation method ($MAD$) [37]. The $MAD$ for each group was calculated with:

$$MAD = median(|x_i - \tilde{x}|) \tag{10.1}$$

Modified Z-scores ($M_i$) for each score were calculated with:

$$M_i = \frac{0.6755 * (x_i - \tilde{x})}{MAD} \tag{10.2}$$

Where $x_i$ represents individual scores and $\tilde{x}$ is the median. 0.6755 is the multiplier used for outlier detection recommended by Iglewicz and Hoaglin for samples of our size. For this method, values of $M_i$ greater than $(3.5 * MAD)$ were removed as statistical outliers. This resulted in the removal of two participants.

Table 10.1 describes the quantitative results of the study regarding successful pattern choice by

novice users. For both Scenario 1 and Scenario 2 novices were more successful at making the same choices as experts when using the pattern hierarchy with a success rate increase over the control group of 22.13% and 10.66% respectively. Regarding the lower increase for Scenario 2, we found that fewer users asked the domain expert questions for that scenario. We took this as an implication of misappropriated familiarity with the situation. Unless a user has sufficient experience with the situation, it is necessary for them to ask questions in order to properly specify requirements.

We used a standard t-test to measure the significance of our results. We tested the null hypothesis ($H_0$) that the mean population success rates for users without the hierarchy ($\mu_C$) and users with the hierarchy ($\mu_I$) were equal:

$$H_o : \mu_c - \mu_I = 0 \tag{10.3}$$

The p-values produced by the t-tests indicate that the null hypothesis could be rejected for both scenarios and that the results of our study were statistically significant.

As an indicator of the strength of our sample size, we calculated effect size using Cohen's d [21]. This was done by taking the difference between the mean sample success rate for the control group ($\bar{x}_C$) and the mean sample success rate for the intervention group ($\bar{x}_I$) divided by the average standard deviation for both groups, $s_{IC}$:

$$d = \frac{\bar{x}_I - \bar{x}_C}{s_{IC}} \tag{10.4}$$

Based on Cohen's conventions, the values for both Scenario 1 and Scenario 2 indicated a large effect size. While the p-values explained in the previous paragraph give us indications of whether or not our null hypothesis can be rejected, and thus our hypothesis be accepted, effect size is an indicator of whether the result of our experiment is meaningful regardless of our sample size. Based on Cohen's conventions [21], the values for both Scenario 1 and Scenario 2 indicated a large effect size. This implied meaningful results regardless of our sample size.

Efficiency in terms of completion time was recorded as a total for both groups due to the design of our study. Table 10.2 shows an average decrease in selection time of more than 80% for

| Average Completion Time (minutes) | | Decrease | Speedup | Standard Deviation (s) | | Sample Size (n) | | Significance | Effect Size |
|---|---|---|---|---|---|---|---|---|---|
| Control ($\bar{x}_C$) | Intervention ($\bar{x}_I$) | ($\bar{x}_C$ - $\bar{x}_I$) | | Control | Intervention | Control | Intervention | (p-value) | (Cohen's d) |
| 28.56 | 15.82 | 12.74 | 1.81 | 11.85 | 4.95 | 16 | 17 | 0.00037 | 1.52 |

**Table 10.2**: Pattern Selection Efficiency

the group using the pattern hierarchy. We attribute this increase in selection speed to the guidance the hierarchy provided. Without the hierarchy, the control group was forced to use their previous knowledge and the Internet to make decisions. Participants were able to make decisions faster while covering the same amount of patterns with the aid of the straightforward questions provided by the hierarchy. We calculated statistical significance and effect size with the same methods used for success rates. Both values implied statistical significance and meaningful results.

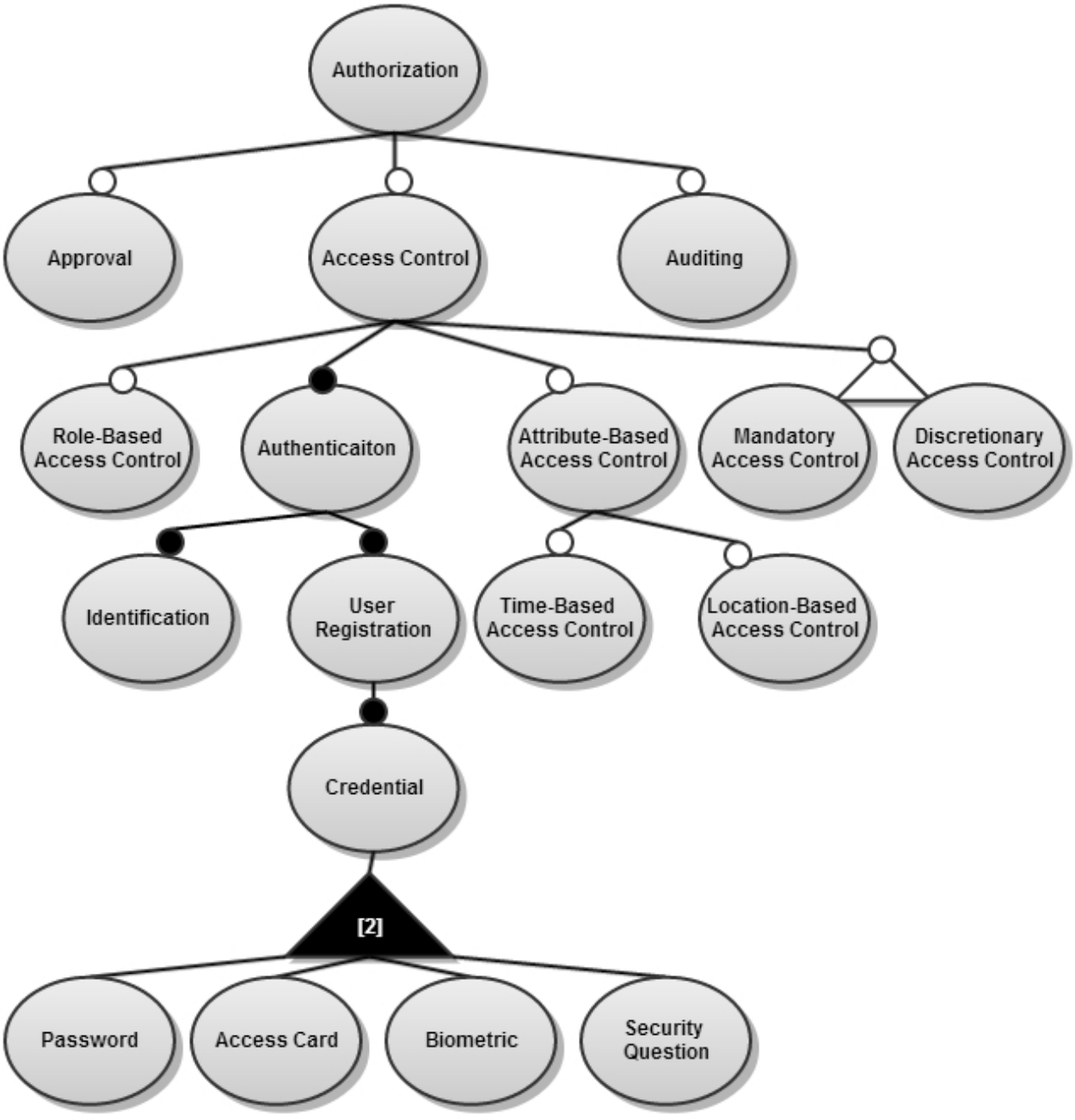



**Figure 10.6**: Access Control Requirements Hierarchy

# CHAPTER 11: CONCLUSION

Privacy policy alignment is an important topic not only for end users, but software developers, auditors, and policy writers. The misrepresentation of an app's data collection with regard to privacy can result in negative consequences in the form of data leaks, violation of regulations, and possible fines or audits. For a practical approach to aligning privacy policies with their corresponding application code, it is necessary to cover many aspects of the application's life-cycle. By providing methods for the analysis of source code, byte code, and natural language privacy policies both during development and deployment, the necessary tools can be made for aligning code and policy as well as the detection of misalignments.

Moving forward, this framework establishes a grounds for much future work. First, contextual polarity can be tailored to the domain of mobile application privacy by annotating phrases extracted from real-world privacy policies with sentiment labels. Such a corpus would further hone the precision of misalignment detection. Second, mappings from policy phrases to constructs other than API methods can be explored. The problem of policy misalignment is not limited to sensors and platform-level sources. Other sources and sinks such as those provided by third party APIs can contribute to the misalignment problem. If the approach in this work can be adapted to fit such elements, privacy policy confidence can be further improved. Finally, the efficacy of this methodology in other other domains (e.g., Internet of Things, web applications, etc) as well as privacy regulations (e.g., HIPAA) can be assessed.

The approach in this work is founded upon a relationship between natural language policy and application code in an effort to improve privacy policy trust. I have introduced a mapping upon which a framework is built that is both scalable and verifiable. Furthermore, I have introduced the POLIDROID tool suite which exemplifies the practical capability of the framework with static and dynamic analysis tools to aid stakeholders in policy creation and alignment. To improve accessibility, and thus potential impact, these tools have been made available as web- and IDE-based applications. Efficacy is further validated through empirical studies over actual Android devel-

opers. Through these contributions, I have provided a strong basis for improving confidence in privacy policies for end users and developers alike.

# APPENDIX A: SCRIPTS AND QUESTIONNAIRES

## A.1 Requirements Pattern Study

### A.1.1 Expert Script

The following script was used in expert interviews to begin discussion. Questions were based on the ICM and, in many cases, clarification was required through the form of follow-on questions. Examples of such follow-on questions are included.

(Who) Who should have access to the system? (Clarification and follow up may be required) Ex:

- Who are the users?
- Are there any other users that may need access to the system?

(What-kinds-of) What information should be shared between institutions? (Clarification and follow up may be required) Ex:

- Can you think of any other types of information?

(What-kinds-of)

What information should we protect to enforce information confidentiality and integrity? (Clarification and follow up may be required)

(What-kinds-of)

What access information shall be kept to provide accountability? (clarification and follow up may be required)

(How-to)

How do you authenticate? (Clarification and follow up may be required)

Ex:

[if password] ->(Follow-on/What-kinds-of)

What kind of password shall we require?

[if biometric] -> (Follow-on/What-kinds-of)

What kinds of biometric scans should be used? (fingerprint, face, iris, hand, etc.)

[if access card] -> (Follow-on/What-kinds-of)

What kind of access card should be used? (active/passive, magnetic/radio)

(What-kinds-of)

What kind of threats do we want to prevent?

(Clarification and follow up may be required)

(When)

Are there any time constraints on access within this system? (Clarification and follow up may be required)

(What-if)

If an employee gets moved to a different position what needs to be considered? (Clarification and follow up may be required)

(How-to)

Describe the organization of an access control approach for this type of scenario? (Clarification and follow up may be required)

(Follow-on)

Why would other approaches not be suitable for this scenario? (Clarification and follow up may be required)

(How-to)

Which access control policies and enforcement techniques should be considered for this scenario? (Clarification and follow up may be required)

(Follow-on) Are there any benefits of choosing a different technique? (Clarification and follow up may be required)

-SCENARIO SPECIFIC QUESTIONS-

*SCENARIO 1 QUESTIONS*

(What-if)

What if someone loses his or her PIN? (Clarification and follow up may be required)

(What-if)

What if many invalid PINs have been attempted? (Clarification and follow up may be required)

(How-to)

What mechanisms should be employed to ensure the security of the system? (Clarification and follow up may be required)

(Follow-on) How does this affect the usability and availability of the system? (Clarification and follow up may be required)

*SCENARIO 2 QUESTIONS*

(What-if)

What if someone loses his or her username or password? (Clarification and follow up may be required)

(What-if)

What if many invalid username/password combinations have been attempted? (Clarification and follow up may be required)

(How-to)

What mechanisms should be employed to ensure the security of the system? (Clarification and follow up may be required)

(Follow-on)

How does this affect the usability and availability of the system? (Clarification and follow up may be required)

### A.1.2 Novice Questionnaire

The following questionnaire was used for the novice user portion of the study in lieu of a graphical hierarchy:

1. Authorization

   (a) Are there parts of the system where permission to gain access to a resource cannot be automated (i.e., an actual person must be consulted for the authorization)? *If yes, check "Approval".*

   (b) Will access to the system or resources need to be restrictive on a selective basis (example: only members have access to member data)? *If yes, check "Access Control".*

   (c) *Check "Authorization" and "Auditing".*

2. Access Control

   *If "Access Control" is not checked, proceed to question 3.*

   (a) *Check "Authentication".*

   (b) Can users of the system be categorized into roles that will have different access privileges? A role can be described as job function or title which defines an authority level. (examples of roles: student, professor, department chair, administrator, guest etc)? *If yes, check "Role Based Access Control".*

   (c) Must user, session, or resource attributes be used to restrict or grant access to resources? An attribute can be described as a quality, trait, or characteristic. (examples: time, location, user age, desired action) *If yes, check "Attribute Based Access Control".*

   (d) Choose at most one (either none or 1) of the following (2.d.i or 2.d.ii)

      i. Can resources be assigned labels by the system so that users are given clearance to access resources based on levels of clearance (i.e., resource owners cannot change the clearance needed to access the resource)? *If yes, check "Mandatory Access Control".*

ii. Can access to resources be regulated by the owner of the resource? *If yes, check "Discretionary Access Control".*

3. Authentication

   *If "Authentication" is not checked, proceed to question 4.*

   (a) Check the following three: "Identification", "User Registration", and "Credential".

4. Credential

   If "Credential" is not checked, proceed to question 5.

   (a) Choose any two of the following (4.a.i through 4.a.iv)

      i. Do the majority of users NOT have sufficient knowledge to correctly implement and use certificate-based or public key authentication techniques? *If yes, check "Password".*

      ii. Are the following two statements true: (1) Is usability (e.g., speed, ease of remembering) a large factor? (2) Will access be limited to locations where specific hardware can be installed? *If yes, check "Access Card".*

      iii. Are the following two statements true: (1) Does security outweigh the cost of implementing an authentication mechanism? (2) Will access be limited to locations where specific hardware can be installed? *If yes, check "Biometric".*

      iv. *If only one of the above three can be chosen, check "Security Question".*

5. Attribute Based Access Control

   If "Attribute Based Access Control" is not checked, this concludes the questionnaire.

   (a) Should access to resources be restricted based on time of access (e.g., time between accessing, time of access, total time accessed)? *If yes, check "Time Based Access Control".*

(b) Should access to resources be restricted based on location accessed from? *If yes, Check "Location Based Access Control".*

## A.2 PoliDroid-AS Followup Study

### A.2.1 Participant Questionnaire

1. In this study:

   - You are required to have an Android application developed by you or by a team with you as a member. You need to have permission to use the application for our study.
   - We will not store any information about your application, nor will we store the application code, we will only store your answers to our survey. Survey responses will be anonymized and we will not attribute any survey responses to individual participants.
   - You will be provided with a flash drive that contains a plugin that you will install into Android Studio. The flash drive is for you to keep after the study. Also, we will provide you with $20 Amazon gift card after completing the survey.

2. Current Project

   For this study, please choose one of the Android apps that you developed as a solo developer, or as a member of a team.

   (a) How many people were involved in the development of your chosen app, including yourself (e.g., answer 1 if you were the solo-developer)?

   (b) How familiar are you with the chosen app's *entire* codebase?

   - Extremely familiar
   - Very familiar
   - Moderately familiar
   - Slightly familiar

| **Questions** |
|---|
| Is the data type necessary for your app's basic functionality (that is, within the reasonably expected context of the app's functions as described to users)? |
| Is the data type necessary for business reasons (such as billing)? How will you use the data? |
| Will it be necessary to store data off the device, on your servers? |
| How long will you need to store the data on your servers? |
| Will you share the data with third parties (such as ad networks, analytics companies, service providers)? If so, with whom will you share it? |
| How will third parties use the data? |
| Who in your organization will have access to user data? |
| What parts of the mobile device do you have permissions to access? |
| Can you provide users with the ability to modify permissions? |

- Not familiar at all

(c) Is your application source code available from an online repository such as GitHub?

- Yes
- No

3. Privacy Statement Generation

A privacy policy is a statement or a legal document that describes how a party collects, uses, discloses, and manages personal data. It may also fulfill a legal requirement to protect a customer or client's privacy. In this section, we provide you with some guidelines about how to create privacy statements for the application that you have considered in section 2.

(a) As a first step, list all the personally identifiable data that your application potentially collects, uses, retains, or transfers. Each text box should be used only for one data type.

- 20 textboxes labeled "Data Type"

(b) In this step, for each data type that you listed in step 3a, write policy statements that describe the data practices. You can use the following questions as your guideline to create policy statements for each data type.

(c) In this step, we ask you to select the data types that are potentially collected, used, retained, or transferred through your application. This list is provided by California

| **Policy Statements Example for "contact information"** |
| --- |
| "We collect contact information when you access our services. We use your contact information to communicate with you as a user and administrators regrading the use of services, personalize your experience as part of our provision of services, and conduct data and system analytics. We may disclose your contact information to third-party service providers to perform analytics. Users are able to modify their contact information through the form provided in the application." |

Department of Justice in a document called Privacy On the Go, recommendations for the mobile ecosystem. You can select more than one data type.

- Unique device identifier
- Geo-location (GPS, WiFi, user-entered)
- Mobile phone number
- Email address
- User's name
- Text messages or email
- Call logs
- Contacts/address book
- Financial and payment information
- Health and medical information
- Photos or videos
- Web browsing history
- Apps downloaded or used

(d) Describe any challenges that you encounter when writing the privacy policy in few sentences.

(e) While listing the data types that your application collects, uses, or discloses, how many times did you check your code to find whether you have used an API that returns a private data type?

- 0

- 1-2
- 3-5
- more than 5

4. PoliDroid Privacy Specifications

   In this step, you install an Android Studio plugin, called PoliDroid, that is used to advise developers about privacy. The plugin is provided to you by the flash drive. After a short tutorial using the plugin, we will ask you to take a brief survey. PoliDroid provides users with suggested privacy statements based on the API calls in the application code. It is also used to identify the mismatches between privacy policies and mobile apps. The following steps will guide you through the process.

   (a) Open your application codebase in Android Studio and install the plugin.

   (b) After installing the plugin, click on PoliDroid tab on the menu and select *privacy specification generation* component.

   (c) The plugin will provide you some privacy specifications based on your code. Review the specifications and copy them to the text box below labeled as "Privacy Specifications Created by PoliDroid".

   [text area input]

5. Mismatches between application code and privacy policy

   In this section, use PoliDroid to identify the mismatches between the privacy statements that you created in section 3 and the application code. For this reason, follow the steps below:

   (a) Create a .txt file and copy the policy statements shown in the text box below in the .txt file and save the file.

   (b) Upload the mappings file and the ontology file provided to you on the flash drive along with the .txt file from the previous step which contains your privacy statements.

(c) Based on the results and following definitions, please answer the following questions.

**Definitions:**

- **Strong Misalignment** occurs when the application code is calling an API that returns a data type that is not described in a privacy specification. For example, a strong mismatch occurs when calling `android.view.KeyEvent.getDeviceId()`, which returns the "device ID", but the "device ID" or "device information" is not mentioned in the privacy specifications.
- **Weak Misalignment** occurs when a privacy specification refers to a more abstract data type that what the API returns. For example, referring to "device information" in the specification instead of "device ID" which is returned by the call to `android.view.KeyEvent.getDeviceId()`.

(a) Are the mismatches identified by the plugin valid? If not, please copy the part of privacy policy that covers the data practice for the identified mismatch in the following box.

[text area input]

(b) For valid mismatches identified by the plugin, to the best of your knowledge, why does the privacy policy not describe this data practice?

[text area input]

6. Background and Demographic Questions

(a) What is your age group?

- 18 - 24
- 25 - 34
- 35 - 44
- 45 - 60

- above 60

(b) What is your gender?

- Male
- Female
- Prefer not to answer

(c) How many years of experience do you have developing for the Android platform?

[text input]

(d) What is the highest level of education that you have completed?

- Some high school
- High school diploma or GED
- Some college
- Associate degree from two-year college
- Baccalaureate degree from four-year college
- Some graduate school
- Masters or other professional degree
- Doctorate degree

(e) Are you a student?

- Yes, undergraduate student
- Yes, graduate student
- No

(f) If you are employed in industry, what is your job title?

[text input]

(g) If you are employed in industry, how many years of professional experience do you have?

[text input]

(h) Have you ever published an Android app in an app store?

- Yes, as a solo developer
- Yes, as part of a group or company
- No

(i) Do you have a privacy policy for your application?

- Yes
- No

## BIBLIOGRAPHY


[1] FTC report on Credit Karma and Fandango. https://www.ftc.gov/news-events/press-releases/2014/03/fandango-credit-karma-settle-ftc-charges-they-deceived-consumers, 2014.

[2] FTC report on Snapchat. https://www.ftc.gov/news-events/press-releases/2014/06/ftc-testifies-geolocation-privacy, 2014.

[3] Developer economics q1 2015: State of the developer nation. https://www.developereconomics.com/reports/developer-economics-q1-2015/, 2015.

[4] Smartphone os market share, q1 2015. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2015.

[5] PUBLIC LAW 104-191. *Health Insurance Portability and Accountability Act of 1996*. 104th Congress, 1996.

[6] PUBLIC LAW 106-102. *The Gramm-Leach-Bliley Act of 1999*. 106th Congress, 1999.

[7] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Joaquim Romaguera i Ramió, Max Jacobson, and Ingrid Fiksdahl-King. *A pattern language*. Gustavo Gili, 1977.

[8] James P Anderson. Information security in a multi-user computer environment. *Advances in Computers*, 12:1–36, 1972.

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick 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*, pages 259–269, 2014.

[10] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[11] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *SP*, pages 184–198, 2006.

[12] Kent Beck, Ron Crocker, Gerard Meszaros, James O Coplien, Lutz Dominick, Frances Paulisch, and John Vlissides. Industrial experience with design patterns. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 103–114. IEEE, 1996.

[13] Emmanuel Bello-Ogunu and Mohamed Shehab. Permitme: integrating Android permissioning support in the ide. In *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*, pages 15–20. ACM, 2014.

[14] Jaspreet Bhatia and Travis Breaux. Towards an information type lexicon for privacy policies. In *8th IEEE International Workshop on Requirements Engineering and Law (RELAW)*, pages 19–24, 2015.

[15] Barry W Boehm. Software engineering economics. *IEEE transactions on Software Engineering*, (1):4–21, 1984.

[16] Travis Breaux and Florian Schaub. Scaling requirements extraction to the crowd: Experiments on privacy policies. In *22nd IEEE International Requirements Engineering Conference (RE'14)*, pages 163–172, 2014.

[17] Travis D Breaux, Hanan Hibshi, and Ashwini Rao. Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements. *Requirements Engineering*, 19(3):281–307, 2014.

[18] J Brinkkemper and A Solvberg. Tropos: A framework for requirements-driven software development. *Info. Syst. Engg.: State Art Res. Themes, Lecture Notes Comput. Sci*, 1:261–273, 2000.

[19] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: a new symbolic model checker. *Int. Journal on Software Tools for Technology Transfer*, 2:410–425, 2000.

[20] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[21] Jacob Cohen. Statistical power analysis. *Current directions in psychological science*, 1(3):98–101, 1992.

[22] PCI Security Standards Council. Payment card industry data security standard, 2016. version 3.2.

[23] Robert Crook, Darrel Ince, and Bashar Nuseibeh. On modelling access policies: Relating roles to their organisational context. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 157–166. IEEE, 2005.

[24] Manik Lal Das. Two-factor user authentication in wireless sensor networks. *IEEE Transactions on Wireless Communications*, 8(3):1086–1090, 2009.

[25] Henry DeYoung, Deepak Garg, Limin Jia, Dilsun Kaynar, and Anupam Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In *WPES*, pages 73–82, 2010.

[26] David Dietrich and Joanne M Atlee. A pattern for structuring the behavioural requirements of features of an embedded system. In *Requirements Patterns (RePa), 2013 IEEE Third International Workshop on*, pages 1–7. IEEE, 2013.

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, 2010.

[28] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994.

[30] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 151–162. ACM, 2011.

[31] Jonathan Godfrey and Courtney Bernard. State of the app economy 2014. 2014.

[32] Michael Grüninger and Mark S Fox. Methodology for the design and evaluation of ontologies. 1995.

[33] Kamala D Harris. *Privacy on the Go: Recommendations for the Mobile Ecosystem*. 2013.

[34] Frank Annunzio Henry Reuss. Right to financial privacy act, 1978. Public Law 95-630.

[35] Hanan Hibshi, Rocky Slavin, Jianwei Niu, and Travis D Breaux. Rethinking security requirements in re research. *Technical Report, Tech. Rep. Report CSTR-2014-001*, 2014.

[36] Mitra Bokaei Hosseini, Sudarshan Wadkar, Travis D Breaux, and Jianwei Niu. Lexical similarity of information type hypernyms, meronyms and synonyms in privacy policies. In *2016 AAAI Fall Symposium Series*, 2016.

[37] Boris Iglewicz and David Caster Hoaglin. *How to detect and handle outliers*, volume 16. Asq Press, 1993.

[38] Michael Jackson. *Problem frames: analysing and structuring software development problems*. Addison-Wesley, 2001.

[39] Masita Abdul Jalil and Shahrul Azman Mohd Noah. The difficulties of using design patterns among novices: An exploratory study. In *Computational Science and its Applications, 2007. ICCSA 2007. International Conference on*, pages 97–103. IEEE, 2007.

[40] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. Privacy as part of the app decision-making process. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3393–3402. ACM, 2013.

[41] Darrell M Kienzle, Matthew C Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository version 1.0. *DARPA, Washington DC*, 2002.

[42] Dae-Kyoo Kim, Pooja Mehta, and Priya Gokhale. Describing access control models as design patterns using roles. In *Proceedings of the 2006 conference on Pattern languages of programs*, page 11. ACM, 2006.

[43] Moo Nam Ko, Gorrell P Cheek, Mohamed Shehab, and Ravi Sandhu. Social-networks connect services. *Computer*, 43(8):37–43, 2010.

[44] Manuel Koch and Francesco Parisi-Presicce. Formal access control analysis in the software development process. In *Proceedings of the 2003 ACM workshop on Formal methods in security engineering*, pages 67–76. ACM, 2003.

[45] Sascha Konrad and Betty HC Cheng. Requirements patterns for embedded systems. In *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, pages 127–136. IEEE, 2002.

[46] Vivek Krishnan, Mahesh V Tripunitara, Kinson Chik, and Tony Bergstrom. Relating declarative semantics and usability in access control. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 14. ACM, 2012.

[47] Peifung E. Lam, John C. Mitchell, and Sharada Sundaram. A formalization of HIPAA for a medical messaging system. In *TrustBus*, pages 73–85, 2009.

[48] Soren Lauesen. *Software requirements: styles and techniques*. Pearson Education, 2002.

[49] Manshan Lin and Heqing Guo. Present situation and development of single sign-on tehnology. *Journal of Computer Applications*, 24(6):248–250, 2001.

[50] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[51] Shinichi Matsumoto and Kouichi Sakurai. A proposal for the privacy leakage verification tool for Android application developers. In *Proceedings of the 7th International Conference on Ubiquitous Information Management and Communication*, page 54. ACM, 2013.

[52] Gary McGraw. Testing for security during development: Why we should scrap penetrate-and-patch. *IEEE aerospace and electronic systems magazine*, 13(4):13–15, 1998.

[53] Mark A Musen. The Protégé project: a look back and a look forward. *AI matters*, 1(4):4–12, 2015.

[54] Tetsuji Nakagawa, Kentaro Inui, and Sadao Kurohashi. Dependency tree-based sentiment classification using crfs with hidden variables. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 786–794. Association for Computational Linguistics, 2010.

[55] Object Management Group. Unified Modelling Language (Superstructure), v2.3, 2010. Internet: www.omg.org.

[56] Cristina Palomares, Carme Quer, Xavier Franch, Cindy Guerlain, and Samuel Renault. A catalogue of non-technical requirement patterns. In *Requirements Patterns (RePa), 2012 IEEE Second International Workshop on*, pages 1–6. IEEE, 2012.

[57] Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 115–124. Association for Computational Linguistics, 2005.

[58] Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and trends in information retrieval*, 2(1-2):1–135, 2008.

[59] Symeon Papadopoulos and Adrian Popescu. Privacy awareness and user empowerment in online social networking settings. `http://www.computer.org/web/computingnow/archive/january2015`, 2015.

[60] Gabriele Petronella. *Analyzing Privacy of Android Apps*. PhD thesis, Politecnico di Milano, 2014.

[61] Livia Polanyi and Annie Zaenen. Contextual valence shifters. In *Computing attitude and affect in text: Theory and applications*, pages 1–10. Springer, 2006.

[62] Colin Potts, Kenji Takahashi, and Annie I Antón. Inquiry-based requirements analysis. *IEEE software*, 11(2):21–32, 1994.

[63] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing Android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[64] Joel R. Reidenberg, Jaspreet Bhatia, Travis D. Breaux, and Thomas B. Norton. Automated comparisons of ambiguity in privacy policies and the impact of regulation. *Journal of Legal Studies*, 2016.

[65] Joel R Reidenberg, Travis Breaux, Lorrie Faith Cranor, Brian French, Amanda Grannis, James T Graves, Fei Liu, Aleecia M McDonald, Thomas B Norton, Rohan Ramanath, et al. Disagreeable privacy policies: Mismatches between meaning and users' understanding. 2014.

[66] Vassiliki Rentoumi, Stefanos Petrakis, Manfred Klenner, George A Vouros, and Vangelis Karkaletsis. United we stand: Improving sentiment analysis by joining machine learning and rule based methods. In *LREC*, 2010.

[67] IBM research. Enterprise privacy authorization language (EPAL) version 1.2, nov. 2003. http://www.zurich.ibm.com/pri/projects/epal.html.

[68] Atanas Rountev and Dacong Yan. Static reference analysis for gui objects in Android software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 143:143–143:153, New York, NY, USA, 2014. ACM.

[69] Mark Rowan and Josh Dehlinger. Encouraging privacy by design concepts with privacy policy auto-generation in eclipse (page). In *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*, pages 9–14. ACM, 2014.

[70] Johnny Saldana. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2012.

[71] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[72] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[73] Hui Shen, Ram Krishnan, Rocky Slavin, and Jianwei Niu. Sequence Diagram aided security policy specification. Technical Report CS-TR-2014-005, UTSA, 2014.

[74] Hui Shen, Ram Krishnan, Rocky Slavin, and Jianwei Niu. Sequence diagram aided privacy policy specification. *IEEE Transactions on Dependable and Secure Computing*, 13(3):381–393, 2016.

[75] Hui Shen, Mark Robinson, and Jianwei Niu. Formal analysis of Sequence Diagram with Combined Fragments. In *International Conference on Software Paradigm Trends*, pages 44–54, July 2012.

[76] Rocky Slavin, Jean-Michel Lehker, Jianwei Niu, and Travis D Breaux. Managing security requirements patterns using feature diagram hierarchies. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 193–202. IEEE, 2014.

[77] Rocky Slavin, Hui Shen, and Jianwei Niu. Characterizations and boundaries of security requirements patterns. In *Requirements Patterns (RePa), 2012 IEEE Second International Workshop on*, pages 48–53. IEEE, 2012.

[78] Rocky Slavin, Xiaoyin Wang, Mitra Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D. Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in Android application code. In *38th International Conference on Software Engineering (ICSE)*. ACM/IEEE, 2016.

[79] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. PoliDroid-AS: A privacy policy alignment plugin for Android studio. Technical Report CS-TR-2017-002, UTSA, 2016.

[80] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. PVDetector: a detector of privacy-policy violations for Android apps. In *Proceedings of the International Workshop on Mobile Software Engineering and Systems*, pages 299–300. ACM, 2016.

[81] Benjamin Snyder and Regina Barzilay. Multiple aspect ranking using the good grief algorithm. In *HLT-NAACL*, pages 300–307, 2007.

[82] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality

over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer, 2013.

[83] Statista. Number of available applications in the Google Play Store from December 2009 to September 2016. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2016.

[84] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android permission creep. In *Proceedings of the Web*, volume 2, 2011.

[85] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 45–55. ACM, 2004.

[86] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. GUILeak: Identifying privacy practices on gui-based data. Technical Report CS-TR-2017-004, 2017.

[87] Tom Warren. Google touts 1 billion active Android users per month. http://www.theverge.com/2014/6/25/5841924/google-android-users-1-billion-stats/, 2014.

[88] Michael Weiss and Haralambos Mouratidis. Selecting security patterns that fulfill security requirements. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 169–172. IEEE, 2008.

[89] Stephen Withall. *Software requirement patterns*. Pearson Education, 2007.

[90] Robert K Yin. Case study research: Design and methods. sage publications. *Thousand oaks*, 2009.

[91] Robert K Yin. *Applications of case study research*. Sage, 2011.

[92] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. *Progress in informatics*, 5(5):35–47, 2008.

[93] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. Does organizing security patterns focus architectural choices? In *Proceedings of the 34th International Conference on Software Engineering*, pages 617–627. IEEE Press, 2012.

[94] Leah Xinya Zhao. Privacy sensitive resource access monitoring for Android systems. Master's thesis, Rochester Institute of Technology, 2014.

# VITA

Rocky Slavin was born in San Antonio, Texas and has spent most of his life in south Texas. He graduated from the University of Texas at San Antonio in 2012 with a B.S. in Computer Science with a concentration in Computer and Information Security and immediately continued his research in the areas of Software Engineering and Cyber Security as a Ph.D. student. Rocky has taught Software Engineering courses beginning in 2017 and looks forward to a future in academia both as a teacher and researcher.

Rocky and his lovely wife Sarah have been married for 11 years and have a wonderful daughter named Elene, age two.