ABSTRACT

Title of Dissertation: CONTEXT-DEPENDENT PRIVACY AND SECURITY MANAGEMENT ON MOBILE DEVICES

Prajit Kumar Das
Doctor of Philosophy, 2017

Dissertation directed by: Prof. Anupam Joshi and Prof. Tim Finin
Department of Computer Science and Electrical Engineering

There are ongoing security and privacy concerns regarding mobile platforms which are being used by a growing number of citizens. Security and privacy models typically used by mobile platforms use one-time permission acquisition mechanisms. However, modifying access rights after initial authorization in mobile systems is often too tedious and complicated for users. User studies show that a typical user does not understand permissions requested by applications or are too eager to use the applications to care to understand the permission implications. For example, the Brightest Flashlight application was reported to have logged precise locations and unique user identifiers, which have nothing to do with a flashlight applications intended functionality, but more than 50 million users used a version of this application which would have forced them to allow this permission. Given the penetration of mobile devices into our lives, a fine-grained context-dependent security and privacy control approach needs to be created.

We have created MITHRIL as an end-to-end mobile access control framework

that allows us to capture access control needs for specific users, by observing violations of known policies. The framework studies mobile application executables to better inform users of the risks associated with using certain applications. The policy capture process involves an iterative user feedback process that captures policy modifications required to mediate observed violations. Precision of policy is used to determine convergence of the policy capture process. Policy rules in the system are written using Semantic Web technologies and the Platys ontology to define a hierarchical notion of context. Policy rule antecedents are comprised of context elements derived using the Platys ontology employing a query engine, an inference mechanism and mobile sensors. We performed a user study that proves the feasibility of using our violation driven policy capture process to gather user-specific policy modifications.

We contribute to the static and dynamic study of mobile applications by defining application behavior as a possible way of understanding mobile applications and creating access control policies for them. Our user study also shows that unlike our behavior-based policy, a deny by default mechanism hampers usability of access control systems. We also show that inclusion of crowd-sourced policies leads to further reduction in user burden and need for engagement while capturing context-based access control policy. We enrich knowledge about mobile application behavior and expose this knowledge through the Mobipedia knowledge-base. We also extend context synthesis for semantic presence detection on mobile devices by combining Bluetooth, low energy beacons and Nearby Messaging services from Google.

CONTEXT-DEPENDENT PRIVACY AND SECURITY MANAGEMENT ON MOBILE DEVICES

by

Prajit Kumar Das

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Anupam Joshi, PhD, Chair
Tim Finin, PhD, Co-Chair
Tim Oates, PhD
Nilanjan Banerjee, PhD
Arkady Zaslavsky, PhD
Dipanjan Chakraborty, PhD

© Copyright by
Prajit Kumar Das
2017

APPROVAL SHEET

Title of Dissertation: CONTEXT-DEPENDENT PRIVACY AND SECURITY MANAGEMENT ON MOBILE DEVICES

Name of Candidate: Prajit Kumar Das
Computer Science, 2017

Dissertation and Abstract Approved: _____

Anupam Joshi, PhD
Professor, Chair
Department of Computer Science and
Electrical Engineering

Date Approved: _____

To Mom and Dad;

who loved in absence of reason,

who believed in absence of conviction,

who supported in absence of outcome!

Acknowledgments

I would like to express my deepest gratitude to my advisors Dr. Joshi and Dr. Finin. They have been incredible mentors to me. They understood my strengths and weaknesses and have always steered me in the right direction. They pushed when required, while being supportive in letting me work at my pace. I learned a lot from them, about Computer Science and about research life.

I owe a debt of gratitude to my dissertation committee; Dr. Oates, Banerjee, Zaslavsky and Chakraborty. Their critical inputs were vital for my research and have hopefully allowed me to make a contribution to the field of mobile access control.

I will forever be indebted to all members of the Ebiquity Research Group that made this dissertation possible. Some of my friends like Abhay, Sunil, Jenn, Clare, Lisa, Mahbub, Ankur, Rajarshi, Dibyajyoti, Shehab, Sayantan and Varish have provided critical feedback on my work, for years. I am thankful to them for listening to my rants, speeches and presentations. I have had some incredible collaborators through my graduate studies like Roberto, Primal, Varish, Sandeep and Sudip and I am thankful to them for collaborating with me. Being a non-native speaker of a language comes with quirks that one might find difficult to avoid while writing. Therefore, I am extremely thankful to Renee and Shehab for proof-reading this dissertation and my research papers.

The splendid work that the CSEE department staff do every day, to keep our work free of complications, has huge implications on students finishing their dissertation on time. I am very thankful for their hard work, constant support and kind words (or ice-cream recipes) I received from Jane, Vera, Olivia, Keara, Dee Ann and Kara.

I understand the value of and appreciate the support I received through several of my advisor's grants including NSF grants 0910838, 1228198, 1439663, MURI award FA9550-08-1-0265 from the Air Force Office of Scientific Research, funds from the Oros Family Professorship, UMBC CSEE Department Award and the UMBC Graduate School Dissertation Fellowship.

Finally, I would like to thank my parents for helping me pursue my dreams.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
BOW	Bag-of-words
IDF	Inverse Document Frequency
TF	Term Frequency
SVM	Support Vector Machine
APK	Android Package Kit
SWRL	Semantic Web Rule Language
OWL	Web Ontology Language
DL	Description Logics

Chapter 1

INTRODUCTION

The future is Mobile! Mobile devices are the primary medium of user engagement [8] today, surpassing number of PCs on Earth. During I/O 2017, Google announced that the Android platform has over two billion monthly active users and last year Apple declared that they had sold the one billionth iPhone in July 2016 [58]. There are more than 3 million mobile applications on the Google Play Store [77] and Apple mobile application Store contains over 2.2 million mobile applications [78]. In spite of their massive popularity, these platforms have not had any major modifications to their access control model since inception. Both platforms use application sandboxes for mobile application's operations and use a permission based security model to provide access to system resources that are required for such operations. Users are expected to decide which of these access requests are to be allowed and which ones to deny. Unfortunately for users though, mobile device proliferation has provided fraudsters and identity thieves with ample opportunity to violate user privacy and security and steal users' data.

User's personal device are not the only devices that face challenges with respect to access control. The Bring-Your-Own-Device (BYOD) principle, adopted by corporations in recent years [54] creates a key challenge for access control for corporate data, as well. Similar to a personal device, in corporate environments, access rights can be

context-dependent. For instance, it might be permissible to send some generic data over the corporate VPN, but not have it uploaded to Facebook. It might be OK to use camera generally, but not inside the company facility. Reporting GPS locations to a platform provider (e.g., Google, Apple) might be fine in general, but not when inside a sensitive compartmented information facility (SCIF). The current "permit once" model followed by most mobile OSs are inadequate for handling such context-dependent access control tasks. In light of such potential problems, we submit that there is a need for fine-grained, dynamic and context-driven access control policies to protect the privacy and security of a user and her data.

1.1 Problem description

The challenges of mobile access control and the risks to users' data leads to two major problems. First problem is a need for methodologies to detect events happening on a user's device, in a specific context and capturing user's preference when an mobile application causes these events. Second problem is a need for informing a mobile user about a mobile application's expected behavior and providing a pre-defined policy for mobile applications with such behavior.

1.1.1 Thesis statement

A semi-automated approach that combines mobile application analysis with violation monitoring techniques can reduce the amount of user interaction required in capturing better access control policies that are fine-grained and context-dependent.

1.1.2 MITHRIL framework

In this work, we have created the MITHRIL [1] framework. MITHRIL is an end-to-end context-dependent access control framework that monitors mobile application activities on a user's mobile device, in various contextual situations and captures their access control preferences in that context. We use violation metric as the theoretical model for our framework. It helps us to determine convergence of the policy capture process.

MITHRIL also studies mobile application behavior and informs users of potential risks associated with an app. We have built the MITHRIL framework to fulfill the vision of context-dependent, fine-grained access control on the Android operating system. Details of the framework's operating features on Android have been described in Chapter 6. We show in this dissertation, that it is fair to use system calls to model mobile application behavior and then use such a model to create an initial default policy for users.

1.1.2.1 Policy capture

MithrilAC, the first component of MITHRIL is its mobile access control middleware. In MithrilAC [2], we capture mobile application behavior along-with user-context and compare them to currently known policy. Any deviation from known policy is then

[1] MITHRIL is a precious, lightweight and extremely strong silver steel from the Lord of the Rings which protected its wearer, Frodo, from life threatening dangers: `http://lotr.wikia.com/wiki/Mithril`

[2] MithrilAC requires certain operating system level privileges. Android being open source allows us to make these changes so we have used it for our prototype building but the concepts we have used applies to all mobile platforms.

submitted for review to user and their feedback helps MITHRIL refine their policy thus capturing user's access control needs. The refinement process is complete when no new deviations are observed or the precision of the captured policy is above a pre-defined threshold. We use the Semantic Web Rule Language [39] (SWRL), to represent our access control policy rules. We use the Platys ontology [41] written using the Web Ontology Language (OWL) [17], to model a hierarchical notion of user context. MITHRIL combines information about users' context, requested information and requester info as antecedents in policy rules that allows us to express complex rule conditions.

1.1.2.2 Mobile application analytics

Heimdall [3] is the second component of MITHRIL and acts as the mobile application analytics back-end. Heimdall uses machine learning classifiers to accomplish two separate classification tasks; i.e. classify mobile applications into benign-ware and malware using static permission features and perform multi-class behavior classification task using dynamic mobile application features like system calls made by an mobile application while running on a mobile device. The behavior classification results are used to enrich the Mobipedia knowledge-base (KB) [63], which [4] is an evolving KB created in a previous project. Mobipedia integrates mobile application knowledge from various sources and publishes it using Semantic Web technologies.

The key results of this work, presented in Chapter 8 proves the feasibility of using,

[3]Heimdall is the all-seeing and all-hearing Asgardian: `http://marvelcinematicuniverse.wikia.com/wiki/Heimdall`

[4]Mobipedia website:`http://mobipedia.science`

precision of captured policy, to determine completion of user policy capture process. They also prove the feasibility of using, dynamic system call features, to determine a mobile application's behavior class and static mobile application permission features, to detect malware mobile applications. Finally we show that using a crowd-sourced policy created using our mobile app analytics leads to reduction in user interaction required in the feedback process.

1.2 Contributions

In this dissertation, we have made the following contributions:

- Created an end-to-end context-dependent access control approach that monitors mobile application activities on a user's mobile device, in various contextual situations and enables the capture of their access control preferences in that context.
- Created a back-end mobile application analytics system capable of determining mobile "application behavior" and generating an initial default policy based on crowd-sourced data.
- Created a mobile-middleware system capable of observing mobile application behavior and using "violation metric" as a way to determine completion of policy capture process.
- Reduced user interaction required using curated policy output from app analytics back-end
- User study performed to show feasibility of using violation metric

- Custom ROM built for executing context-dependent policies
- Enriched the Mobipedia [62] KB with mobile application behavioral facts.
- Enhanced presence context detection using nearby messages and beacons.

Jointly these contributions form the MITHRIL framework which aims to provide a scientific basis for using mobile application behavior and policy violation as a means to determine mobile application risk and convergence of policy capture process. We have used machine learning classifiers to perform our behavioral analytics and user study with 30 users running LineageOS Android custom ROM (same as Android version 7.1.2). It is imperative to note that the user study did not focus on usability, A/B testing or developing better user interfaces. Rather we wanted to see the effectiveness of our methods in helping user's define their policies.

1.3 Dissertation document structure

- In Chapter 2 we present the background knowledge required to understand this dissertation. We also discuss the related work from the literature pertaining to this work.
- Through Chapter 3- Chapter 6 we explain the conceptual model and inner workings of the MITHRIL framework.
- Chapter 7 describes a pilot user study performed with 24 user to prove the effectiveness of using our techniques to carry out user policy capture.

- We discuss our experiments and evaluation results in Chapter 8.

- Finally, we conclude this dissertation with a discussion of our conclusions and possible future works in Chapter 9.

In the next chapter we discuss the various related work from the literature and background knowledge required to understand the domain. users

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter we will look at some of the background knowledge required to understand the domain. We will also take a look at some related works from the malware and behavioral analytics, access control, usable privacy, and context generation domains.

2.1 Android background

Android is an open source, Linux-based software platform containing five major components shown in Figure 1. The Android platform is built on top of a Linux kernel [30]. The hardware abstraction layer (HAL) provides interfaces to the device's hardware components that can be utilized by the higher level Java API framework to perform various tasks. For example, HAL consists of multiple library modules, that can be used to handle camera or bluetooth functionality. Each and every Android app, starting from Android version 5.0 (API level 21) or higher, runs in its own process and with its own instance of the Android Runtime (ART). ART is capable of running multiple virtual machines on low-memory devices by executing DEX files. DEX is a Dalvik EXecutable file using a bytecode format designed specially for Android that's optimized for minimal memory footprint. Android also includes a set of core runtime libraries that provide most of the functionality of the Java programming language, including

some Java 8 language features, that the Java API framework uses. The Java APIs form the building blocks of Android mobile applications by simplifying the reuse of core, modular system components and services, which include a View System you can use to build a mobile applications UI, a Resource Manager, providing access to resources like localized strings and layouts, a Notification Manager enables mobile applications to display custom alerts, an Activity Manager that handles application life cycle and Content Providers like Contacts. Many core Android system components and services, come in the form of native libraries written in C and C++. The Java framework APIs are used to provide interface to these functionalities. A core set of vital mobile application functionality are provided on Android devices using system mobile applications like Settings, email, SMS, calendars, browsing, contacts etc. These mobile applications are traditionally installed on the */system* partition of the device and have a set of privileges unavailable to mobile applications installed to user-installed apps. Any mobile application that wants to control how resources are accessed on the device needs to have certain privileges that are only available to System apps.

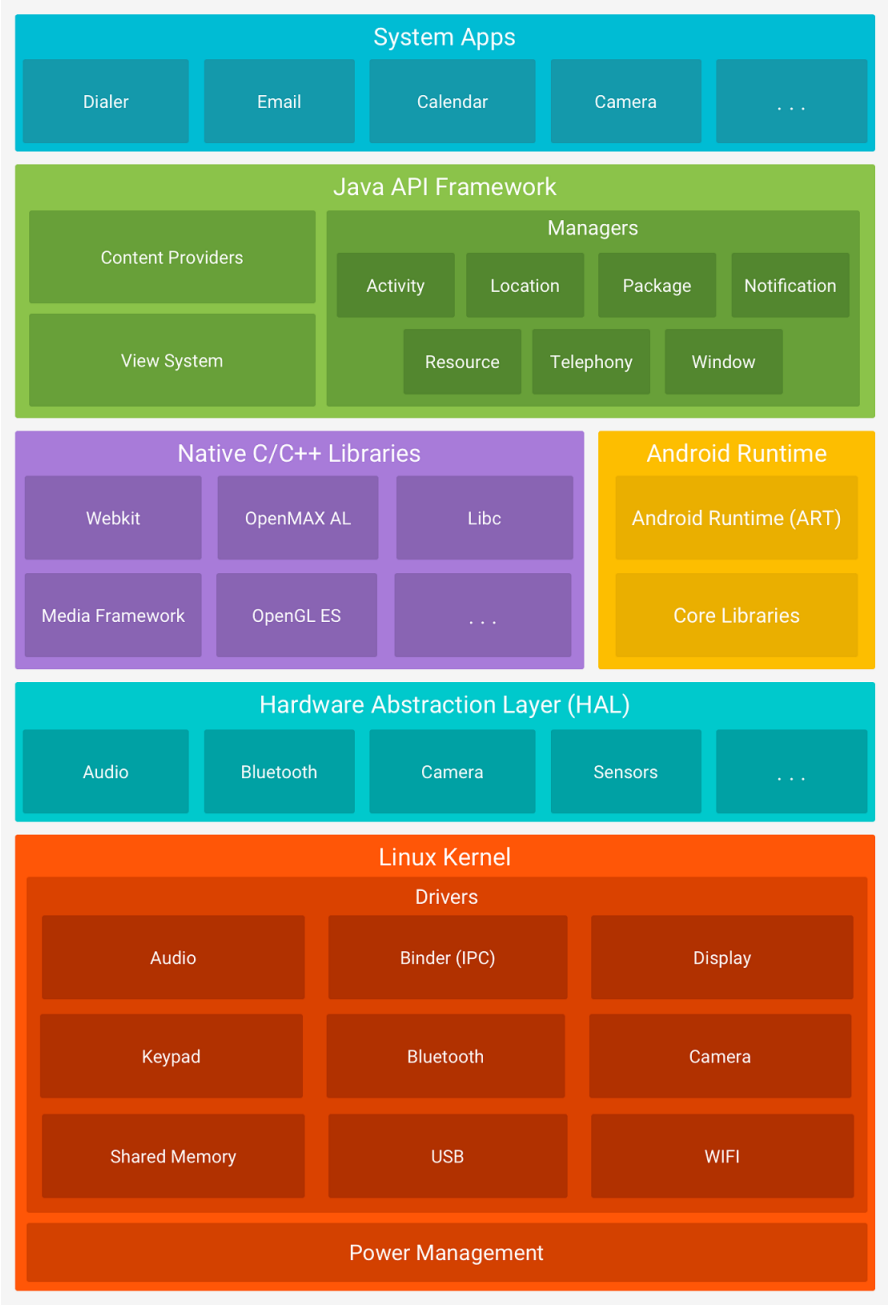

Figure 1: Android stack *courtesy: Google*

2.1.1 Android security model

The kernel [66] is a computer program that is the first program to be loaded when the machine boots up and has complete control over every resource in the system. The kernel connects application software to the hardware of a computer using a low-level interface called a system call. Android takes advantage of the security features offered by the Linux Kernel. Like in Linux, Android is also a multi user operating system. However, unlike a traditional desktop system where users have user ids assigned to them, Android assigns applications with a unique UID at installation time. Each mobile application on Android runs in a dedicated process associated with it's UID and is also given a dedicated data directory to which only the application has permission to read and write. Thus mobile applications on Android are isolated to their *sandbox* both at process as well as data levels.

The sandbox however, creates a new problem. Applications that can only access it's own files cannot really provide any interesting functionality to it's users. In order to use any resources on the device, android mobile applications have to explicitly request a *permission* that would allow it to say, connect to the Internet or use some service like location.

2.1.2 Application signatures and permissions

Android applications are signed with a digital key created by it's developer. This allows the system to enforce signature permissions for an application's target processes and shared user ids. Android uses four keys to maintain platform security:

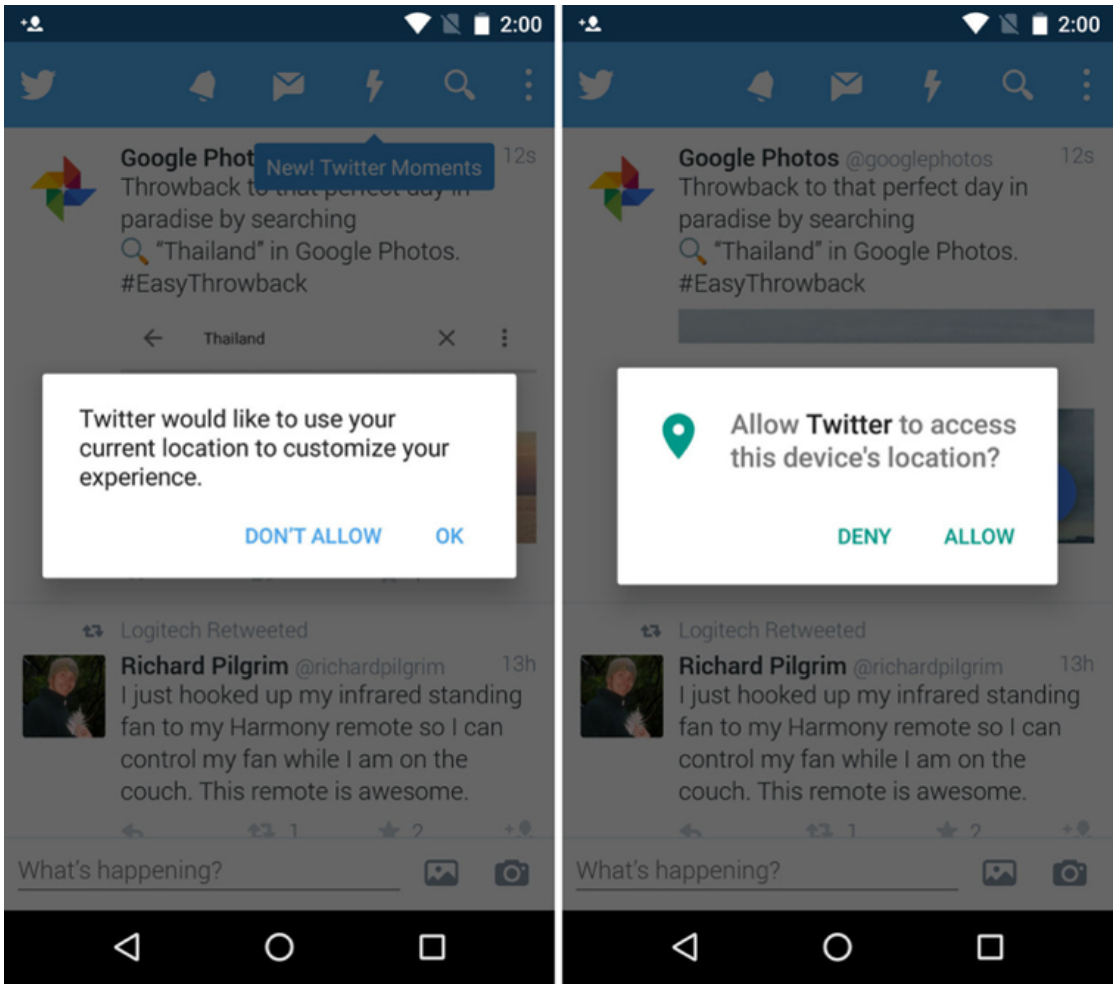

Figure 2: Android run time permissions

- platform: key for packages part of the core platform
- shared: a key for packages shared in the home/contacts processes
- media: a key for packages part of the media/download system
- testkey: default key to sign when unspecified

Keys come as two separate files: the certificate, with extension .x509.pem, and the private key, with extension .pk8. The private key is used to sign the application package

and should be kept secret. The certificate, in contrast, contains the public key and is used to verify a package has been signed by the corresponding private key. Android Package Kit (APK) is the file format used for distribution of mobile application executables for the Android Operating System. Prior to distribution, the APK file needs to be signed by the developer private key. When a new version of an application is created, that APK also needs to be signed by the same key as the old application in order to get access to the old applications data. Otherwise the old application would have to uninstalled before the new version is installed. Two or more applications may share data or each others resources by defining a shared user ID and signing all applications with the same key. Application signatures allow the Android operating system to associate access rights to them through a Linux UserID. Android has two types of "permissions":

- Standard UNIX/Linux file system permissions
- Android (JAVA) API permissions

A discussion on Linux file system permissions is beyond the scope of this work. Android API permissions on the other hand are something that we do control. Most low-level system functionality in Android are provided using system calls but mobile applications mostly access Android high-level APIs. These APIs in turn communicate with the low-level system calls that have the proper permissions to access the drivers and components defined in the Hardware Abstraction Layer or the Linux kernel. In order to access these low-level system services through the high level APIs applications are required to request Android API permissions. These requests are included in a file

called the AndroidManifest.xml. Android permissions have potential risks implied that are defined using protection levels:

- A “normal” protection level implies lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user.
- A “dangerous” permission implies higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. Starting from Android 6.0 Marshmallow dangerous permissions have to be explicitly allowed by users at run-time when the mobile application tries to use it the first time. See Figure 2. If not handled properly by a developer, this causes a `SecurityException`.
- “signature” is a permission that is granted only if the requesting application is signed with the same certificate as the application that declared the permission.
- Finally, “signatureOrSystem” is a permission that the system grants only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission.

2.1.3 Application operations

mobile application Ops or application operations is an Android API that was first developed as part of Android 4.3 “Jelly Bean” and was removed from the public API as of Android 4.4.2 “KitKat”. The API is now hidden and no longer accessible in the pub-

lic Android SDK. This API allowed programs to interact with "application operation" tracking on a mobile device. It is part of the `AppOpsManager` class and instances of this class must be obtained using `Context.getSystemService(Class)` with the argument `AppOpsManager.class` or `Context.getSystemService(String)` with the argument `Context.APP_OPS_SERVICE`. As of Android 7.1.2 it is only possible to access this API through a rooted phone on a custom ROM that does not block access to the API for third-part apps. Using this API it is possible to revoke a permission granted to an mobile application during installation or at run-time.

In Chapter 6 we describe how we took advantage of Android APIs and how we created a "hacked" version of the Android SDK to build a system level application. We also built the vision of context-dependent, fine-grained access control on Android by recompiling a custom ROM of our own based on Android version 7.1.2, which is the current official version of Android.

2.2 Access control background

The domain of access control is well researched. We discuss some of the areas of access control, that are relevant to this work.

2.2.1 Policy representation

Role Based Access Control (RBAC) [1] and Attribute Based Access Control (ABAC) [40] are two most popular access control models, that have been used to achieve the goal of managing access control, in various domains. In the mobile do-

main, Ghosh et. al. [27] used a semantically rich context model to manage data flow among applications and filter them at a deeper granularity than it was possible using available security mechanisms on smart-phones. The CR$\hat{e}$PE system [14] was one of the earliest known ABAC model implementations using the XACML standard [64] for fine-grained context-related policy enforcement on smart-phones. CR$\hat{e}$PE didn't use Semantic Web but it followed the ABAC model.

Kagal et. al. [45] used distributed policy management as an alternative to traditional authentication and access control schemes. Rei, a policy language described in OWL and modeled on deontic concepts of permissions, prohibitions, obligations and dispensations [45], have used Semantic Web technologies to express what an entity can/cannot do and what it should/should not do. In Rei, credentials and entity properties like user, agent, etc are associated with access privileges. This allowed Rei to describe a large variety of policies ranging from security policies to conversation and behavior policies. The Rein framework [44] which builds on Rei and is based on N3 rules and uses a CWM reasoning engine for distributed reasoning. KAoS [80] relies on a DAML description-logic-based ontology of the computational environment, application context, and the policies. The KAoS system was capable of supporting runtime policy changes and was extensible to a variety of platforms. In ROWLBAC [23], the Web Ontology Language (OWL) [17] was used to support the standard RBAC model and extending OWL constructs used to model ABAC.

In our work, we use Semantic Web technologies like an hierarchical context ontology defined using the Web Ontology Language (OWL) [17]. Our rules are defined

using the Semantic Web Rule Language [39] that allows us to express rules that are more expressive than ones that can be defined using OWL. We use ABAC as our access control model. On the mobile device we have used reasoners created by Roberto et. al. [87] for Android devices to extract and infer knowledge about user context. MITHRIL's mobile middleware connects sensed context data to a high-level abstraction of context and executes defined rules to protect user data.

2.3 Mobile security research

Security research in mobile domain has focused on three different aspects, malware detection, static/dynamic code analytics and mobile application behavioral studies.

Mobile malware detection [7, 12, 55, 76, 79, 89] has shown fair amount of success. However, a 2014 McAfee Labs report [75] predicted that mobile technologies would see an escalation of attacks, due to openly available mobile malicious source code. Malware are not the only threats faced by mobile users anymore. A Google taxonomy provides us with additional threats that users face through Potentially Harmful mobile applications (PHA) [74], such as Billing Frauds, Spyware, Hostile Downloaders, Privilege Escalators, Ransomware, Rooting mobile applications. According to this report the mobile threat model has changed and today a PHA mobile application may use legitimate components of user's device in addition to operating system security vulnerabilities to harm users. In fact, legitimate and popular mobile applications like Pandora, Deutsche Telekom [9] have been previously exposed to have created serious

privacy and security concerns over handling of user sensitive data like location, device id and other personally identifiable information (PII) including gender and year of birth.

TaintDroid [20] is a dynamic code analytics project that performed dynamic taint tracking to detect when data leaves user device [20]. Few other static code analysis projects attempted to mix static code analytics with malware detection [24, 68, 67]. NLP techniques have been used by Pandita et. al. [61] to perform mobile application behavioral analysis. This project was able to achieve an 83% precision and 82% recall in determining why an application uses a permission through NLP techniques. Although a good first step in behavioral analysis, it leaves a lot of room for improvement because their analysis included only 3 popular permissions used by apps. In [34], researchers have attempted to map an app's description from the Google Play Store [28] to its actual behavior. Gorla et.al. [34] provided the following insights. Application vendors hide what their mobile applications do. While Google maintains no standards to avoid deception on a developer's part. This results in developers deceiving users. One such incident occurred in case of the "Brightest Flashlight" mobile application, when it collected user location and surreptitiously uploaded the same to an advertising network [47]. The other insight was Android's permission model is flawed and requires lay users to read incomprehensible permission descriptions like "allow access to the device identifier". We use our behavioral analysis technique to determine expectations from an app, which can then be matched to appropriate restrictive policies.

A study by Kosoresow et. al. [48] tells us that system calls can be used to determine application behavior. Consequently, we use system call analytics in our work

to achieve the complicated goal, of mobile application behavior classification. We consider this task to be a complicated when compared to malware classification due to the fact that, it is sometimes difficult to determine if an app's behavior was a legitimate functionality or an illegitimate behavior. We attempted to capture this distinction in our annotation process.

2.4 Usable privacy research

The state-of-the-art in research on policy capture stops at determining generalized privacy profiles [2, 71, 52, 53]. These works conclude that it was possible to create privacy "profiles" applicable to user categories on mobile devices with reasonable accuracy. When it comes to defining their own rules, it was observed by [71] that users were not good judges of how well a rule meets their true needs or preferences. However, in their other work they showed that with enough "privacy nudges", explaining how their location was being shared, users could be guided into modifying their preference. We argue that given a set of policy violations and a hierarchical context model, users would be able to define their preferred policy. We focus on using context generalization and specialization with assistance from our Platys ontology [41] driven context model, and combining that with user feedback to reach an individual user's preferred specific policy.

An important issue with privacy preservation on mobile devices is that users tend to be privacy pragmatists [49]. Although every person would prefer that their data remain secure and private, the moment they realize the potential advantage of using

an application, they choose to ignore such preferences. One way to ensure user data remain private and secure would be to educate them about mobile applications behavior. For that purpose MITHRIL uses system call analytics to perform mobile application behavior classification. Upon successfully classifying an mobile application into it's behavior class a policy for that class is created in our system using crowd-sourced data. To the best of our knowledge app-behavior classification and policy association has not been carried out by anyone till date.

2.5 Context discovery research

Context discovery on mobile devices [88] has shown significant success with semantic location [50, 88], activity recognition and complex activity recognition [11, 22, 72], and situational awareness [85, 5]. This work takes advantage of knowledge of context discovery techniques from earlier projects and adds-on a preliminary presence context detection mechanism. In addition to the standard context information of location and activity we use Android Nearby APIs and Bluetooth IDs to discover presence information. In our previous work [16], we used the Nearby API [31] from Android along with beacons to answer questions while preserving privacy of the enterprise data. We incorporate those techniques in the current work but reverse it to detect presence of relevant actors to our user.

In the next chapter we talk about the high level conceptual model of the MITHRIL framework.

Chapter 3

CONCEPTUAL MODEL OF MITHRIL

In this chapter we describe the high-level conceptual model of the MITHRIL framework (see Figure 1 for the conceptual design diagram). Additionally, we present our algorithm for violation based policy capture. Next we define some technical terms and concepts involved in this dissertation.

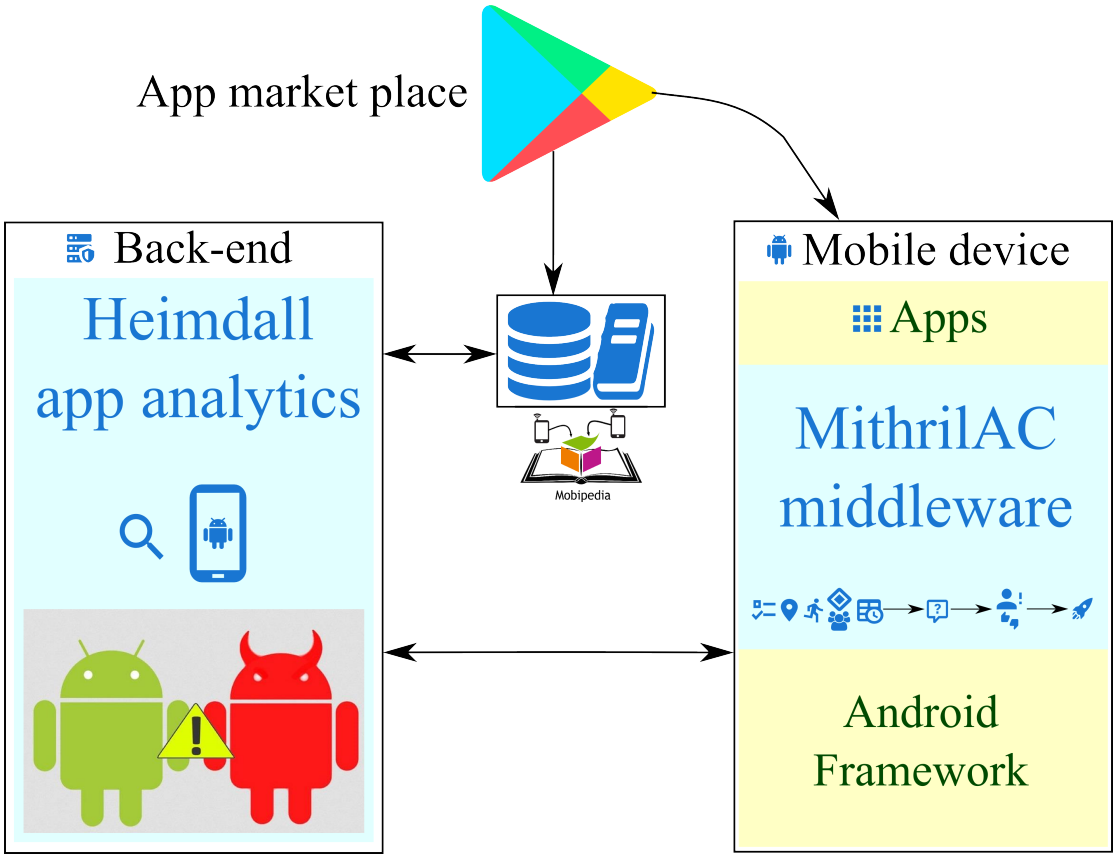

Figure 1: MITHRIL conceptual model

3.1 Concepts of MITHRIL

We have mentioned policies and policy rules previously. A policy applies to a user and her device and consists of a set of rules that control the behavior of an app in a given context. Following is a formal definition of "rule" by Fuernkranz [25]:

Definition 1. *[...] an expression of the form:*

IF Conditions THEN c

where "c" is a class label, and "Conditions" are a conjunction of simple logical tests describing properties that have to be satisfied for the rule to 'fire'.

MITHRIL uses context-sensitive privacy policy rules defined in the Semantic Web Rule Language (SWRL) [39] to manage the privacy of data. The abstract syntax for SWRL rules follow the Extended Backus-Naur Form (EBNF) notation which, while useful for XML and RDF serializations, isn't particularly easy to read. For readability, we use the following informal format: `antecedent` $\Rightarrow$ `consequent`. Antecedent(s) must hold for a consequent to apply. Multiple antecedents in a rule are defined as a conjunctions of atoms. The consequent atom states whether the access is allowed or denied. Antecedents in our rule specification consist of the *context of a requesting entity*, along with the *entity type* that is being requested. A more abstract representation may be considered as a triple (R, C, Q) which contains: R, that represents the requester's context, C is user's context and Q is the query received by the system. The user context 'C' follows the definition suggested by Dey et. al. [18]:

Definition 2. *"[...] any information that can be used to characterize the situation of an*

entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and application, including the user and applications themselves."

Dey et. al. [18] also decompose context into two categories: primary context pieces (i.e., identity, location, activity, and time) and secondary context pieces (pieces of context that are attributes of the primary context pieces e.g., a user's phone number can be obtained by using the user's identity). Secondary context could provide us with added knowledge but in MITHRIL we have focused only on primary context pieces to reduce the complexity of the rule capture process and to reduce energy consumption of our rule execution system. Additionally, since our work focuses on user specific mobile access control policies, the use of identity becomes redundant. We therefore use the presence information, or who is nearby to our mobile user as part of our user context.

We have previously stated policy rules in our framework are fine-grained. "Fine grained" access control refers to the

Definition 3. *"[...] amount of details that define whether a certain permission will be granted or denied."*

For example, attribute-based access control [43] through a standard like XACML [64] can consider attributes about users, resources, environment and for deciding whether to grant or deny an access. Role-based access control on the other hand focuses on user role thus losing some granularity of control. When applied to mobile access control this essentially signifies that we are considering user context, app attributes, resource requested etc. to decide if the access request will be granted or denied.

User access control policy in MITHRIL are context-dependent. "Context-dependent"

policy refers to

Definition 4. *"[...] any set of rules that allows access control over data depending on current context."*

Essentially, access control decisions are dependent on user location, what activity they are involved in, time of day or who is nearby to the user (i.e. is the user in presence of their co-workers, superiors, subordinates, family, strangers etc.)

3.2 Motivation for Context-dependent access control

Context-dependent access control is at the epicenter of this work, so we need to take a look at couple of motivating examples to further elucidate this key concept. Let's assume for a moment that we have the ability to determine if an app is "safe".

Example 3.2.1. Let's take a look at Jane Doe, a researcher working at a university on a government project. She regularly meets representatives of the government agency for her work. She uses two calendar mobile applications to manage her meetings. Her university calendar called *UniCal* and another calendar application called *SmartCal* with interesting features like auto to-do lists, etc. She receives calendar invites on her mobile device, a phone that she uses for both personal as well as, work purposes. Given the sensitive nature of her work she prefers that any calendar information about her work remain inaccessible to *SmartCal*. She is careful and shares her work related information through a work email id, which is only accessible when she's behind the firewall at the university. We assume that her calendar provider, a mobile component that allows mobile mobile applications to access calendar data, does not store the data on the phone

in order to save space and fetches it dynamically when requested. We know that the only way a calendar app can use a calendar provider on the phone is to ask for the mobile operating system's calendar permission from Jane. When she first installed *SmartCal*, she allowed the calendar permission. She now realizes that allowing this permission can lead to potential data-breaches and has to go and block it every time before she uses her corporate calendar at work. Now, imagine we have a permission model that dynamically blocks access to calendar data at her work location for her mobile applications unless the app has been specifically approved. A rule-set that ensures calendar access be allowed to *SmartCal* only if Jane is NOT at *Work* would prevent such a data-breach.

...a dynamic context-dependent privacy policy would stop such a data-breach

Example 3.2.2. Let us take a look at another example. Jane recently found a calculator app with interesting features like unit conversion, date conversion etc. The default mobile app that came pre-installed on her device can only do simple arithmetic computations. She installs this app and discovers that it requests location acces. She finds it a bit odd but decides to use the app because sometimes she uses it's unit conversion feature. She allows location access but doesn't realize that this app tracks people with jobs in the government sector near a specific location, supposedly to provide targeted ads but really for nefarious reasons. Now, imagine a mobile operating system that explains to her that this app could potentially be unsafe because calculator mobile applications generally don't need location to function. Given such a warning, she could have made the decision to uninstall the app or block location access when she's in her *Work state* (which is a sensitive piece of data for Jane). Note we are using a hierarchy of context

in this case to provide greater location coverage for a defined policy. This is equivalent to creating a dynamic policy, for our questionable app, at every location in Jane's work state.

...a hierarchical context-dependent policy would stop exposure of such sensitive data

3.3 Framework design

MITHRIL has two major components as shown in Figure 1. MithrilAC mobile middleware and Heimdall app analytics back-end.

3.3.1 Framework component: MithrilAC middleware

MithrilAC is the primary component of MITHRIL. It is a mobile middleware with system manipulation privileges and uses context-dependent policy rules to manage access control on the device. Context in MITHRIL is defined using a hierarchical model that was first proposed in research work done in the Platys project [42].

3.3.2 Context ontology

The MithrilAC middleware uses the Platys ontology that allows us to define a high-level abstraction of context. The Platys project [41] builds on previous work where strong support for context reasoning using ontologies for explicit semantic representation of context [10] has been developed. MithrilAC uses Semantic Web technologies to specify high-level, declarative policies in the form of SWRL rules. In this ontology, context is modeled to include a semantic notion of a Place. Although Android APIs

capture a user's location at the level of position, i.e., geospatial (latitude-longitude) coordinates, it can then be mapped to a Place or geographic entity, such as a region, political division, populated place, locality, and physical feature. A position while being valuable on it's own, from the standpoint of context, Place is a more inclusive and a higher-level abstraction. Using the Platys ontology a *User* is associated with a *Device* whose *Position* maps to a geographic place (*GeoPlace*) such as *"UMBC"* and to a conceptual place (*Place*) such as *"At Work"*. Some *GeoPlaces* are part of others due to spatial containment and such relationship (*part_of*) is transitive. The mapping from *Positions* to *GeoPlaces* is many to one and the mapping from *Positions* to *Places* is many-to-many (the same *Position* may map to multiple *Places*, even for the same *User*; and, many *Positions* map to the same *Place*). Mapping from *Positions* to *Places* is done through *GeoPlaces* (*maps_to* is a transitive property). An *Activity* involves *Users* under certain *Roles*, and occurs at a given *Place* and *Time*. *Activities* have a compositional nature, i.e., fine-grained activities make up more general ones. This approach reflects the pragmatic philosophy that the meaning of a place depends mainly upon the activities that occur there, especially the patterns of lower-level activities. The idea applies at both the individual and collaborative level. Such hierarchical context enables generalization or specialization of conditions that apply for a policy rule.

3.3.3 Presence context using Nearby

The Nearby APIs were created by Google last year for creating interaction patterns with nearby objects [31]. For presence context detection we have defined a re-

lationship in the Platys ontology as sitsIn. This relationship allows us to define that a person has an office room assigned to them in an organization. The subject of this relationship can be a “Supervisor”, “Subordinate” or “Colleague”. The object of the relationship can be a “Location_Rroom”. In order to obtain this information we use nearby messaging API from Google that allowed us to deploy a Physical Web of low energy Bluetooth beacons. An example of the utility of such a web or physical infrastructure is the Carlton project [16] which was used to achieve privacy of the organization while responding to natural language queries made about entities in the organization. Using this technique we are able to generate the notion of “User is in front of her Boss’s cabin”, which then allows us to execute policies that contain antecedents that represent presence constraints.

3.3.4 Violation Metric

As discussed in Chapter 2, user driven policy capture process have shown some promise. However, they have also faced challenges of user indecision perhaps originating from a lack of understanding of an app’s behavior [51]. Deciding that the system has indeed captured the access control preferences of an individual user thus becomes a challenge. We present a violation detection driven process as a way to determine when the policy capture process is complete. Violations are actions performed by apps (due to user action or otherwise) that are prohibited by current known policy. We argue that when an such a violation is recorded in a specific context, our current policy requires a modification in order to correctly represent user’s preferred restrictions.

For the violation detection process, MithrilAC detects app launches and actions performed by the app. Since policy rules in our framework are written as a function of context, the middleware is able to determine if an action performed by an app in a Semantic user context, is in-fact allowed by a currently active policy. If the action is not allowed, it marks the action as a potential policy violation and presents to the the user information about the event. If the user feedback is to block the action in question in the future then we have captured a "True Violation"(TV in short). On the other hand, if the user wants to allow the action or wants to change the currently active policy, a "False Violation"(FV in short) has been captured. When we capture a false violation, the user is allowed to add/delete/generalize or specialize the conditions for a policy rule. The hierarchical context ontology enables the generalization or specialization of rule conditions. Assuming true violations as true positives and false violations as false positives, we can then compute the precision of the current policy as follows:

$$VM = \frac{TV}{FV + TV}$$

.

We refer to policy precision as the "Violation Metric" (or VM metric), throughout this work. Since we capture user feedback periodically, if at the end of a feedback period the VM metric reaches a pre-defined threshold, we know that we have captured most situations that the user truly intends to block. This means that it is safe to start implementing our captured policy by executing the rules.

3.3.5 Dual operational mode

There are two operational modes for the rule capturing middleware; OBSERVER and ENFORCER. OBSERVER mode corresponds to the phase where the system passively observes events on a mobile device. ENFORCER mode refers to the phase when our middleware actively blocks operations on the device that are in violation of captured access control policy rules. The first module of our rule capturing middleware is "Violation Recorder" module. This module records policy violations in our mobile internal knowledge-base "MithilKB". The internal KB is populated with policies from our Mobipedia knowledge-base [62]. Captured policy modifications are also stored in this internal KB. We denote the initial policy as *P* and our goal is to capture the modified user policy of *P'*.

An important factor in policy capture process is the initial policy. When the MithrilAC middleware is installed on the mobile device, it downloads an initial default from the app analytics back-end. We have run experiments with users and found that if we use a default deny policy, we can capture policy rules using MITHRIL but it also causes the system to present the user with a lot of feedback point. This leads to user either not responding to them or responding to just a few of them. The state-of-the-art for user rule capturing have established that it is possible to do so from scratch. However, the range of variation of accuracy is pretty high, from 30% to 80%, as seen in [2, 71]. Due to these reasons, we use policy rules that have been collected from the crowd. A crowd-sourced policy would be less likely to cause wild user policy fluctuations. It would also create much less feedback points due to less number of likely

violations occurring on the device. Although it is possible to use MITHRIL to capture policy rules from scratch, our focus is on capturing the modified policy *P'* starting from an initial default policy *P*. We show that given proper information about events happening on their mobile devices, in context, users can choose their data sharing rules better. An example rule, is shown below in plain English along with it's SWRL form.

Example 3.3.1. *If user location is Work, in presence of Supervisor, and requester is a Social Media app, requesting access for Camera, then deny access* to camera

```
@prefix platys:<https://www.ebiquity.org/ontologies/platys/1.0>.
platys:App(?requester) ∧
platys:AppCategory(?requester,"Social_Media") ∧
platys:Reuqest(?requester,"Camera")
platys:Resource("Camera")
platys:hasCurrentLocation(?userB,"Work") ∧
platys:Location("UMBC") ∧
platys:PresenceInfo(?userA) ∧
platys:Supervisor(?userA,?userB) ∧
platys:affiliatedWith(?userA,"UMBC") ∧
platys:affiliatedWith(?userB,"UMBC") ∧
⟹
platys:denyAccess("Camera")
```

Figure 2: SWRL representation for example rule

3.3.6 User Feedback Algorithm

The USER FEEDBACK ALGORITHM helps us capture modifications to the initial policy. It uses the hierarchical contextual options encoded using the Platys ontology to

capture the afore-mentioned modified policy *P'*. In this algorithm the user has the option of accepting the rules or modifying rules by adding, deleting or changing contextual conditions in which a rule applies.

Algorithm 1 "User Feedback capturing" - Algorithm to learn user rules from user feedback

1: $appsOnMobileDevice$=get apps on mobile device
2: **for each** $appsOnMobileDevice$ **do**
3: Observe app launches.
4: Capture resource requested by app.
5: Collect Snapshot of context.
6: Find out policy rules that deny resource access to app in current context.
7: Store potential violations in for deferred user feedback.
8: **end for**
9: **for each** $recordedViolations$ **do**
10: **if** User denoted as false violation **then**
11: Ask user to modify rule.
12: **if** User wants to add a condition to the rule **then**
13: Let user choose one of the conditions to add.
14: **else if** User wants to delete a condition from the rule **then**
15: Let user choose one of the conditions to delete.
16: **else**
17: **if** User wants to generalize a condition **then**
18: Provide user with a more generic condition as defined by the Platys ontology.
19: **else**
20: Provide user with a more specific condition as defined by the Platys ontology.
21: **end if**
22: **end if**
23: **else**
24: User denoted as a true violation.
25: **end if**
26: **end for**

The completion of the rule capture process is determined using the Violation Metric we defined earlier in this chapter. Once the process is complete the system shifts to the ENFORCER. The policy precision or VM metric value can be pre-adjusted at a threshold when the execution of policies will commence. Depending on the purpose of the mobile device, as in the device is used in a BYOD scenario or as a personal device, the threshold may be adjusted by an administrator or mobile user.

In the work done by [71] it was observed that using standard machine capturing techniques like Random Forest Classifier [37], it was possible to improve rule accuracy. However, they noted that such a system is not a good solution because the user would lose control over what the rules would do or even understand them. We ascertain a secondary issue where significant amounts of data would be required to perform such a learning task and a learning system would require retraining and model creation, if the policy preferences of the user changes over time. Given that we are using a violation based technique, MITHRIL will be able to easily determine such a change and start collecting them automatically.

As an alternate solution to machine learning techniques, they proposed a user defined rule capturing system. They also carried out a user burden study and observed that with a complex rule definition and higher number of rules, accuracy of the policy was significantly higher than using a simple whitelist (define every entity that is allowed to receive said data) approach. Therefore, we hypothesize that it is possible to increase the accuracy of the rules by making it easier for the user to understand the rules.

3.3.7 Framework component: Heimdall back-end

MITHRIL framework also contains a back-end app analytics system that collects mobile app metadata from the source market place. In our case, this is the Google Play Store. We have collected metadata about 2.3 million apps and permission information about 934k apps that are available from the Google Play Store. We incorporated permission data from the Playdrone project [82]. We then transformed our collected data into RDF triples and stored it in our Mobipedia knowledge-base [62]. In this component, we also perform our app analytics by downloading the executable file for an app and running it on an emulator and capturing behavioral patterns using a automated scripts that simulate clicks performed by a user on a mobile device. The results of our analysis are also stored in the Mobipedia knowledge-base to be queried by systems that want to use knowledge of an app's behavior.

System calls have been traditionally used to monitor programs in computing systems [48]. In MITHRIL we use this theory to try and determine if system calls can be used to distinguish between how an app "behaves" and its perceived/stated purpose. There are mechanisms to capture system calls made by an app on a mobile device. The goal of Heimdall, MITHRIL's app analytics system, is to classify an app into it's behavioral class and generate an initial set of policy rules using using crowd-sourced policy rules. We generate policies based on majority voting for a specific permission request in a given behavioral class. There are major challenges involved in behavioral classification. We discuss them in the next chapter but it is imperative to state that even if an app's behavioral class cannot be determined, we have collected app category data from

the Google Play Store for almost 80% of the store. We use the Play Store category to generate the initial set of policies for an app.

Why is behavioral classification useful in creating an initial set of policies?

Let's consider the case of the "Brightest Flashlight App". This app was sanctioned [47] in 2013 for collecting user locations and surreptitiously uploading the same to an advertising network. In this case, the behavior class of "flashlight" tells us that this app should not have access to any other permission than camera to turn the flash on or off. However, this app did more than such an expected behavior and collected location information of users. Determining whether an app's behavior was indeed legitimate is often complicated and requires in-depth analysis of app behavioral patterns and matching them to expected behavioral patterns. However, if it were possible to incorporate the human knowledge of "location access" being weird for an app's functionality then we could possible prevent leakage of user privacy. That is why we use policies obtained from the crowd and use system call analysis because in general a flashlight app, would only make system calls related to the camera API. The above app however, would make additional system calls related to GPS and Networking.

In the next chapter we discuss violation driven user policy capture.

Chapter 4

POLICY CAPTURE MIDDLEWARE

In this chapter, we describe the functionality of the mobile middleware. The key research question we are trying to answer through this system is as follows:

RQ1: Given an initial policy P and user goal policy P', can violation metric be used to determine the completion of the capture process?

4.1 System Overview

The system architecture of our policy capture middleware, MithrilAC, is shown in Figure 4. MithrilAC contains four main components: a policy enforcement module, a policy decision module, a policy store module and a user policy control module. MithrilAC sits between application layer and framework layer. It takes as input a request for data or component access. It's output contains requested data or access to a component or an exception stating that data or component is unavailable. To represent access constraints, MithrilAC uses an attribute-based access control (ABAC) model [40], where the attributes represent user context, requested resource and requester meta-data. We use ABAC as it provides us the flexibility of having any number of attributes to be added to our rule.

MithrilAC has two operating modes: OBSERVER and ENFORCER. In observer mode the system does not enforce access control policies, but simply notes all violations

of current policy rules. In this mode, feedback requested from user periodically, on the recorded violations, in order to capture their 'preferred policy'. The frequency for feedback is a system setting that is adjustable by user or system admin. After an initial round of policy capture and user interaction MithrilAC moves to ENFORCER mode, in which it enforces current applicable policy rules. The transition between the two modes is determined using a predefined, but adjustable, threshold for the VM metric.

Definition 5. A policy rule VIOLATION, is recorded when a rule defines an access restrictions for an app and a behavior is observed that tries to defy such a restriction.

Le's assume our rule is "Do not share **camera** resource with **social media** applications at **work**". A violation is recorded if for some reason the camera is accessed by a social media app at work. During the feedback cycle this violation will be marked as a TRUE violation (hereafter denoted as *TV*), if the user agrees that this behavior is indeed unexpected. For example, if the camera was accessed by Instagram, a social media app, when at work and the user did not expect this observed behavior, then we have a true violation captured. A violation is considered to be a FALSE violation (hereafter denotes as *FV*), if the user expected observed behavior. For example, the camera might have been accessed at work by Instagram, but the user initiated it while at lunch in the cafeteria. Using the *TV* and *FV* frequencies we compute our Violation Metric (VM). Our VM metric, computed as follows, helps us determine if MithrilAC is ready to transition from OBSERVER mode to ENFORCER mode:

$$VM = \frac{TV}{FV + TV}$$

This VM metric computes the precision of the known policy, as in the ratio of true positives and sum of true and false positives. Here, we are defining true violations as true positives, which signifies that the default policy *P* and the user's preferred policy *P'* were the same and *NO* modifications to the original policy will be required. On the other hand false violations or false positives are situations when the default policy *P* and the user's preferred policy *P'* differ and we need to capture change in current policy. A high value of the VM metric signifies we are closer to a user's "personalized" policy.

4.1.1 Assumptions

Throughout our work, we use the following example to explain internal working mechanisms of MithrilAC and the example does not denote the limitations of the system rather it is used for clarity. Our example uses a policy applicable users a "graduate student". We assume that our users work for a confidential research project and protecting their data is of critical importance. We assume that they start from an initial policy *P*, that they are allowed to modify to better protect their data. We use Android mobile devices for our system implementation as Android is open source thus allowing us to modify system behavior and execute policies to block access to resources on the device.

4.1.2 Policy Store

The policy store module in MithrilAC has a knowledge base containing a currently applicable policy for the mobile device. Research conducted by [2, 71, 52, 53] have established that it is possible to fairly accurately create privacy profiles applica-

ble to user categories on mobile devices. The MITHRIL framework intends to extend this domain by creating a methodology to capture user specific policy rules. We use an initial policy, as a starting point for our system.

The storage module takes as input a requester app's information and information about the requested resource and searches the policy knowledge base for the applicable policy rules and returns the same to the policy decision module. The second task that the policy storage handles is updating a policy rule as requested by the user policy control module. Now, let us take a look at how rules are represented in MithrilAC.

Rule Representation: Rules, in our system, are expressed using SWRL [39]. A more abstract representation may be considered as a triple (R, C, Q) which contains: R, that represents the requester's context, C is user's context and Q is the query received by the system. The consequent of a rule defines the action to be taken. We define some use cases in detail in the following section but for now we present an example rule where, we have an app that belongs to the social media category. Let us take a look at a rule from our policy called GRADSTUDENTPOLICY for graduate students, called SOCIALMEDIACAMERAACCESSRULE. The rule states that, while the student is in a university building, social media apps are not allowed to access camera on her mobile device. The rule is shown Figure 1.

```
@prefix platys:<https://www.ebiquity.org/ontologies/platys/1.0>.
platys:ResourceRequested(?r, "Camera") ∧
platys:requestingApp(?app) ∧
platys:hasAppType(?app, "SocialMedia")∧
platys:User(?u) ∧
platys:userLocation(?u, ?l) ∧
platys:hasLocationType(?l, "University Lab")
⟹
platys:denyAccess("Camera")
```

Figure 1: Simple rule for controlling social media camera access

Example of a higher granularity rule can be seen in Figure 2, which has more conditions incorporated. In plain terms we are now stating that instead of just being applicable in a university building, we "Do not allow camera access to "Social Media" apps when the time of day is between 9AM and 5PM and it is a weekday and the user is at university lab location in presence of her Advisor and has a meeting scheduled with her Advisor".

```
@prefix platys:<https://www.ebiquity.org/ontologies/platys/1.0>.
platys:ResourceRequested(?r, "Camera") ∧
platys:requestingApp(?app) ∧
platys:hasAppType(?app, "SocialMedia")∧
platys:User(?u) ∧
platys:userTime(?u, ?t) ∧
platys:timeAfter(?t, "0900") ∧
platys:timeBefore(?t, "1700") ∧
platys:userDayOfWeek(?u, ?d) ∧
platys:hasDayType(?d, "Weekday") ∧
platys:userActivity(?a) ∧
platys:hasActivityType(?a, "Advisor\_Meeting") ∧
platys:userpresenceInfo(?p) ∧
platys:hasPresenceType(?p, "Advisor")∧
platys:userLocation(?u, ?l) ∧
platys:hasLocationType(?l, "University Lab")
⟹
platys:denyAccess("Camera")
```

Figure 2: Rule with higher granularity, for controlling social media camera access

4.1.3 Policy Decision

The policy decision module receives as input, a request meta-data from policy enforcement module. The current context is obtained using a context synthesizer sub-module. The context synthesizer keeps user context facts updated using an OWL-DL reasoner and a context ontology to infer high-level and semantically rich context. A similar technique for context inference from low level sensor information was explored in [35]. We use the Platys ontology [41] to semantically represent user context. We use

classes defined in the Platys ontology to define hierarchical context models that enables us to generalize or specialize over user context. An example of how this is used is shown in section 4.3.

We use a knowledge-base on the phone that stores facts about apps including app categories. The facts are extracted from various sources like the Android Marketplace [28] and the DBpedia ontology [57]. The facts include meta-data like app manufacturer, download count, maturity rating, user rating, developer country of origin, number and category of permissions requested by the app etc. The facts about user context and apps are stored in form of RDF triples, which helps us query the knowledge-base for properties like app types or location types. These information enables the inference mechanism as the rules are stated in terms of the properties of apps and user context.

The final piece of information needed to make a decision are the rules for the current request meta-data, which are provided by the policy storage module. A requester, resource tuple can have multiple policy rules applicable based on contextual conditions. Once rules are obtained, using context and app facts from knowledge-base a specific rule applicable is inferred by an OWL-DL reasoner. The consequent of a chosen rule is the applicable action. If action is deny then a data request is marked as a possible violation of current policy rules.

In observer mode, the violation meta-data, which consists of a request meta-data along-with an applicable rule and user context is forwarded to User Policy Control module and no response is sent to policy enforcement module. In enforcer mode however,

action inferred by reasoner is simply returned to policy enforcement module to manage access to requested resource.

4.1.4 Policy Enforcement

For policy enforcement, MithrilAC has to be the system admin. We achieve the goal of inserting ourselves in between applications and Android framework and acting as an admin by using a custom ROM. Our solution uses ideas from the Privacy Guard project [38]. The policy enforcement module receives as input, data requests from apps and serves them with data as dictated by the "action" returned by policy decision module. In observer mode, policy enforcement module does enforce access control on the mobile device. In this mode it simply passes data request tuples consisting of a requested component name or type of data and a requester name (henceforth referred to as: request meta-data) to policy decision module. In enforcer mode, it passes on a request meta-data but expects policy decision module to provide an "action". If the action is to allow access, it simply makes a request to the Android framework for the data and returns the same to the requesting app. If action is to deny access, it prohibits request from going any further. Implementation details of policy enforcement module can be found in Chapter 6.

4.1.5 User Policy Control

Finally, we take a look at the user policy control module. This module is of key importance in this paper and will be discussed in detail in Section 4.3 but we provide

a brief overview here. As we have explained before, MithrilAC starts with an initial policy for a particular user category as defined by the occupation chosen by the user at installation time for MithrilAC. We collect user's identity and some basic profile information. This information includes user's identity, work location, home location, occupation category defined by our ontology etc. Using the policy control module we capture a user's preferred policy. We use an ontology to define contextual information using a hierarchical context model. We use Location and Activity generalization as was shown in our group's previous work [88] and discussed in Section 4.3.

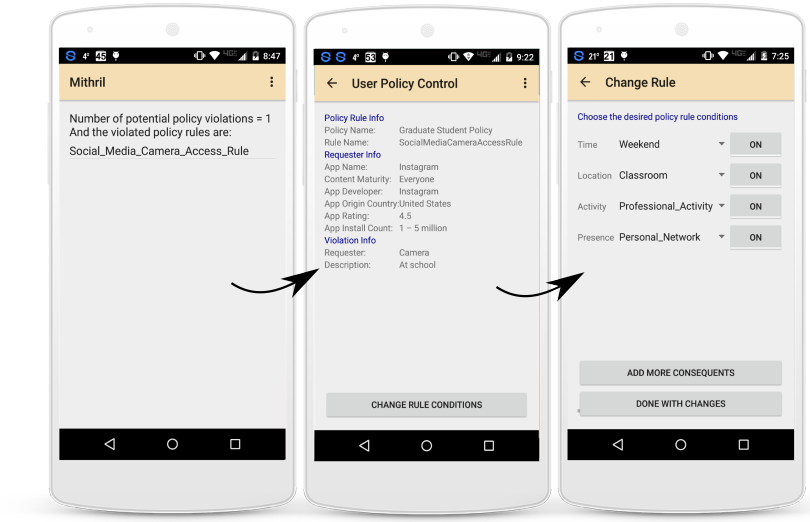

Figure 3: Transitions shown for prototype app

4.2 Use Case Scenarios

The use cases that we will discuss represent the possible scenarios we envision in our policy violation and user feedback process. We used CM13, a fork of Android 6.0.1 (Marshmallow) for creating the application that allows us to capture user feedback.

Before Android Marshmallow, we had a permission model of install-time permission acquisition for data access allowed to an app. In Marshmallow we saw the launch of run-time permission acquisition model. Point to note here is that we now have Android 7.0.1 (Nougat) out in the wild but from an access control model perspective, nothing has changed, so using CM13 is okay for now. However, we still do not have context-sensitive, fine-grained and dynamic access control in Android. In our running example, we have an initial policy for graduate students, i.e., GRADSTUDENTPOLICY that contains a few rules like the following:

- SOCIALMEDIACAMERAACCESSRULE: Do not share camera resource with social media applications at work
- SOCIALMEDIALOCATIONACCESSRULE: Do not share location with social media applications at work
- TOOLAPPSNETWORKACCESSRULE: Do not share network information with Tool apps
- PRODCUTIVTYAPPSIDENTITYACCESSRULE: Do not share identity with productivity apps

In our example scenario we will use the SOCIALMEDIACAMERAACCESSRULE for explanations. We are assuming that the user is a graduate student at UMBC's Computer Science department. In the OBSERVER mode, we mentioned earlier, MithrilAC captures violations of the current applicable policy. Now imagine that the user takes a picture at the university cafeteria during lunch hours and uploads to Instagram. Our system is

able to use the Platys ontology to determine that university cafeteria is "part_of" the university and therefore the user is at 'Work'. Using our app knowledge-base we are also able to determine that Instagram is a Social Media app. Since we have a rule that states that social media applications are not allowed location access at 'Work', we detect this as a violation of applicable policy. In the next user feedback cycle we present all such "violations" of the initial policy to the user. At this juncture we study the five use case scenarios that can happen in our system.

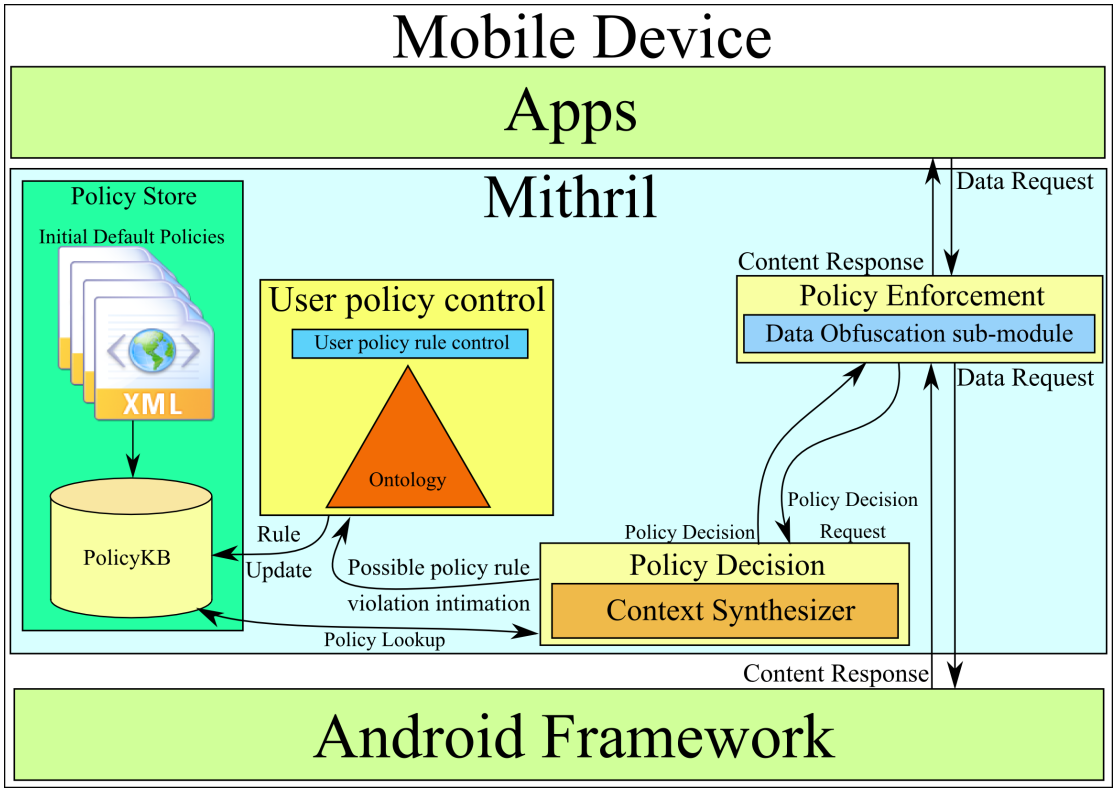

Figure 4: MithrilAC middleware architecture

4.2.1 Use case - True Violations:

"Rule is good, keep it". User is presented with a violation and the user determines that this was a **TRUE** violation. As stated before, this type of violation signifies that user did not expect observed behavior and the policy requires no change. In this case, the response we capture is used as a confirmation of the rule as being true. We will not ask the user about this rule again unless some sort of system wide reset happens. In this scenario we will enforce this rule as-is in ENFORCER mode. This use case signifies MithrilAC will now make "Do not share camera resource with social media applications at work" a quasi-permanent policy. By quasi-permanent we mean that the policy is not going to change unless there is an explicit system reset performed to go back to the initial policy applicable to the user category. This could happen if some static user profile information is changed that was collected at the install time.

4.2.2 Use case - False Violations:

The rest of the use cases state situations when policy modifications are required but we still have some variations in how modifications are carried out. The rest of the use cases explain these situations.

USE CASE FV-1: "Rule is not required, delete it". User is presented with a violation and the user determines that this was a **FALSE** violation requiring deletion. This scenario indicates that the user expected this behavior and thus the policy rule that causes current observation to be determined as a violation is no longer applicable. Similar to the above use case we will not ask the user about this rule again unless some

sort of system wide reset happens to the original default policy. In this scenario we will delete this particular rule and not enforce it in ENFORCER mode.

USE CASE FV-2: "Rule requires antecedent generalization, modify it". User is presented with a violation and the user determines the policy rule to be **FALSE** violation but an imprecise rule. That is the observed behavior although might not have been unexpected but the current rule does not clearly define the user's preferred policy. As a result, observed behavior cannot be clearly stated as a violation of user's policy. An example of such a scenario would be, our rule stated was "Do not share camera resource with social media applications at work". Observed behavior was Instagram, a social media app, was used at University Cafeteria. The cafeteria is inferred as a work location as it is part of the University. However, the user expects to use their mobile to take pictures during lunch. Therefore, the rule requires more conditions like a temporal restriction or a more precise location restriction or an activity restriction. Such restrictions would mean modification would be required for the rule antecedents or new antecedents would have to be added to the rule. As such, at this point we can have four different outcomes. Since the user determines the rule as imprecise, they are allowed to modify the rule. Modifications could include changing a specific contextual antecedent into a more generic contextual antecedent. See the app design diagram in Figure 7. Our example rule stated that "Do not share camera resource with social media applications at work". The 'at work' part of the rule for a graduate student profile is used to infer that a location related antecedent applies and University Campus is a 'work' location. Now, user may choose to make location antecedent into a more generic antecedent by

going up the relationship chain defined in our ontology. An example of such generalization would be that user wants to apply the camera restriction at city level. New generic rule now applies as “Do not share camera resource with social media applications in Baltimore county”.

USE CASE FV-3: “Rule requires antecedent specialization, modify it”. User could also choose to make the rule more specific. For example they could state that the rule applies only at the ‘University Lab’. The reasoner will be able to infer that a modified rule needs to be enforced at a more specific location than previously captured. In pretty much the same way as the above use case, user is allowed to choose a more specific location antecedent by parsing down the relationship chain in our ontology. The modified policy thus becomes “Do not share camera resource with social media applications at University Lab”.

USE CASE FV-4: “Rule has too many or is missing conditions, delete or add them”. The most interesting use case is that of adding or deleting antecedents to the rule. As we saw in Figure 2. We could have a situation where the rule only applies if it is official work hours or in presence of certain other people. In such a scenario, our system allows the user to add or even remove contextual and other antecedents to the rule. Thus allowing us to capture more fine-grained policies than previously possible. Our example policy for social media camera resource access thus becomes “Do not share camera resource with social media applications at University Lab between 9AM and 5PM on a weekday in presence of Advisor”. Such a rule can be captured by user feedback process only, thus justifying the need for our system.

USE CASE FV-5: "New rule is required". An extension of the above use case would be that user needs a new rule altogether. This option is also available to user through our system. The user may simply choose to start an empty rule and add new antecedents and state a consequent that captures some aspect of the user's policy that was not covered by the initial policy. This flexibility allows our system to be capable of defining policies with all possible combinations of our system's known antecedents. As a result, with proper feedback we will always be able to reach the user's preferred policy.

4.3 System Implementation

Since MithrilAC uses a feedback mechanism to iteratively modify policy rules, we need to take a look at the rule capture interface and process. We have implemented a prototype system that has four modules. The first module detects app launch and API call behavior. This is to determine when an app, for example, requests location update. The second module gathers contextual information. The third module intercepts the calls made by an app and either returns dummy responses or no response at all. Since Android Marshmallow no data return is an acceptable behavior and we take advantage of this feature. Thus achieving data privacy and security with low system instability.

The user policy control is the final part of our system implementation. Figure 5 shows the violation meta-data as seen by the user and Figure 7 shows the policy modification options that a user is provided during the process of capturing their preferred policy. We have also added a set of screenshots from our prototype app showing the

steps of rule capture (see Figure 3). A feedback iteration starts with a list of violations, obtained from policy decision module, being presented to a user. When the user chooses to look at a specific rule violation from a list they are presented with a specific rule's violation meta-data, which includes actual rule statement and a list of facts about an app that is violating a rule. User then has the option of further exploring the violation by clicking on "Display Policy Rule Conditions" button for exploring context antecedents for said rule.

The frequency of feedback is a admin or user setting. During each feedback iteration, the user is shown a list of all potential violations on their mobile device. As explained in previous section, user has two options at this point. They can choose to state a violation as a true violation or as a false violation. Our ontology and user context facts allows us to generalize or specialize over user's context. This provides a convenient way for user to modify policy conditions, in order to define changes in the current rules. We use two types of generalization: by location and by activity.

Location generalization in our ontology is achieved by using the transitive properties "is_a" and "part_of". The "is_a" property defines location classes as sub class of other location classes, for example a Home_Location could indicate the home coordinates of a user and it is a type of Location. The "part_of" property on the other hand represents the concept of one location being inside of another. Thus we have classes like Country, State, City, Organization, Building, Room, Point where each lower level class is *partOf* a higher level Location class. Using these classes and the part_of property we define a hierarchical location context model which then allows us to represent

axioms like “Room is a part of Building”. We use an OWL-DL reasoner [87] to infer existing relationships between instances of these classes.

Every activity in our ontology is the sub class of the Activity class. It is possible to obtain user activities, using Google APIs, which are related to a device’s motion. For example if a user is walking, running or in a car or not. On the other hand we can obtain user activity information from User’s calendar too. In our ontology we have classes defined like Professional_Activity, Meeting, Lab_Meeting, Professor_Meeting, Project_Meeting etc. We can see this class hierarchy from the ontology in Figure 6. This allows us to define a hierarchical activity generalization model via sub class relationships between generic and specific activities. Imagine a scenario where an app is collecting microphone data and we want to protect private lab information. We can then define policies for Lab_Meeting or we can define activity context based policies for any “Meeting”, if we want to prevent recordings at all of our meetings.

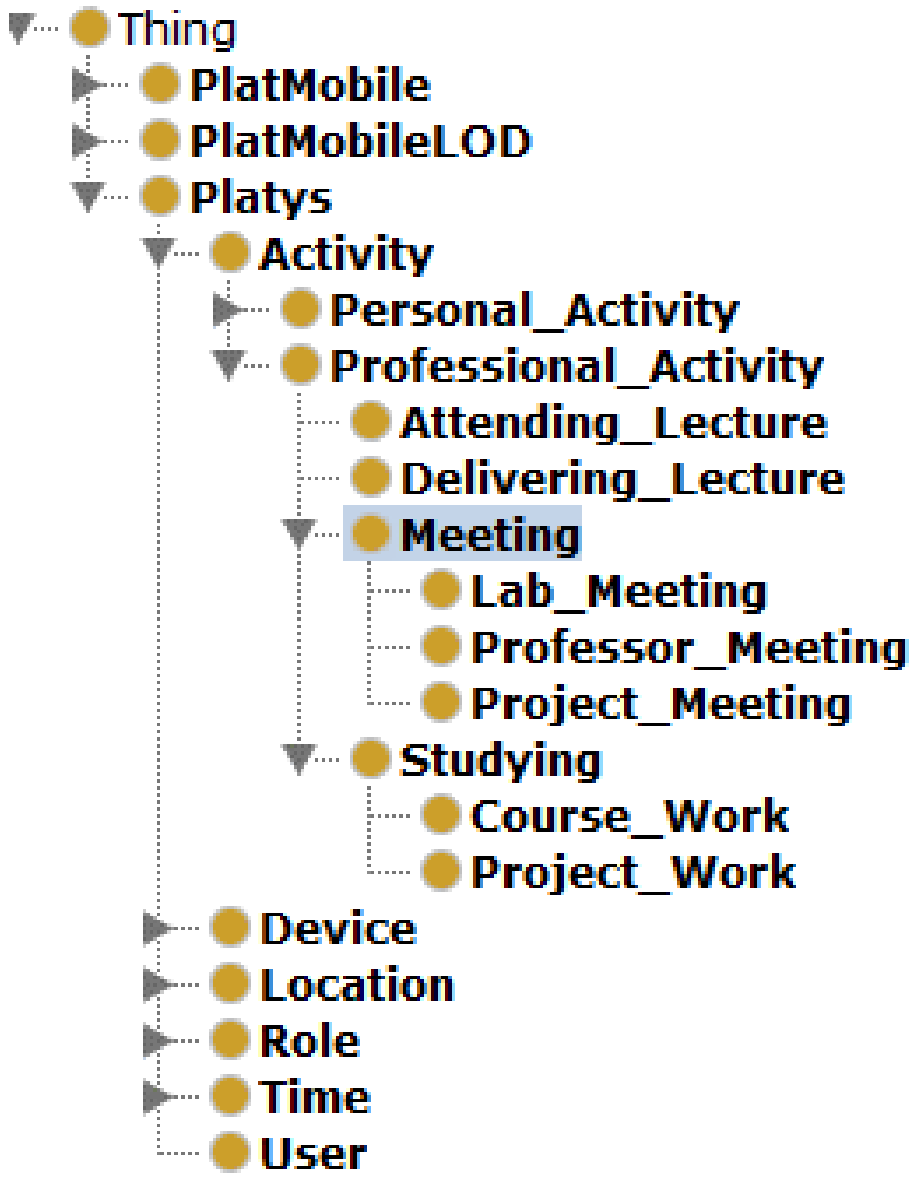

Figure 6: Snapshot of Platys Ontology defining context hierarchy

A sample view of hierarchical choices can be seen in Figure 7. Although we have discussed six use case scenarios that might occur during a policy capture process based on violation information presented to users, it is possible to have more use cases which might be beyond even the violation capture process. One such scenarios would be when

a policy rule's consequent is modified. This could either negate our initial rule or may add conditions on what data could be shared. Such a condition might include data obfuscation techniques. In this case user will have to add antecedents that define those limitations. For example, the user might want to share a fake Location, an inaccurate location or state that location is unavailable.

Clearly, our policy rules are significantly more complicated as opposed to a simple permission based model that Android currently follows. The dynamic nature allowed by the variable actions and the granularity provided by the contextual antecedents are contributing factors to this complexity. However, it also gives more control to the user over her data.

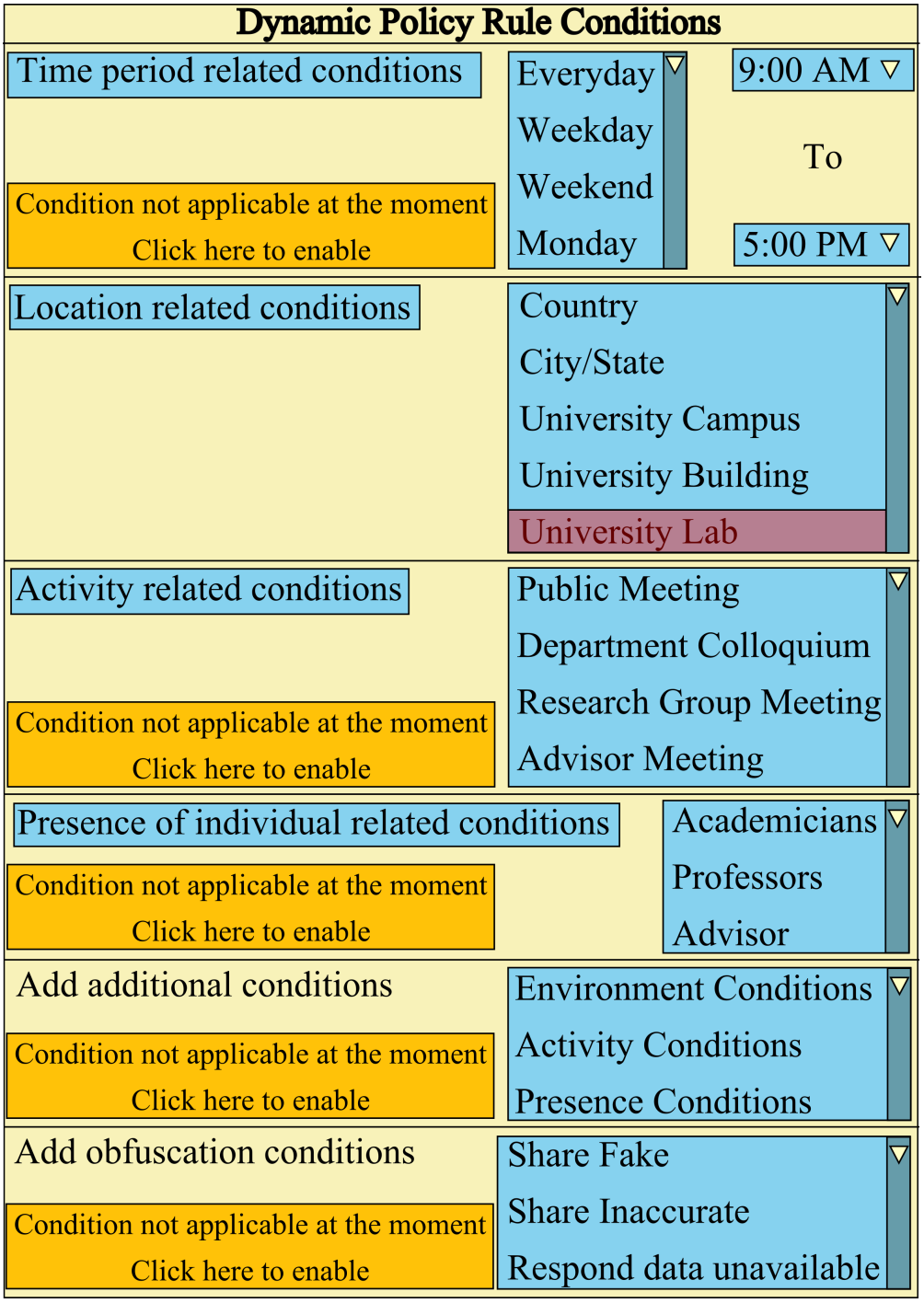

Figure 7: Ontology-driven hierarchical options for rule modification

Towards automatic rule generation: The use of a hierarchical context model via an ontology allows us to infer subsumption relationship between a generic and a specific

rule. As a result, we can use an ontology to infer decisions for contextual situations, for which no "specific" rules exist. For example, if there is a rule that states "Do not allow access to camera at work". That means any location that can be determined to be a work location can be assumed to be a place where camera access is not allowed. However, once the user modifies this rule to a more specific rule stating "Do not allow access to camera in university Building" what can we assume about the locations that are still work and were part of the rule but are not anymore? Can we generate more policies that state "Do not allow camera access in university parking lot"? We can discard the rule that states access denied at cafeteria as we observed the user's response to that specific violation but what about the other conditions? Given that our ontology defines the state of the world, we can possibly generate all such conditional rules using our hierarchical context model. In a similar way, if we observe, as per our contextual model, rules being added for every sub class or context piece at a particular hierarchy then we can infer a more generic antecedent for the specific context piece and reduce the rule set to a smaller set. To the extent of what context we can capture in this hierarchical manner, we may generate rules for a user automatically. Given our system design and an initial policy for a user category, we can carry out the reduction and expansion of policies into a bigger and smaller set of rules. However, such reduction or expansion is beyond the scope of the current paper.

In the next chapter we talk about app analytics back-end that helps produce an initial default policy for MITHRIL.

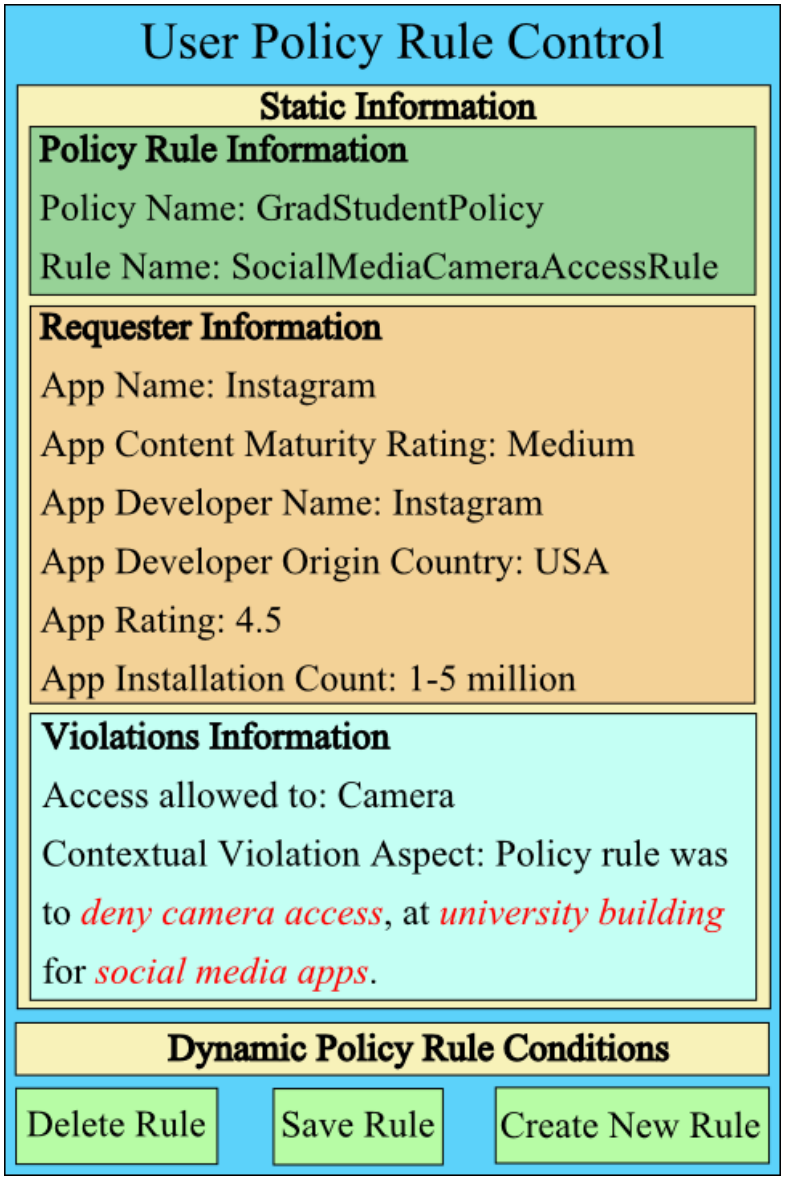

Figure 5: Rule violation meta-data displayed to user

Chapter 5

APP ANALYTICS BACK-END

In this chapter, we describe the functionality of the app analytics back-end. The key research question we are trying to answer through this system is as follows:

RQ2: Can system calls be used as features to classify mobile applications into their behavioral classes?

5.1 App analytics system design

We call the system/pipeline we built to carry out our behavior classification task, the Heimdall system (see Figure 1). We have five main components in our system: Download module, Annotation module, System call module, Feature generation module and Classification module. The input to our system were search terms for testing app categories. The expected output of the system was behavioral class for an app.

A system call (or syscall) may be defined as the fundamental interface between an application and the Linux kernel [46]. The system calls that are part of Android's kernel distributions have been defined in the class android.system.Os [32]. At the lowest level of the operating system, an app's functionality boils down to the tasks and services it requests the kernel to perform, through system calls. As a result, an app could be monitored by observing the patterns in the system calls it executes.

What does a behavioral class represent? A simple representation of behavioral

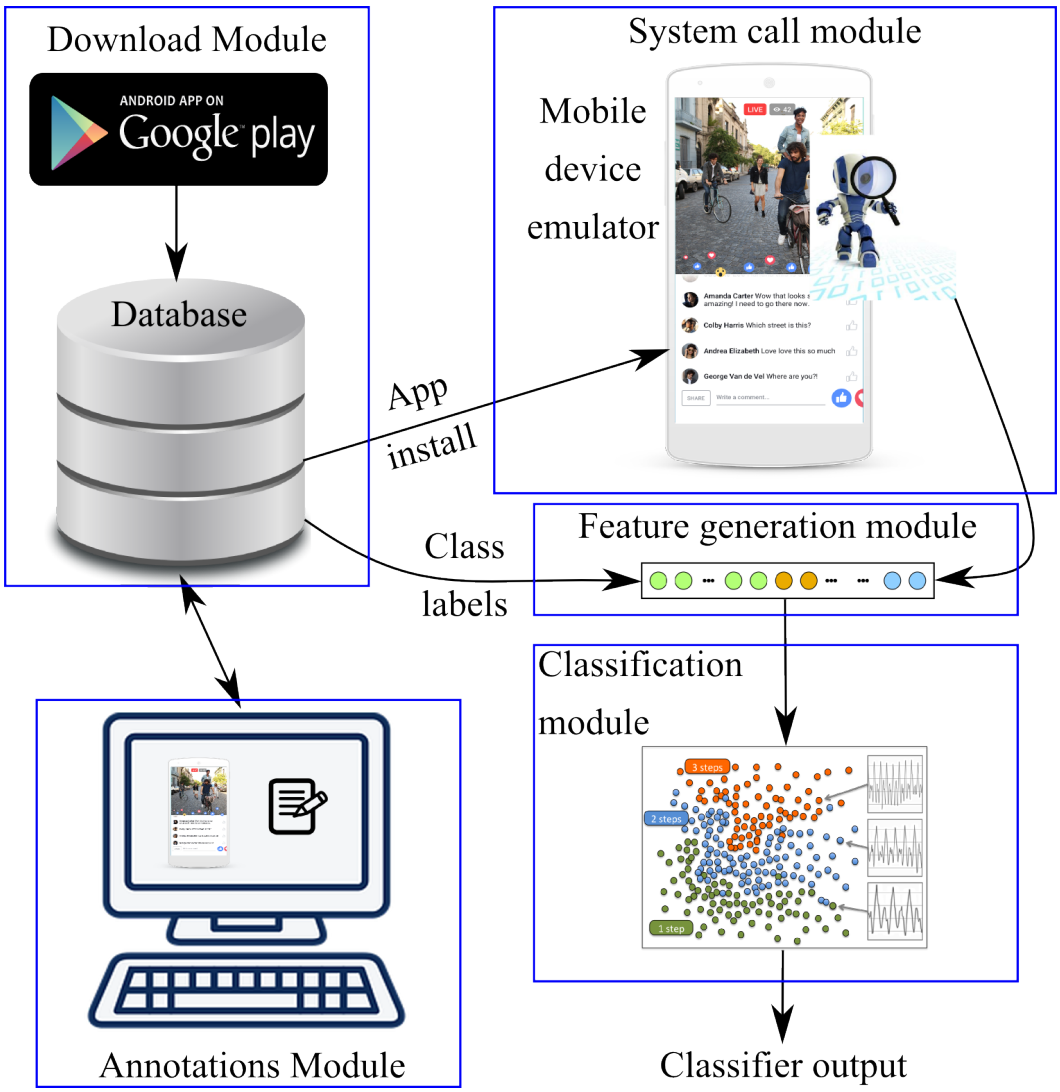

Figure 1: Design of system built for studying app behavior

classes maybe considered as the app category information from Google Play Store. More appropriate category information would be the ones we determined during our behavioral pattern annotation. For example, we found a number of mobile applications that were PDF readers. In order to carry out the annotation, we downloaded all the mobile applications that we could find for this particular behavior category, i.e. PDF readers, on a mobile device. We used the mobile applications and read the mobile application's description on Google Play Store [28] and manually determined the app's

primary usage category. Google categorized most of these mobile applications into either Productivity, Books & Reference or Education categories. Such a discrepancy indicates that determination of a mobile application's category was a complicated task in reality.

The `strace` utility enables diagnostics, debugging and instructional user-space monitoring and modifying interactions between processes and the Linux kernel, which include system calls, signal deliveries, and changes of process state. We have used this utility for studying and capturing system calls in form of interactions between user space mobile applications and kernel space programs.

Practical limitations necessitated usage of an emulator for experiments running simulation of user's actions on a mobile device. For that purpose, we used the "UI/Application Exerciser Monkey" [33] (hereafter called Monkey tool). The Monkey tool is a command-line utility that can run on a emulator or mobile device and sends a pseudo-random stream of user events into the system. This utility allowed us to write programs that controlled the Android device or emulator to install a list of mobile applications that were to be tested and exercise all possible UI behavior that a user could trigger.

Once the system calls were captured, we used standard text processing techniques and information retrieval measures like simple term-frequency and tf-idf to generate feature vectors for our classifiers. These feature vectors were then used to run a slew of classifiers using the weka tool [36]. We have presented the classification results in Section 4.2.

5.2 Machine Learning pipeline setup

Our experiments were executed using an emulated Nexus 6, running Android 6.0.1 (Lollipop build December 2015) with Intel's hardware accelerated execution manager, 1.5GB RAM and 16GB internal storage. The host machine operating system was Ubuntu 14.04, Intel Core-i7 3.4GHz processor, 32GB RAM and 2TB storage space for downloaded apps. The machine learning pipeline was made up of five modules described next.

5.2.1 Download module

There are several mechanisms for downloading Android mobile applications from the Google Play Store [28]. We used open source code [1] with our system specific modifications for this task. Although our system could be used to study any kind of app and its functional behavioral pattern, for the sake of simplification of experimental evaluation, we started our system with 10 specific search criterion on the Google Play Store [28]. A search on the Play Store [28] could be performed using a simple html GET request with the search term as a url parameter [2]. The task of the download module was used to retrieve both metadata about mobile applications (i.e. app name, developer name, descriptions, Google Play category etc.) and Android executable APK files, to run experiments for mobile applications found through the search results.

[1] CMUChimps Lab: https://github.com/CMUChimpsLab/googleplay-api

[2] URL Prefix: `https://play.google.com/store/search?q=`; Search terms: `pdf readers`; URL Suffix: `&c=apps&hl=en`

5.2.2 Annotation module

The annotation module included an interface to read the app description and other meta information and an emulator to install the app and observe its behavior. Based on their observations annotators would ascertain a specific "behavior class" and assign it to the app. We ran our study on 534 mobile applications from 10 specific keyword patterns. The key word patterns we used include: "alarm clock", "file explorer", "to do list", "scientific calculator", "battery saver", "pdf reader", "video playback", "lunar calendar", "drink recipes", "wifi analyzer". We downloaded 1560 mobile applications found in our search. However, a significant number of these apps, were unusable due to emulator issues (app crashes and incompatibility issues) or because they required some sort of user interaction that could not be automated (for example, profile creation). As a result, we annotated 534 apps.

Annotated behavior class	**# apps**	**%age**
Alarm clock	128	23.97
Battery saver	93	17.42
Drink recipes	15	2.81
File explorer	72	13.48
Lunar calendar	12	2.25
Pdf reader	22	4.12
Scientific calculator	61	11.42
To do list	102	19.10
Video playback	5	0.94
Wifi analyzer	24	4.49

Table 5.1: Annotated app categories

Table 5.1 shows the distribution of mobile applications annotated according to their behavior classes. It is interesting to note that for these 534 apps, Google puts them mostly into Tools and Productivity category as shown in Table 5.2. We can con-

Google Play category	# apps	%age
Tools	265	49.63
Productivity	133	24.91
Lifestyle	48	8.99
Education	14	2.62
Personalization	13	2.43
Books & reference	12	2.25
Music & audio	8	1.50
Entertainment	7	1.31
Communication	6	1.12
Health & fitness	6	1.12
Business	5	0.94
Media & video	4	0.75
Medical	3	0.56
Adventure	2	0.37
Social	2	0.37
Travel & local	2	0.37
Arcade	1	0.19
Libraries & demo	1	0.19
News & magazines	1	0.19
Shopping	1	0.19

Table 5.2: Google Play Category

clude from this observation that not only does Google *NOT* maintain a standardized approach to ensure that a developer explain what their app does, they categorize mobile applications in a very generic fashion. Granular behavior categorization thus remains a motivating challenge for further research.

5.2.3 System call module

In the system call module we installed downloaded mobile applications on an Android emulator. We then used the Monkey tool [33] to simulate a real human using an app and all its functionality. We used the monkey tool to adjust the percentage of "system" key events (like Home, Back, Start Call, End Call, or Volume controls) and

```
2966  read(37, ''Android Emulator OpenGL ES Trans''..., 65) = 65
2966  read(37, ''A\0\0\0'', 4)                    = 4
2966  write(37, ''\23'\0\0\24\0\0\0\0\37\0\0\0\0\0\0\0'', 20) = 20
2966  read(37, ''\34\0\0\0'', 4)                  = 4
2966  read(37, ''\344\377\377\377'', 4)           = 4
2966  write(37, ''\23'\0\0000\0\0\0\37\0\0\34\0\0''..., 48) = 48
2966  read(37, ''Google (NVIDIA Corporation)\0'', 28) = 28
2966  read(37, ''\347\377\377\377'', 4)           = 4
2966  read(37, ''\31\0\0\0'', 4)                  = 4
2966  write(37, ''\23'\0\0-\0\214\213\0\0\31\0\0''..., 45) = 45
2966  read(37, ''OpenGL ES GLSL ES 1.0.17\0'', 25) = 25
2966  write(37, ''\23'\0\0\24\0\0\0\214\213\0\0\0\0\0'', 20) = 20
```

Figure 2: System calls

maximize coverage of all activities within the app's package. We varied the number of clicks through Monkey between 1000 and 10000 to maximize coverage of "visible" functionality of an app. Throttling was the final option that we used to ensure stability of execution. The final option made sure that we had fewer app crashes. `strace` was used to capture system calls generated by the process of an app. We used the Android Debug Bridge to control the emulator and extract the results of our experiments.

5.2.4 Feature generation module

The output of the previous module was a series of system calls made by the app. An excerpt of strace output for an app from the "File Explorer" category is shown below:

The first part of each line in `strace` output was the app's process id. After that we have the system call followed by parameters for that particular system call. In our study we collected frequency of system calls made by an app in order to generate features. "Term frequency–inverse document frequency" (tf–idf) [73] is one of the most commonly used term weighting schemes in information retrieval. We compute tf–

idf weight vectors using system calls as terms and mobile applications as documents:

$tfidf(t, d, D) = tf(t, d) \times idf(t, D)$

where, Term Frequency (TF) was computed as: $tf(t, d) = 1 + logf_{t,d}$

and Inverse Document Frequency (IDF) was computed as: $idf(t, D) = log\frac{N}{n_t} \Rightarrow idf(t, D) = log\frac{N}{|\{d \epsilon D : t \epsilon d\}|}$

Here, 'N' represents the total number of documents in the document set 'D'. 't' represents a term in a specific document 'd'. 'f' represents the frequency of a term 't' in a document 'd'. Finally 'n' represents number of documents 'd' with term 't' in document set 'D'. We generated two sets of feature vectors–one hot vectors and tf–idf weight vectors. We ran the classifiers to train and test on two sets of ground truth labels–annotated class labels and Google Play categories as class labels.

5.2.5 Classification module

The classification module consisted of scripts to use the Weka tool [36] and run Support Vector Machine(SVM), Naive Bayes(NB), Decision tree(J48) and Multilayer Perceptron (MLP) classifiers on our generated feature vectors. For each of these classifiers we used 10 fold cross validation technique and recorded the achieved average F1–measure [81]. F1–measure is the harmonic mean of precision and recall and has a range of [0,1]. For SVM we experimented with RBF and Polynomial kernels. Detailed results are in Chapter 8.

5.2.6 N-grams of system calls

When an app performs an action at the Java API level it can result in a sequence of multiple system calls due to abstractions at the API level. These sequences may have structural patterns the way human languages do. For example, in English, the letter 'u' is commonly followed by the letter 'q,'. Since n-grams work by capturing this structure, certain combinations of system calls can perhaps be captured by them and may be representative of the app's behavior class. In order to test this theory, we used our generated system call sequences and generated n-grams from them and then performed the behavior classification task again. The results of the system call n-gram feature based classification are presented in Chapter 8.

5.3 Malware classification

Malware classification is a well researched domain [7, 19, 89]. One of the causes of focusing on behavior classification, in our work was that the malware samples that are available through various sources today, like the Malware Genome Project [89] and VirusShare [69] are quite old. A lack of good sample data should make it difficult for detecting malwares and accuracy of such a task should ideally be low. However, since benign app samples are readily available from official app stores, we wanted to determine if classifiers perform well by capturing the latent temporal distinction in the data sample. This is significant because although the state-of-the-art in malware detection [86] has achieved 96.76% accuracy, due to the quirks of the data set, we argue, that these malware detection projects are capturing features that are in-fact classifying

mobile applications that use Android APIs from different time periods.

Permission based analytics have shown some promise in malware classification [7, 12]. However, our primary goal was not to perform malware detection but rather using feature importance to determine the permissions that are significant in detecting a malware. We use these permissions to then compute the similarity between known malware applications and applications we encountered on the Google Play Store.

5.4 Mobipedia

In the Mobipedia project [63] we created an ontology Figure **??** which models concepts related to mobile applications. The project parsed data from various online sources to consolidate knowledge about mobile applications into the Mobipedia Knowledge-Base. Mobipedia's ontology [63] models information related to mobile applications independent of the mobile operating system. We considered using existing ontologies such as Dublin Core Metadata Initiative (DCMI) and Description of a Project (DOAP) which are used to describe web resources and software projects respectively. But as neither of them is focused on mobile development, the concepts and properties in those vocabularies do not match the requirements for modeling of mobile mobile applications completely. Nevertheless, we linked some of the terms in DCMI ontology with Mobipedia terms using `owl:subClassOf`. For example, "DCMI:Creator" [84] was linked to "Mobipedia:Developer".

Figure 3 shows an excerpt of the ontology including the most important classes

and the object properties that relate them [3]. In addition to the classes we had defined in the Mobipedia project, as part of this dissertation we added the `mobipedia:Behavior_Categor` class, for defining category of application behavior.

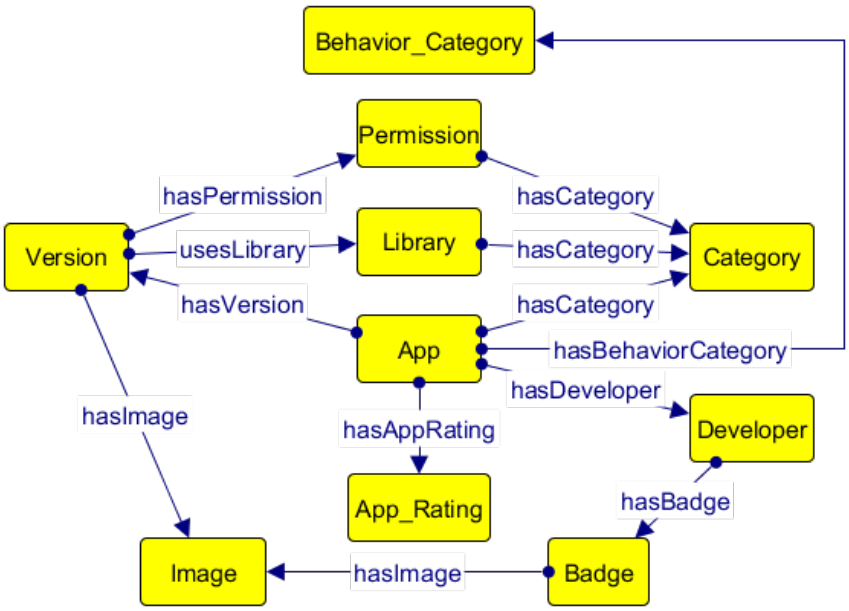

Figure 3: Excerpt of the Mobipedia ontology.

5.4.1 Adding behavior knowledge into Mobipedia

In this dissertation, we used static and dynamic analysis of mobile applications to perform a multi-class classification task that determines an application's behavior class. Since we had already created the Mobipedia KB we incorporated the an additional class defining a mobile application's behavior class. We enrich the KB with the results of our behavior classification task.

[3]The figure has been generated using the Graffoo specification [21].

5.4.2 Accessing Mobipedia

Access to Mobipedia is royalty-free under the terms of GNU free documentation license. Similarly to DBpedia [3], we provide three mechanism of accessing the Mobipedia dataset:

Linked Data. It uses HTTP protocol to retrieve entity information which contains all the triples associated with the entity. This can be accessed using web browsers, Semantic Web browsers, and crawlers.

SPARQL Endpoint. We have also setup a SPARQL endpoint at `https://mobipedia.science/sparql` which can be used for querying the Mobipedia dataset.

RDF Dumps. Larger versions of the Mobipedia dataset in the form of serialized triples can be downloaded from the Mobipedia website as well. These dumps can be used as annotated datasets in research or for the purpose of running various analyses locally.

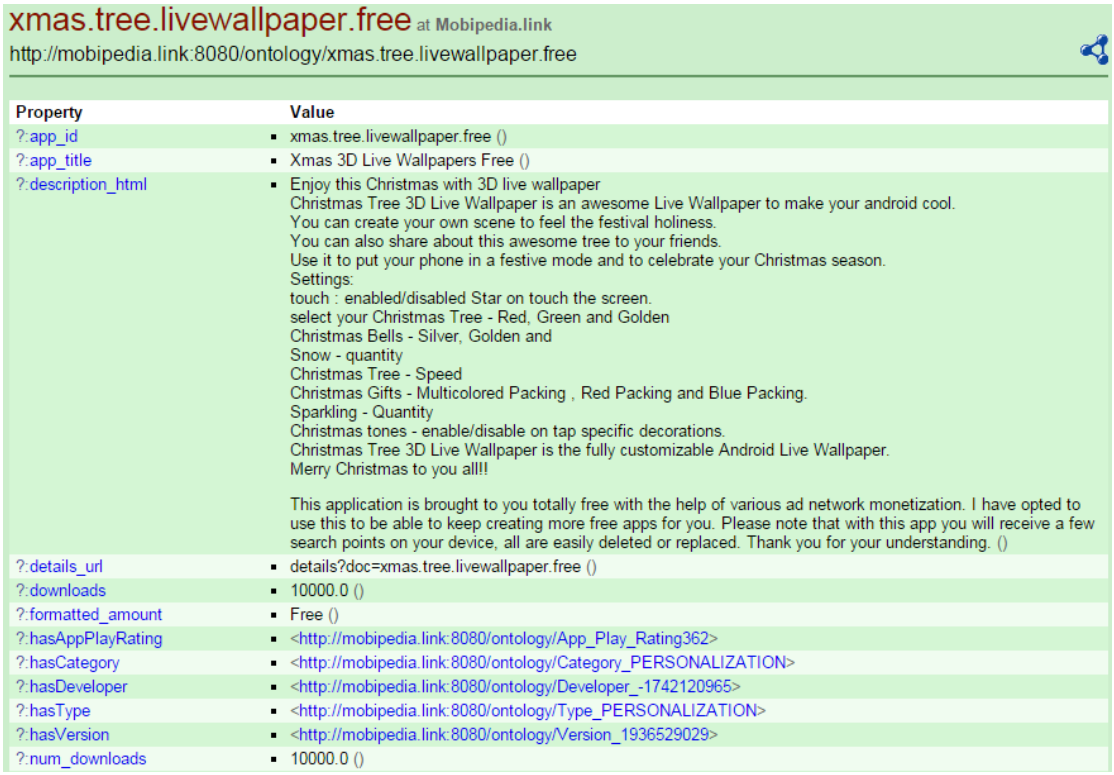

Figure 4: Linked Data interface of Mobipedia as seen in a web browser.

Mobipedia is an evolving project due to the dynamic nature of mobile apps: New mobile applications or versions of existing mobile applications are published every day. One of our goals, when we first started this project, was to involve the integration of other published data sets such as the *Android Malware Genome Project* [89], which contains information about mobile malware applications. Adding the behavior category of an application is a step towards that goal.

Chapter 6

CHALLENGES OF POLICY EXECUTION

Once the policy capture process is complete, we have to start executing the rules to protect user privacy and security. We take a look at our rule execution mechanism in this chapter.

We have described the form of a SWRL rule in Chapter 3. In order for a SWRL rule to execute, we have to verify that the antecedents of the rule hold true. If so, the consequent of the rule will be triggered as a true fact and entered into the knowledge-base. However, when it comes to a mobile device we have to actually take an action that is described by the rule. In order to do so our rule execution system, that is an internal module of the MithrilAC middleware, keeps track of app activities. Keeping track of context changes has been tested by [26, 59] and shown to have major energy implications. In order to avoid high energy consumption, which is important on a mobile device, we focus on app activity detection. It is interesting to note that using this technique our middleware consistently consumed less than a single percentage point of the mobile device's energy budget (see Chapter 8). When an app is launched our middleware is able to detect the actions taken by the app, i.e. the query made by an app. Finally, only when an app and action pair is detected, we take a snapshot of the current context using our internal context synthesis module and determine a high-level Semantic user context. At this point, we have all parts of the triple (R, C, Q) and we can

query our knowledge-base to discover applicable policies. If an action taken by an app in the current context violates any of our captured access control rules, we then block the action from being performed.

The description above might look simple enough but there are quite a few challenges in performing these tasks. For example, we described detecting app launch but since Android Lollipop 5.0 getting app launch has become complicated due to the deprecation of the getRecentTasks API. Android did this because this API could potentially leak personal information of the user. However, we have successfully accomplished the vision of performing context-dependent, fine-grained access control policy execution on Android.

6.1 Android security mechanisms

Over the years there have been a lot of projects, mostly from the open-source domain, that were designed to control things on a mobile device. Some prominent example include XPrivacy [4], Privacy Guard [38] and OpenPDroid [56]. These projects are all focused on Android because Android because policy execution would essentially require administrative privileges and Android being open-source, it is possible to modify system behavior. The XPrivacy [4] project is a module of the XPosed [70] project. XPosed modules act in the same context as the Android Zygote process. The Android Zygote process is similar to the Linux init process. Just like in Linux every application starts as fork of Zygote. An XPosed module can thus act as the initiating process for every application and thus is able to control its behavior. This was an interesting

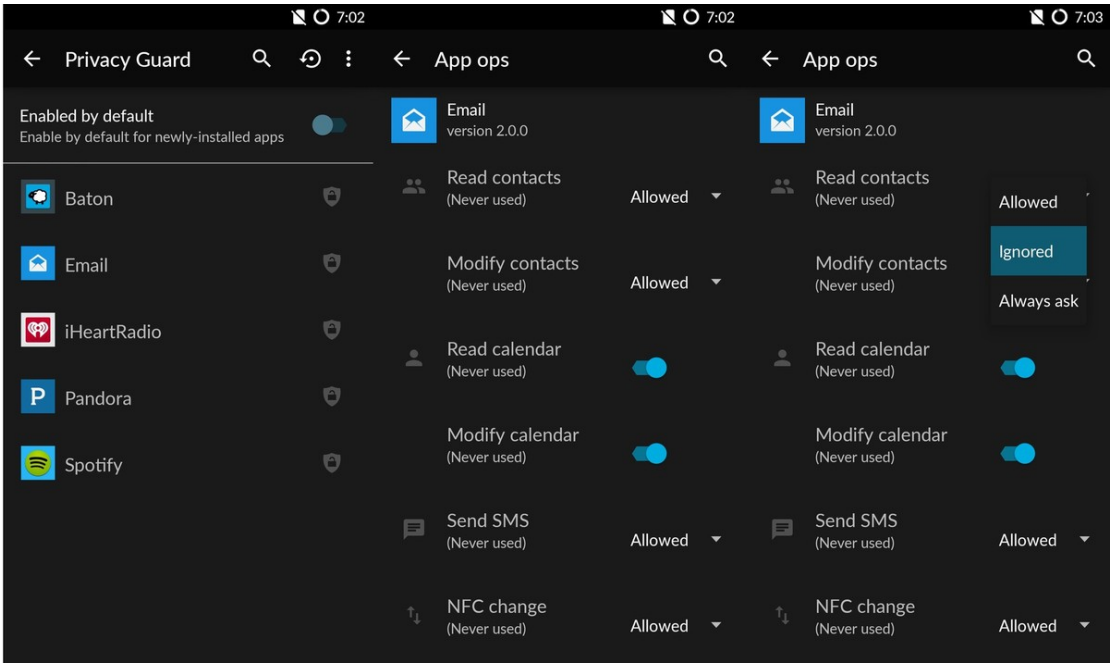

Figure 1: Settings of Privacy Guard

solution but the project stopped development after Android Marshmallow 6.0. As an alternate solution, we came up with an extension of the Privacy Guard project. The Privacy guard project takes advantage of the AppOps API and blocks access to resources on the mobile device Figure 1. It is imperative to note that none of these project implement context-dependent access control. As in, if the user has a different policy at home from what they have at work, these systems would not be able to handle that.

Android Marshmallow 6.0 made it possible for user's to block access to their data through run-time permission acquisition Figure 2. Although an obvious step towards better access control on mobile device, it still did not allow dynamic access control. The advantages of context-dependent access control have been explained in Chapter 3. It is true that an expert user can possibly go through the five step process on Android devices to access the permissions screen Figure 3 to modify their choices, however, in reality

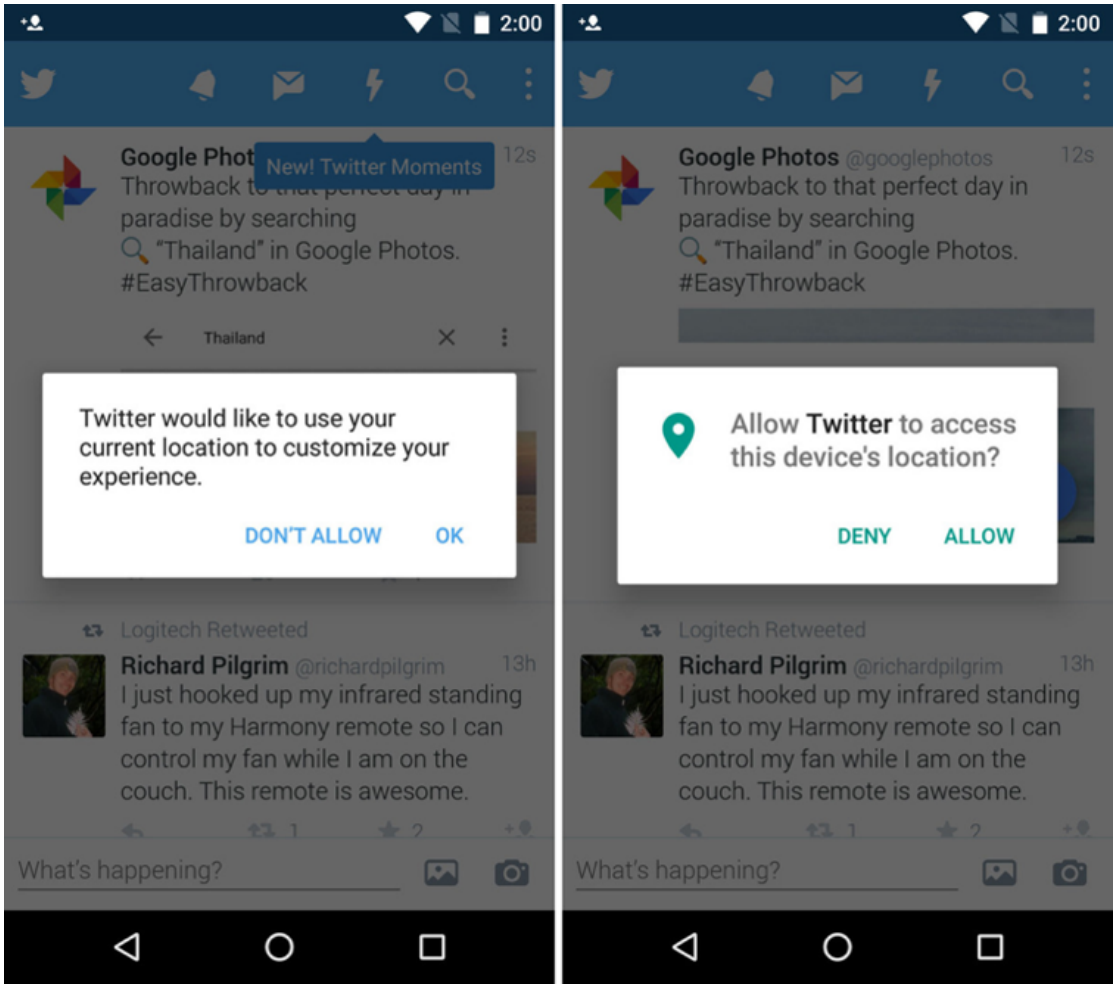

Figure 2: Permission screen for a specific app

these steps are too complicated for normal users, perhaps even for fairly competent users. Additionally, even if they do make these changes, it is tedious to do so if the choices are context-dependent. Our policy execution mechanism alleviates the need to manually make these changes.

6.2 Enhancements in policy execution

There are two different ways that can be used to detect mobile activity on an Android device. Method one requires root privileges and reads the system log to detect the launch of an Activity by monitoring the ActivityManager class using the following command through a shell:

```
logcat -d ActivityManager:I *:S
```

We used the second method; i.e. the Usage Stats API that provides access to the device's usage history and statistics. However, getting the information about what activity an app performed is still not accessible through this API. For that reason, we had to use the AppOps API. It is important to note that although using the AppOps API we may control the mobile device, getting access to this API is challenging. Moreover, the AppOps API also does not implement context-dependent access control.

6.3 Challenges and solutions

The AppOps API is a hidden API and unavailable in the standard Android SDK. Which means our first challenge was that our code for the MithrilAC middleware would not even compile, if we used the public SDK. To access these hidden APIs without using Java reflection the simplest process is to extract it from a real device using the command:

```
adb pull /system/framework/framework.jar
```

The framework.jar is the runtime archive containing all the classes that are used in the android.jar in the SDK. However, the jar contains the runtime optimized version

DEX format files. So we have to use dex2jar [60] to convert them to class files by using the command:

```
dev2jar classes.dex
```

Next we extract everything from our target SDK (in our case API version 25) version's android.jar file from the path `ANDROID_SDK/platforms/android-25/` to a folder and overwrite it with the classes from the framework.jar file folder. Finally we compress the modified classes into a jar file to obtain our "hacked" Android SDK android.jar file with access to the hidden AppOps API.

The second challenge was to detect app actions, as in what resources the app used. Android's documentation does not explain this, but we were able to discover that we can use the AppOps API to detect app actions. Hidden APIs in Java are often accessed using reflection. However, reflection can be tedious and slow [13]. As an alternate solution we were able to use the above-mentioned "hacked" SDK and root privileges to gain access to app actions.

The third task was to actually execute the policies. There are a few ways to do this. The first would be to create an app like Xprivacy. The second would be to install Privacy Guard and use root privileges to modify the settings. However, we decided that third and the best way to do this was through AppOps, since it was provided by Android as an advanced access control tool in Android 4.3. As we explained in Chapter 2 there are signatureOrSystem permissions in Android that are granted only to applications with the same certificate as the application that declared the permission. There are certain permissions that are only granted to apps that are signed by a *platform key*. Two such

permissions control access to the AppOps API:

```
<!-- @SystemApi @hide Allows an application to collect battery statistics -->
<permission android:name="android.permission.GET_APP_OPS_STATS"
    android:protectionLevel="signature|privileged|development" />

<!-- @SystemApi Allows an application to update application operation statistics.
     Not for
     use by third party apps.
     @hide -->
<permission android:name="android.permission.UPDATE_APP_OPS_STATS"
    android:protectionLevel="signature|privileged|installer" />
```

The GET_APP_OPS_STATS permission can be "granted" using root access through a shell however, the second permission, UPDATE_APP_OPS_STATS is the one required to execute policies on the device. It cannot be granted to an app even through a root shell. In order to resolve this we had to compile our own Android ROM. We used the LineageOS Android source for our custom ROM. We did discover that simply compiling a custom ROM won't make our middleware a privileged application. There was one final step that we had to take to ensure that our custom ROM would in-fact be able to execute the policies that we capture. Every app that is compiled as part of an Android build has to be placed in the `packages/apps/application` folder at the root of the Android source directory and a Makefile has to be present in the application's folder that defines rules for the build process. The final step is to add a rule that defines our middleware as a privileged application for the system as follows:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE := MithrilAC
```

```
LOCAL_CERTIFICATE := platform

LOCAL_SRC_FILES := MithrilAC.apk

LOCAL_MODULE_CLASS := APPS

LOCAL_PRIVILEGED_MODULE := true

LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)

include $(BUILD_PREBUILT)
```

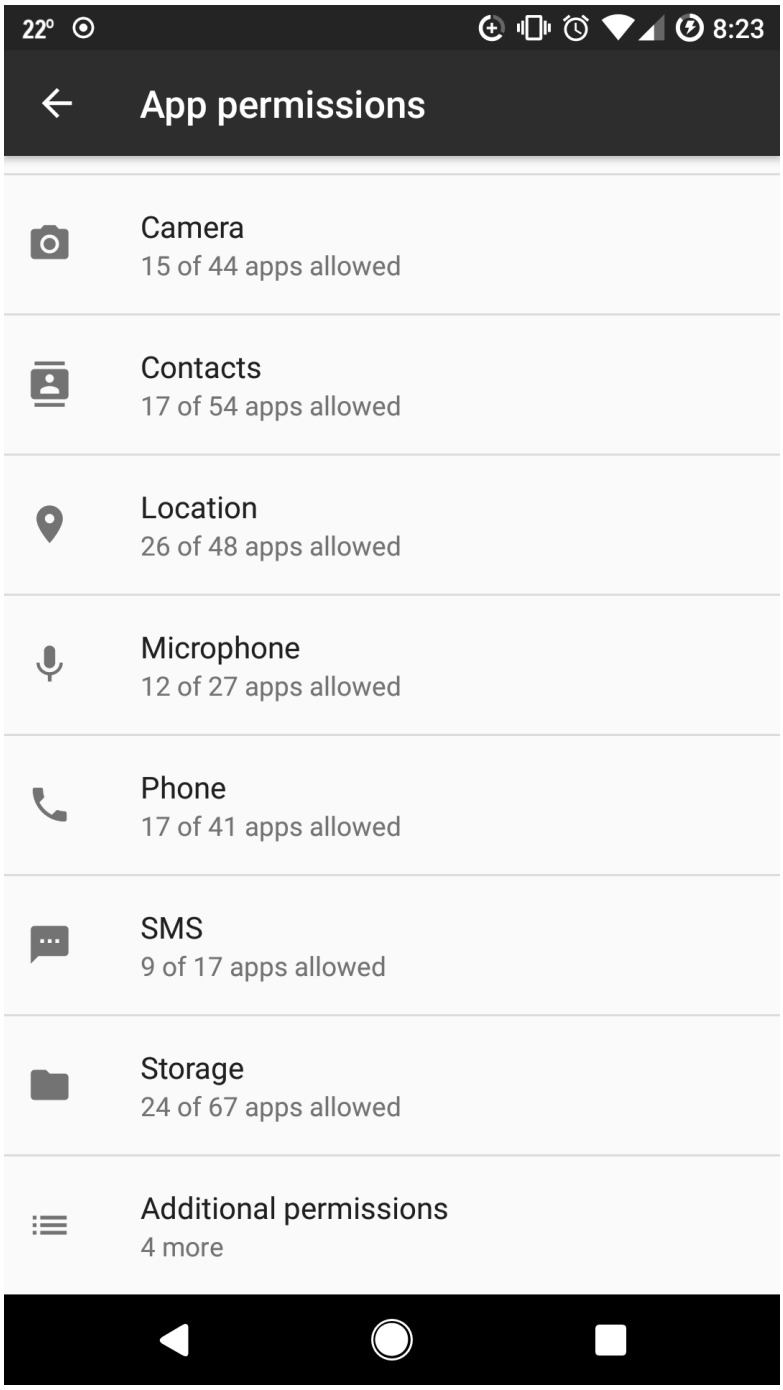

Figure 3: Permission settings on Android Nougat 7.1.2

Chapter 7

USER-STUDY CHALLENGES

In the previous chapter, we described the various implementation challenges for an access control system and making it work on a popular mobile operating system like Android. In this chapter, we will look at the challenges faced while performing our user study.

The MITHRIL user study was conducted for one and a half months during which period rooted mobile devices running a custom LineageOS Android ROM was provided to users. The study was conducted with 24 users and each user used the system for a period lasting at least seven days and at most 30 days. To detect an app's launch and resource consumed by the app requires special privileges and access to certain Android APIs that are unavailable on official versions of Android. This is why we had to use LineageOS, which can be used to enable access to APIs that are inaccessible on an official Android build from Google.

7.1 Default deny policy

We performed the user study in two rounds. The first round of our study did not use a curated policy and assumed a default deny action for everything. As one might guess, this caused a problem for the users who were participating in our study. Upon installation, the MithrilAC middleware started detecting a lot of "violations". These

violations were not necessarily true violations as per our definition from Chapter 3. We discuss the results of our user study in the next chapter. It quickly became evident that a user would find it difficult to provide feedback to all the scenarios that our middleware was presenting to them.

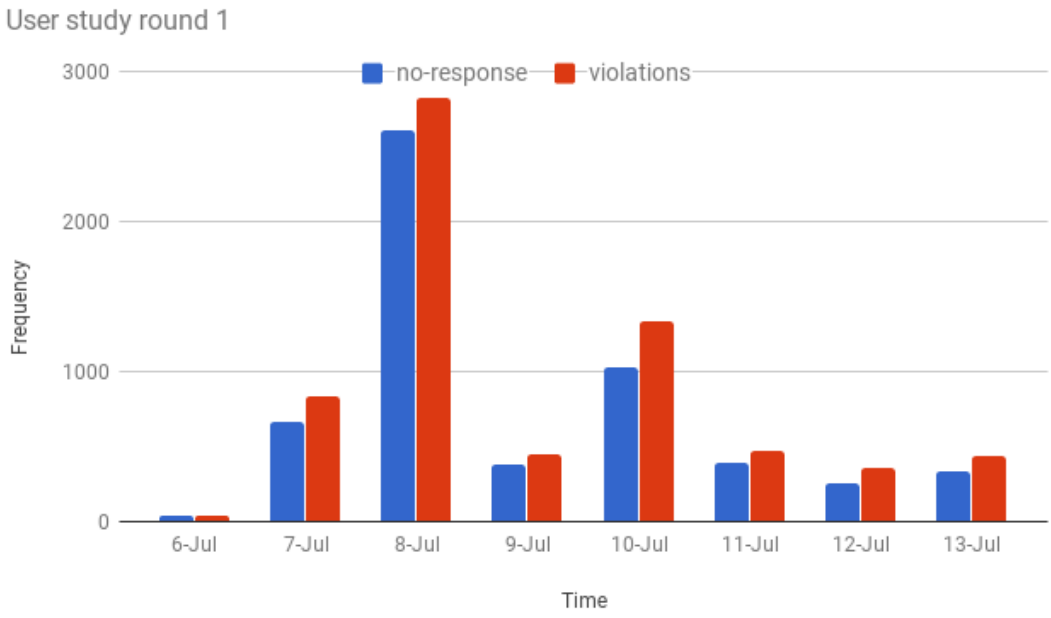

Figure 1: Comparing #violations and #no-response in user study round 1

7.2 Crowd-sourced policy

Following the issues of round 1 of the user study, we decided to use our crowd-sourced data to generate an initial default policy. We restricted the initial policy generation for just a few contextual situations. Essentially we wanted to show through the second round of our user study that it is feasible to use the violation detection method-

ology to determine policy deviations for specific users.

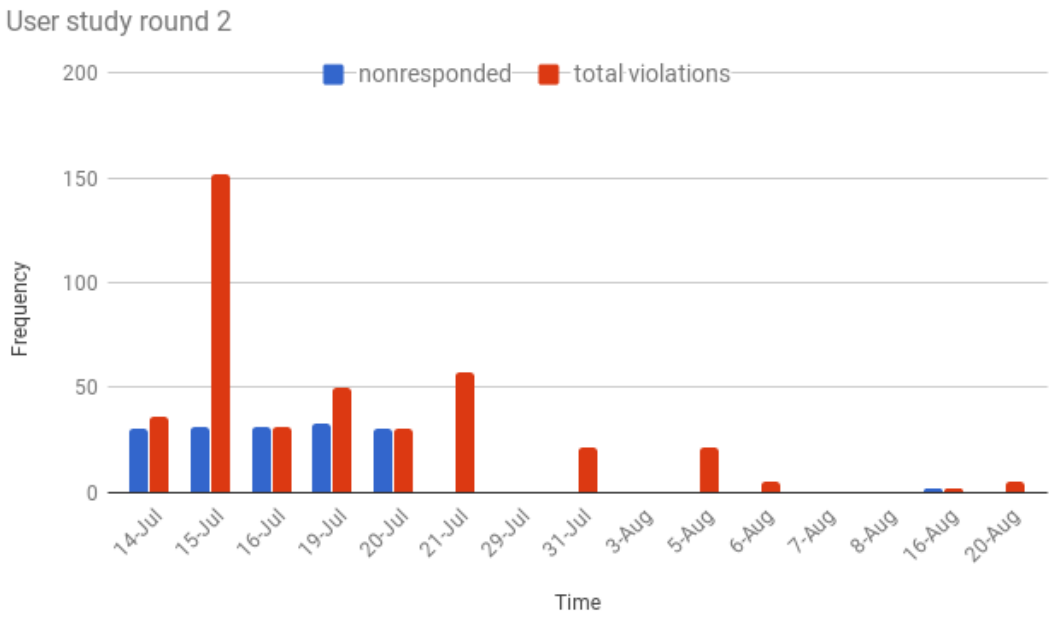

Figure 2: Comparing #violations and #no-response in user study round 2

The crowdsourced policy leads to less number of violations. A comparison of two rounds of user study shows that the number of violations went down in the second round when a curated policy was used. In contrast the first round of user study caused a lot more violations to occur. At the end of the user study users were allowed to respond to a series of questions. A likert-type scale was used for the questionnaire in order to understand the following questions/issues:

- Do mobile apps take too many permissions and the purpose unclear for a requested permission?
- Are mobile operating system security and privacy settings difficult to find?

- Are application knowledge and ratings useful in making my sharing decision?
- Is the user confident in their ability of managing mobile privacy and security settings?
- Does knowledge that an app used a resource in a specific context makes it easy to allow or deny such access?
- Are user privacy and security needs context dependent for example allowing social media access at home but not at work?
- Did the final captured policy better represent? Was using MITHRIL worth the effort given the benefits?

Users were also allowed to provide feedback using the form. Two example responses are shown below:

User 1: "As a naive android user, I did not understand all the meanings of the names of the permissions. Its hard to assess the impact of denying the permission to an app too. The option for whats allowed and what is blocked in custom controls is unclear. Semantics of the privacy setting can be more clear. There is a difficulty understanding the time of old notification. Permissions requested in prior day can nolabelFScoreMalwareDetection longer be configured since the exact time unclear. What does allowed ignore and running mean? All in all its a good starting effort and the setting requires more explanation and further ease to configure."

User 2: "Mithiril is great app and has allowed me to understand my privacy requirements better. I was surprised by the permissions by the number of permissions

that were asked by apps. For example, I am not sure why wikipedia needs my contacts, call log and calendar details. I was happy that mithiril allows context modeling. It is more useful for some apps than others. For example, I would not mind if waze asks for contact information and call log access when I am driving. But, I would not want it to access that information otherwise. Having context is helpful. I also like that number of feedbacks asked by mithiril reduced overtime. Following are few things that will make app more helpful: 1. It should have more contexts like 'Driving'. 2. Rather than having separate context window, I would prefer to add context whenever needed in customize button. It would be nice to have everything in same window, rather than having separate window for customization."

Responses to the feedback questions show that users do feel apps are intrusive and controlling while they are unsure why they have requested a permission while finding permission settings seems difficult to them. 44% users were neutral towards app ratings to decide access decisions while most of them were confident in making such decisions. Context plays a role in their access decisions and knowledge of app activity helps in making allow/deny decisions for users. Finally users felt the final policy was representative of their needs and using the system was worth the effort given the benefits.

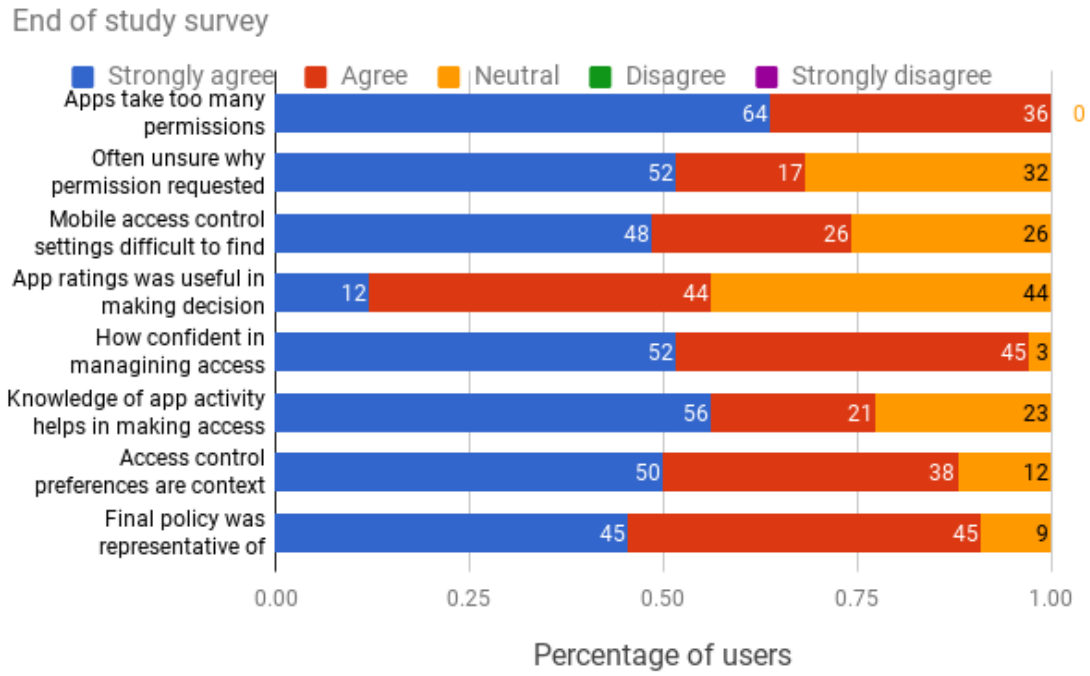

Figure 3: User feedback to questionnaire

In the next chapter we discuss our evaluation methodologies and results for MITHRIL.

Chapter 8

EVALUATIONS

In this chapter, we take a look at evaluations done as part of this dissertation. We will dive deeper into our user study and behavioral analytics for apps.

8.1 Policy capture

The policy capture process was studied using two different experimental setups. The first setup used an emulated LG Nexus 5 device running the popular ROM CynaogenMod 13, which is equivalent to Android Marshmallow 6.0. The goal was to show that it is feasible to use the Violation Metric (VM) to determine the completion of the policy capture process.

8.1.1 Automated study: experimental setup

We setup the experiment with about a 100 different policies curated by hand using various combinations of contextual situations that we envisioned. Some of our sample location context included: Home, Work, Lab, Department_Office, Classroom, Meeting_Room, Supervisor_Office. Sample activity context included: Sleeping, Dining, Traveling, Personal_Activity, Professional_Activity, Meting, Lab_Meeting, Studying, Project_Work. We also used presence info and temporal context of working and non-working hours, in our rules. As stated in our assumptions in Section 4.1.1, the

above semantic context is available to the prototype system. For simplicity of experimental setup, we used NFC tags that were programmed with contextual situations to simulate changes in context. After a context change was observed, we use an automated script to start various apps on the mobile device that would violate our default policy *P*. The examples of such violations has been discussed Section 4.2. All automated user behavior on the mobile device was created using monkeyrunner API from Android [29].

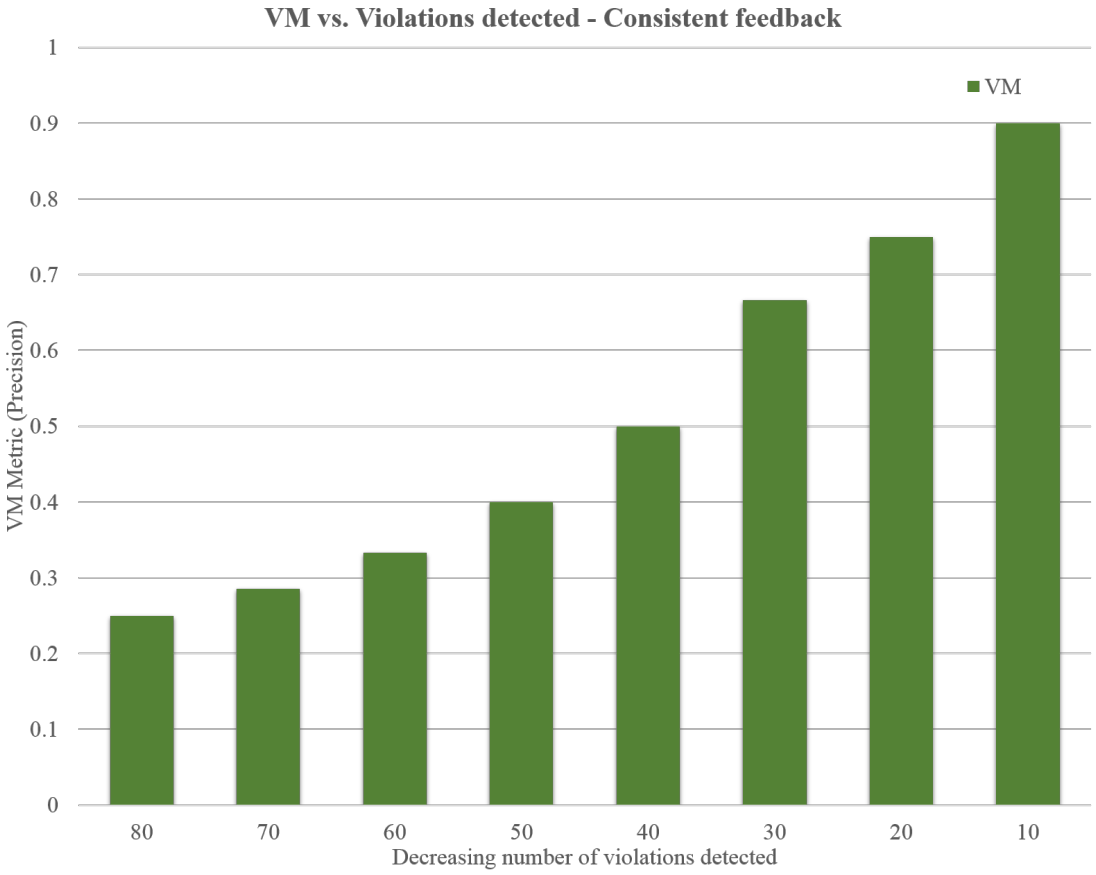

Figure 1: Consistent feedback in policy rule changes by user

8.1.2 Automated study: results

After each feedback iteration, we recorded the value of the VM metric. Take a look at the graph in Figure 1. Our simulation includes experiments that VM metrics over eight feedback iterations from our simulated user. It shows that the variation in VM metric when a user provides consistent feedback over a number of iteration cycles. You can see when the feedback is consistent, the VM metric steadily increases towards a high value. The predefined threshold we use, may be varied by a system admin. However, when MITHRIL reaches a state with high precision as determined by the VM metric, we are able to conclude that policy capture process is complete, and we may transition from OBSERVER mode to ENFORCER mode.

The VM metric initially shows a low value as a lot of policies are being modified. As iterations go by, we see a decline in the number of FALSE violations. Once a violation has been determined to be TRUE we will not require a feedback from user on that rule, anymore. As a result of that, in the graph, we see a constant decline in the number of violations and the VM metric increases in value, getting upto 0.9 by our last feedback iteration. As explained before, this means that whatever violations the system records are denoted as TRUE violations and we have captured the user's preferred policy. Understandably, the VM metric does have certain limitations when it comes to user feedback being erratic. Therefore, we decided to run a user study to better evaluate our system.

8.1.3 User study: round 1 results

In the user study, we ran our experiments on mobile devices with LineageOS 14.1.1, which is equivalent to Android 7.1.1. The study was done in two phases. One of the challenges we faced while carrying out the user study was the wide gap between the number of violations that required user input and the actual number of inputs we received. This issue occurred due to the fact that in we used an default deny initial policy. Understandably, this caused a huge number of violations. Our argument behind using a default deny policy was that if the system did not have any policy rules at all, a safe but not necessarily good policy would be to block events by default. We also wanted to test the feasibility of using the violation detection based approach to capture rules from scratch. Figure 3 shows variation of the VM metric for 10 users over a period of several days. We conclude from the first round of the user study that a quasi-safe "deny by default" policy is actually not a good policy from the perspective of usable privacy and security.

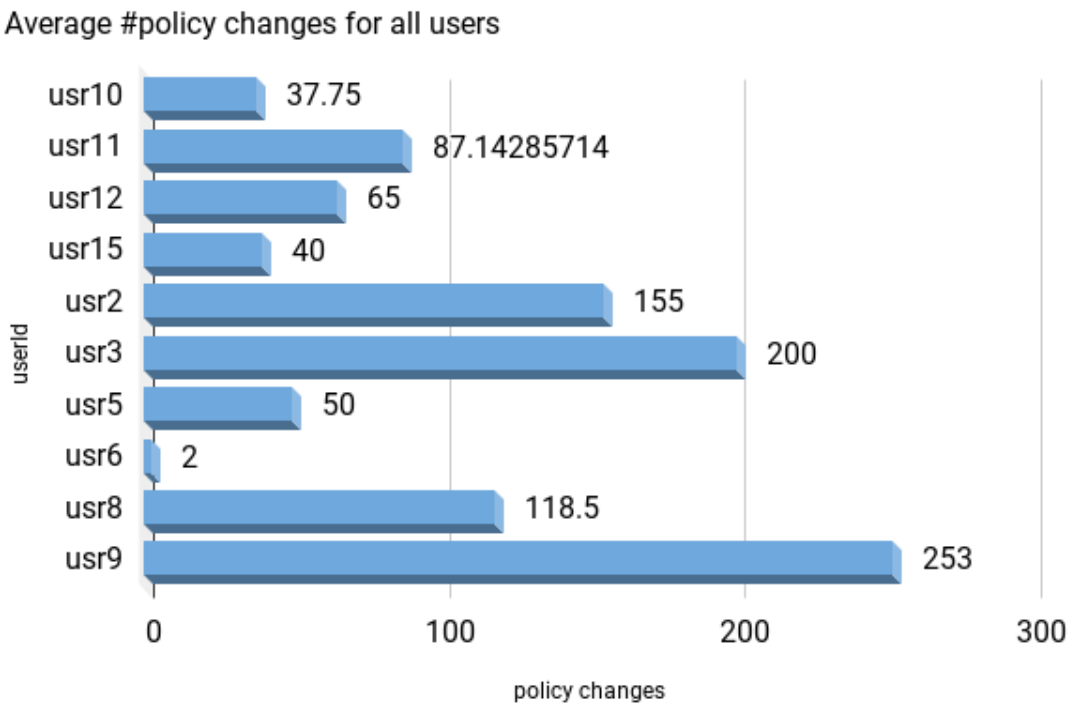

Figure 2: Average number of policy changes made per user in round 1

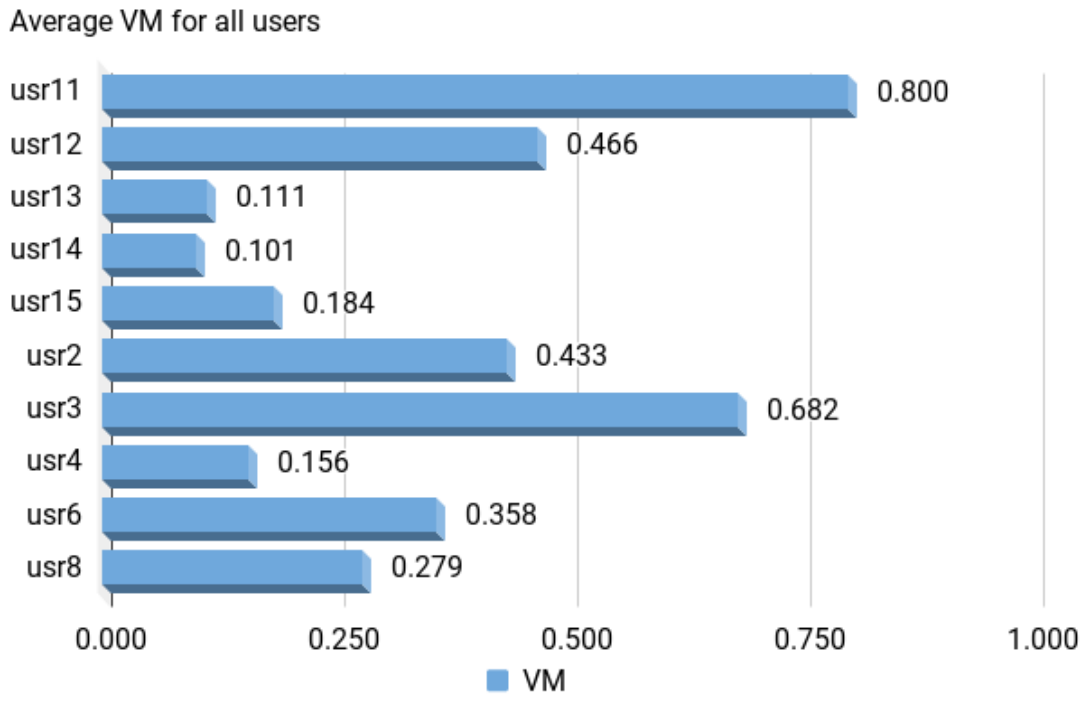

Figure 3: Average violation metric per user across multiple iterations in round 1

8.1.4 User study: round 2 results

Following the conclusions of round one of our user study we modified our methodology to include a crowd-sourced policy. Our crowd-sourced policy was originally collected by the Xprivacy open-source app [4]. The data can be found at the link: `https://crowd.xprivacy.eu/`. We downloaded approximately 21 million rules for 17k apps and then used category-wise majority voting process to create an initial default policy for the second round of our study. The crowd-sourced policy is obtained by the MithrilAC middleware by querying the back-end server. See the results of the second round of the user study in Figure 4.

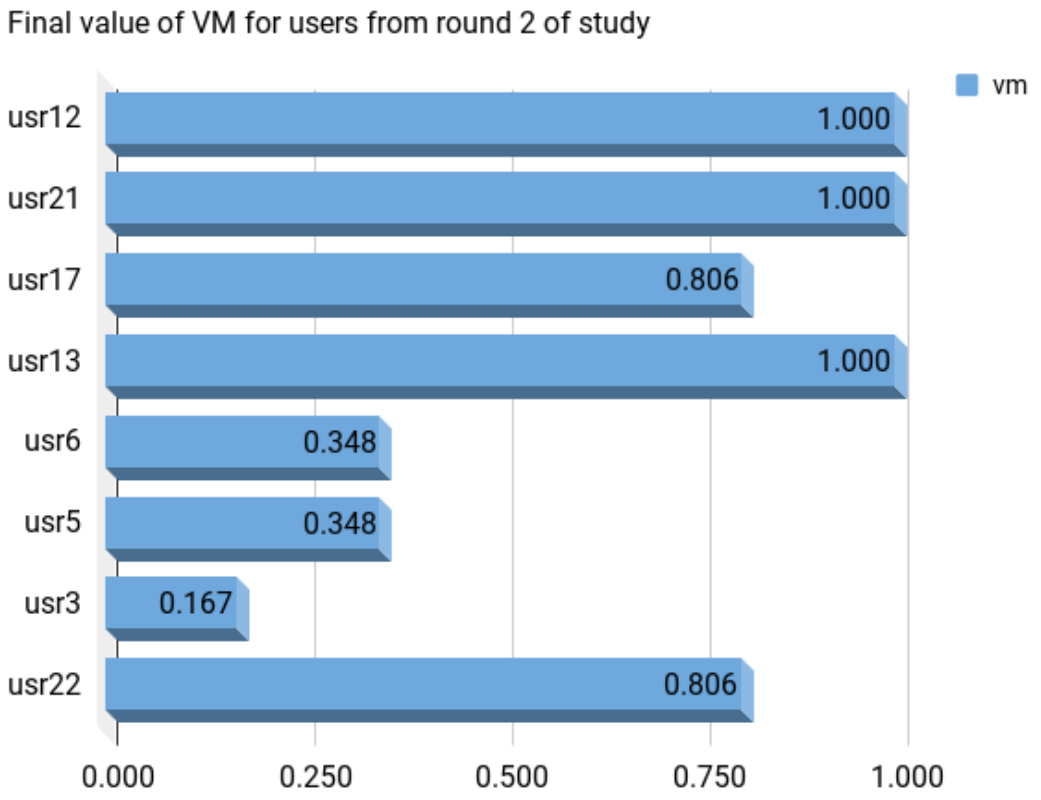

Figure 4: Average violation metric per user across multiple iterations in round 2

A comparison of number of violations and number of feedback inputs from the user can be seen in Figure 2. This graph shows that when using a default deny policy the number of violations captured were high and the number of times users did not respond to such violations in the feedback loop was also high. However, as soon as we started using the crowd-sourced policy as our initial policy generation technique, we saw a drop in number of violations and the number of times users did not respond to our queries were negligible.

8.1.5 Reduction in user interaction required

We did the user study in two rounds. We had 24 users involved in the study. All the users were graduate students from the Computer Science department at UMBC. The maximum number of policies that were created in round 1 for a user was 3200 and that number came down to 800 in round 2. The average number of apps per user in the study was 48. Table 8.1 shows the number of users, violations, true and false violations for the two rounds of user study. From the table we can observe that there is a drop in the number of violations occurring in round 2 of the user study. At the same time we also observe a favorable value for the violation metric in round 2. We discuss the significance of these results in Section 8.5.

Round	**#Users**	**#Violations**	**#True violations**	**#False violations**
1	14	778	228	550
2	10	347	300	47

Table 8.1: User study violation statistics

In round 1 we used a default deny initial policy. This led to a lot of violations and user fatigue. As can be seen in Figure 5, "no-response" count in round 1 was pretty high. As a result, we used a curated policy based off of the Xprivacy crowdsourced dataset. We use normalized frequency counts for our comparison as because the two different phases had varying number of users. The first round ended on July 13 on this chart. The second round of user study thus shows lower number of violations as well as lower number of "no-response" situation from users. The second round of the user study also show a higher number of users with high value of violation metric. This shows that

starting with a curated initial policy leads to less situations where user disagrees with the policy defined. Whereas if a default deny policy is used even legitimate usage of applications get blocked, leading to a larger number of users disagreeing with the policy and hence lower value of violation metric.

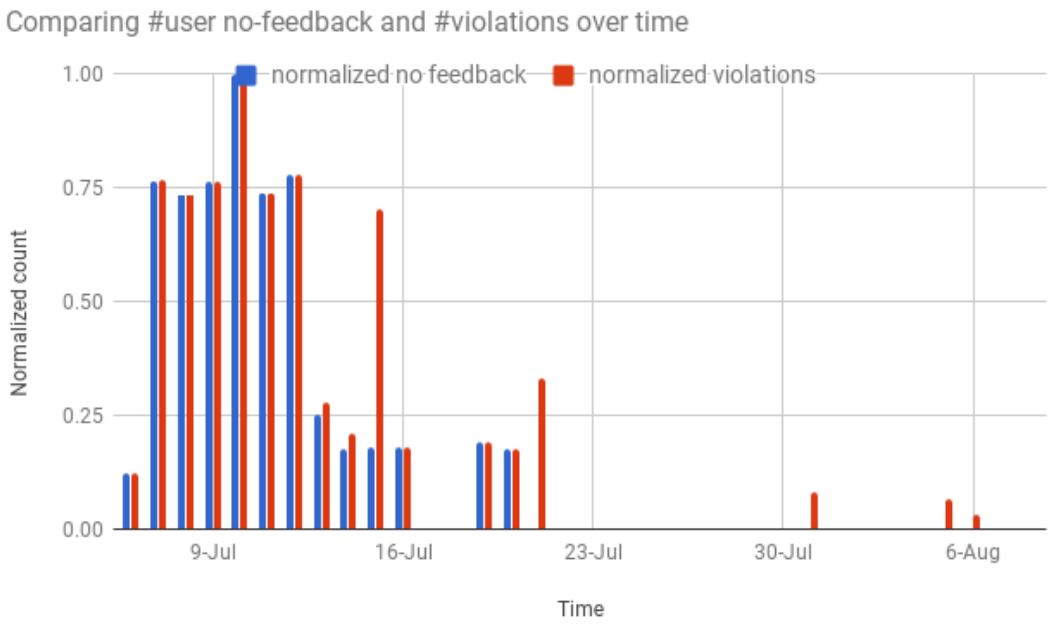

Figure 5: Comparing user no-response and violations over time

8.2 App analytics

Data set: For our app analytics back-end we experimented with several machine learning classifiers and variations of feature sets to determine the best combination of classifier and app features that would perform well for an app behavior classification task. As far as we have found, the work that is closest to ours, as in trying to perform

some kind of app behavior study was CHABADA [34]. CHABADA identifies outliers in a cluster of apps so that if a "weather" app tries to send out "messages" it would be flagged as an anomaly. However, their ultimate goal was to detect malwares using the anomaly detection technique. In our work, we are taking a slightly different approach by first classifying an app into it's behavior category. We define a policy for behavior classes using crowd-sourced data so once classified an app would have an initial default policy and if it attempts to perform an action that is inconsistent with the expected behavior, the policy would block such an action. The first step, therefore is to carry out classification of apps into behavior classes. We used system calls as features to perform our behavior classification task.

We used four different classifiers through the Weka tool [36]. We used 10 fold cross validation technique for all the classifiers. We used 10 fold cross validation technique for all the classifiers. We present the average F1–measure achieved by each of the classifiers for annotated class labels when using tf–idf weighted feature vectors in Table 8.2. F1–measure for annotated class labels when using 1-hot feature vectors are presented in Table 8.3. Unfortunately, none of the classifiers recorded a good enough F1–measure for annotated class labels.

Classifier	F1 score
MLP	0.44
SVM-RBF	0.32
SVM-Poly	0.31
J48	0.27
NB	0.27

Table 8.2: Annotated class labels, TF-IDF features

Classifier	F1 score
J48	0.31
NB	0.27
MLP	0.26
SVM-Poly	0.23
SVM-RBF	0.21

Table 8.3: Annotated class labels, one hot features

For Google's app category based class labels, average F1–measure achieved by the classifiers were low as well, as shown in Table 8.4, when using tf-idf weighted feature vectors or as shown in Table 8.5 when using one hot feature vectors.

Classifier	F1 score
SVM-Poly	0.39
SVM-RBF	0.38
MLP	0.37
J48	0.35
NB	0.14

Table 8.4: Google class labels, TF-IDF features

Classifier	F1 score
J48	0.39
MLP	0.38
SVM-Poly	0.33
SVM-RBF	0.33
NB	0.09

Table 8.5: Google class labels, one hot features

We observed that when using tf–idf weighted feature vectors we were able to achieve comparatively better classification accuracy as opposed to when using one hot vectors. Intuitively this observation makes sense since tf–idf weights better represent the significance of terms in documents as opposed to simply stating that the document has a certain term. However, a comparison of the classifiers paints a disappointing picture. While MLP performed marginally better than other algorithms, for annotated class labels using tf–idf weighted features, NB did slightly worse than other classifiers

for Google categories. In short, none of the algorithms had an outstanding performance and thus leads to the conclusion that system calls cannot be considered as good features for app behavior classification. We note that this observation was somewhat unexpected, given our understanding of system calls and application functionality correlations, from knowledge of similar methods being used successfully, in the literature [48]. As a result, we came up with two possible explanations for these observed results.

The first was that apps have functionality that belong to multiple behavior classes, for example—an app could have the functionality of social media sharing combined with financial transactions. Take for example WeChat, which has taken over workplaces in China [83]. WeChat combines instant messaging functionality with social media sharing while incorporating functions like ride hailing, buying movie tickets, sending payments, settling utility bills as well as online shopping. Such multi-functional apps, sometimes called "super apps" where apps are trying to become the "only" app on your phone by providing a multitude of functionality. This trend can best be explained by a need to retain a high active-user base, which leads to higher ad-revenue. Ad-revenues understandably are critical for an app's survival today because of the free app economy. As a result, our basic assumption that an app would serve a singular purpose no longer holds true and we need to create coarser functional clusters (i.e. "social media-financial" apps) for behavior analysis.

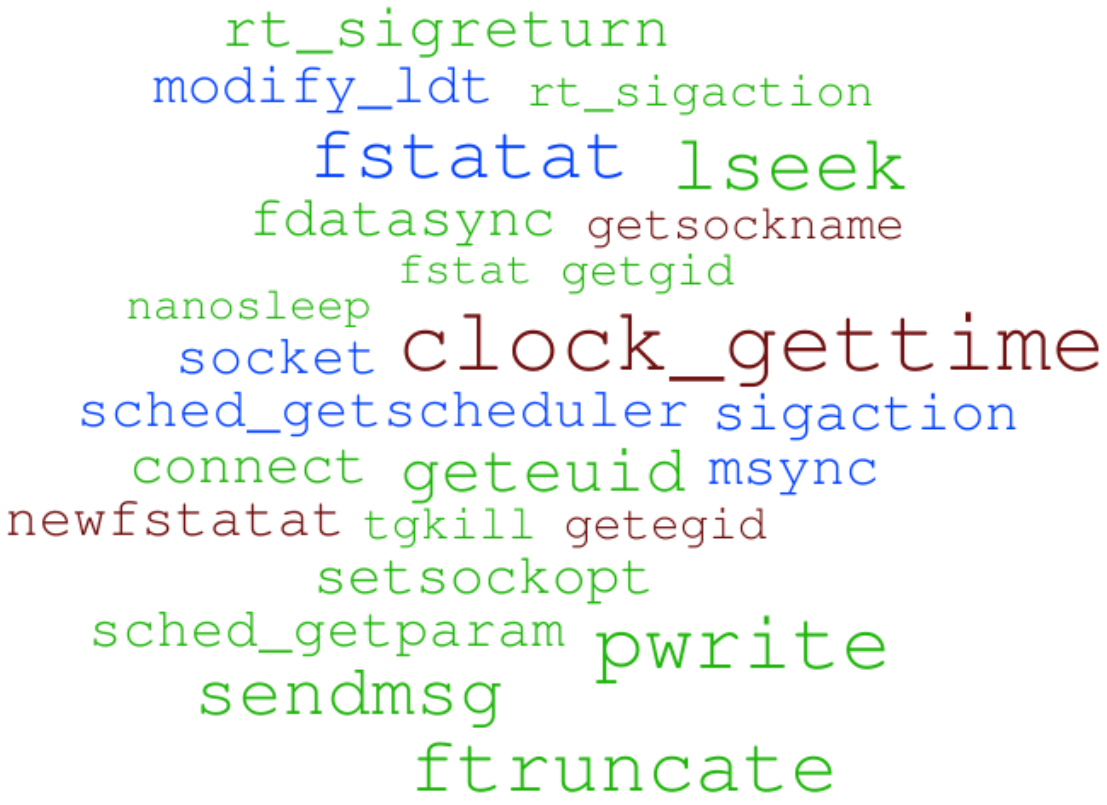

Figure 6: To do list class

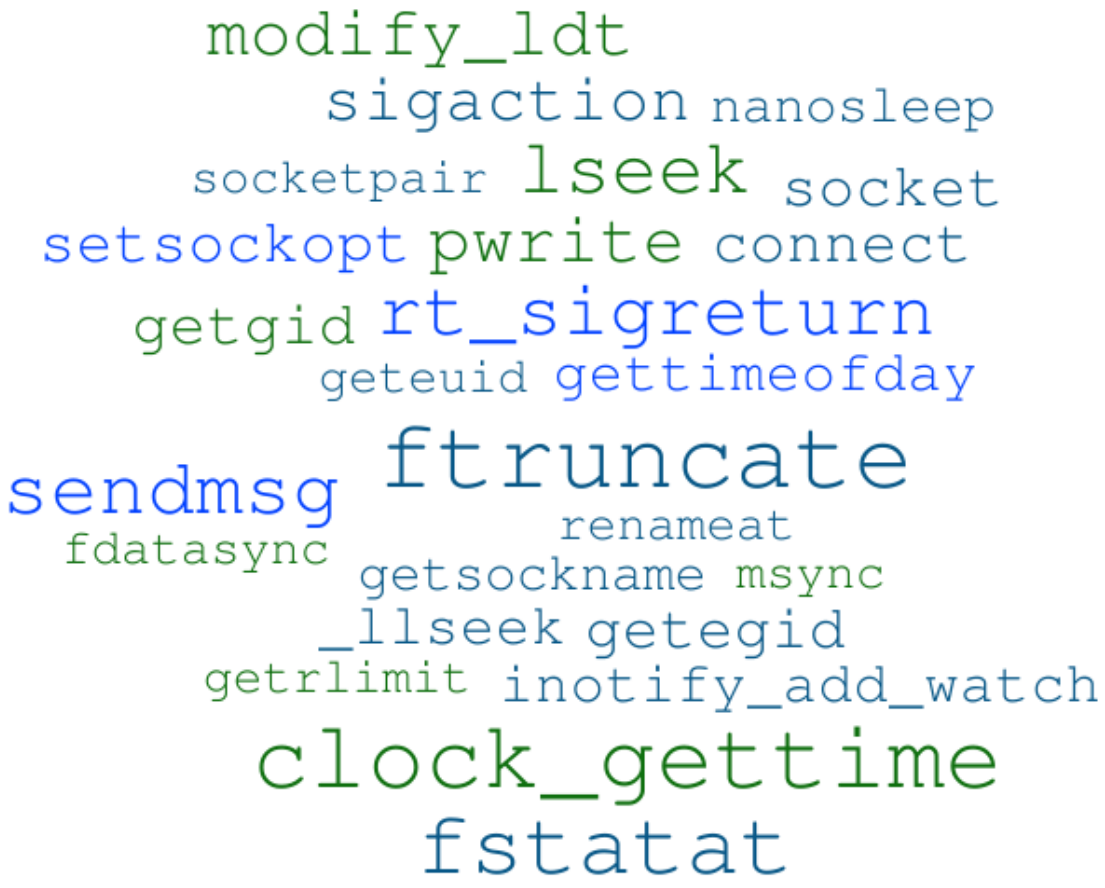

Figure 7: Scientific calculator class

The second related explanation was easier to demonstrate. We observed that since apps are trying to provide a slew of different functionality, they end up making very similar system calls. In order to investigate this further, we generated the tf-idf word clouds for each of the 10 annotated class of apps. Consider the word clouds for "To Do list" and "Scientific Calculator" shown in Figure 6 and Figure 7. We can clearly see that the "ftruncate", "fstatat" and "clock_gettime" have similar tf-idf weights for both these classes. As a results, these classes were not easily "separable" and despite the expectation that they would different behavioral patterns, were in-fact making similar

system calls.

8.3 Deeper dive into app behavior

The results of behavior classification, we have presented till now used system call uni-grams [15]. As we explained in Chapter 5, we hypothesized that system call n-gram features can possibly lead to better classification of behavioral classes. In our experiments we used uni, bi, tri, quad gram sequences. We also performed SVD on the features and used top 10, 100, 500, 1000, 2000, 5000 and 10000 features for our classification task. Both types of class labels, annotated and Google categories were used. We performed the deep dive into behavior classification with TF-IDF weight vectors. This was because when we did not use n-gram features, which is the same as using uni-gram features and all feature vectors the precision and recall achieved with 1-hot vectors or system call frequency vectors were lower than TF-IDF weight vectors (see Figure 8, Figure 9, Figure 10, and Figure 11).

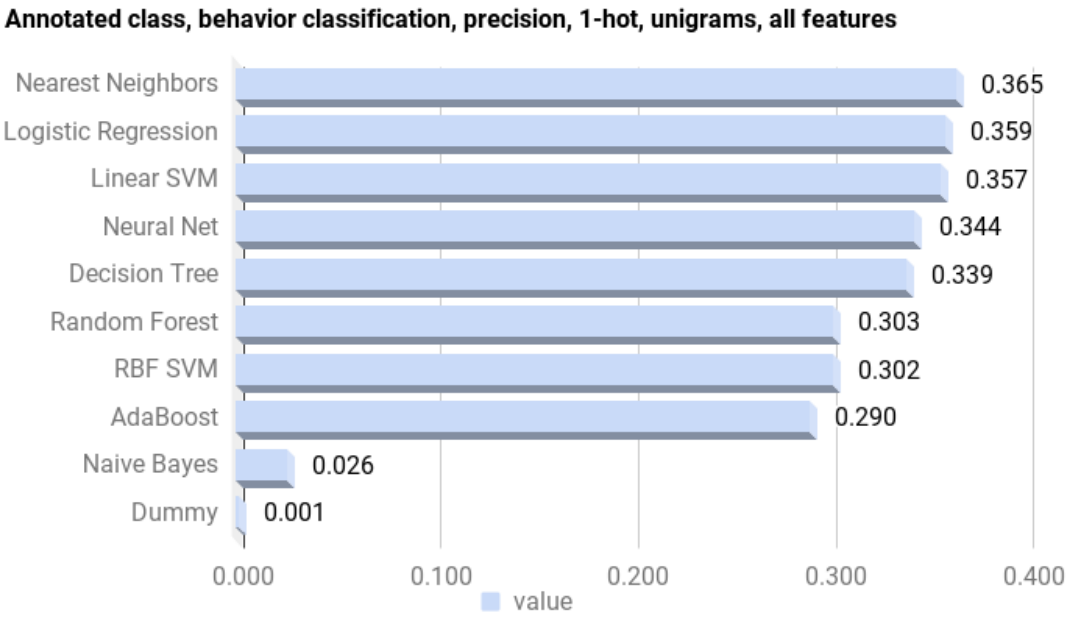

Figure 8: Best possible precision for Annotated class labels using 1-hot features and Uni-gram model

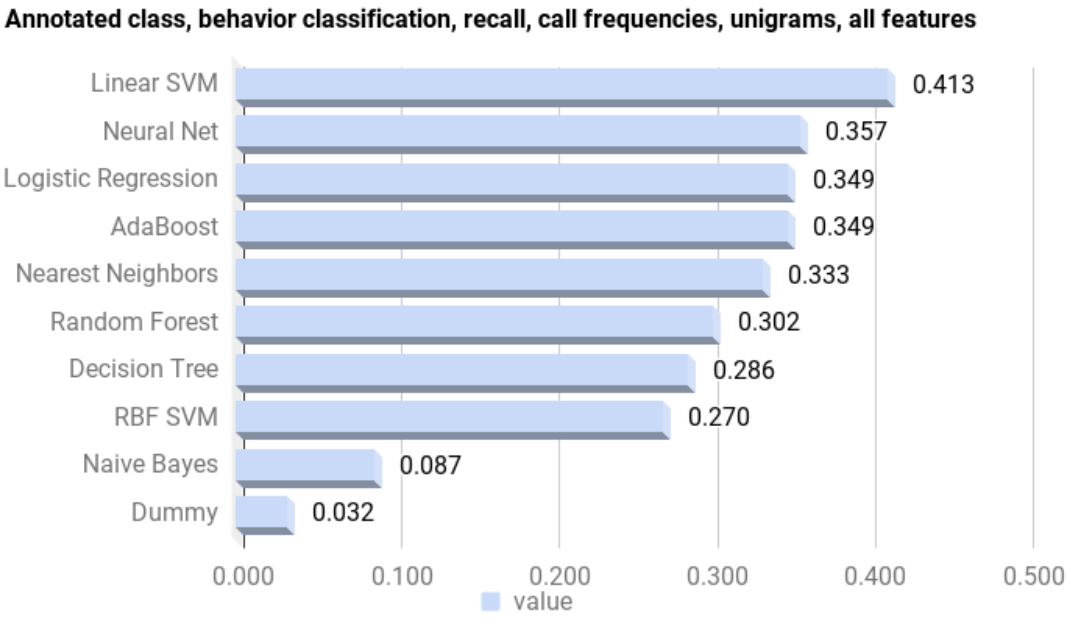

Figure 9: Best possible recall for Annotated class labels using call frequency features and Uni-gram model

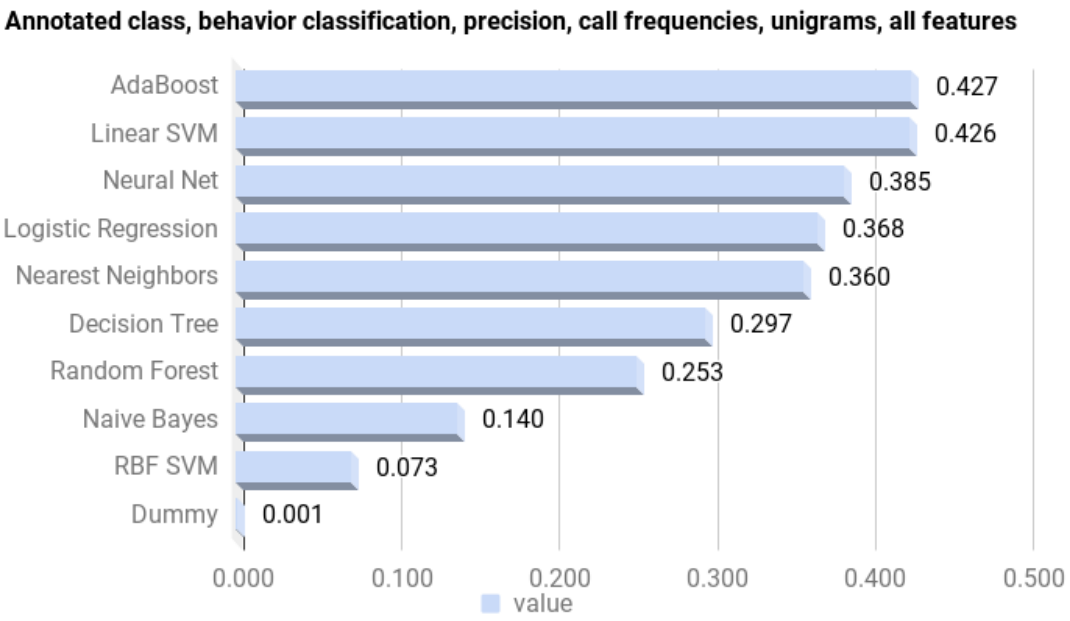

Figure 10: Best possible precision for Annotated class labels using call frequency features and Uni-gram model

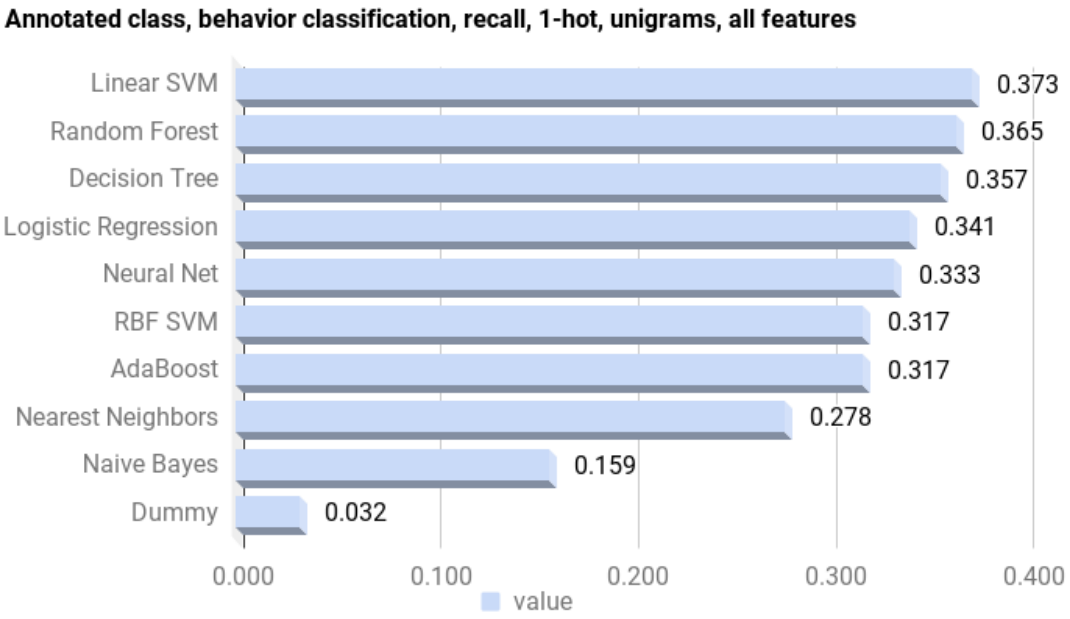

Figure 11: Best possible recall for Annotated class labels using 1-hot features and Unigram model

For the deeper dive and for the sake of automating our classification tasks, we used the python Sci-Kit Learn library [65]. We used the k nearest neighbors (with 3 neighbors), Linear SVM, SVM with RBF kernel, Decision trees, Random forest, Ada boost, Naive Bayes, Logistic regression and Neural Net classifiers. We also used a dummy classifier with most frequent as the option to baseline our classification results. Additionally, since we were carrying out a multi-class classification task, we used the One-vs-the-rest (OvR) multiclass strategy from Sci-Kit Learn [6]. For our own annotated classes the best precision and recall was achieved with a bi-gram model and Nearest Neighbors classifier (see Figure 12 for precision and Figure 13 for recall).

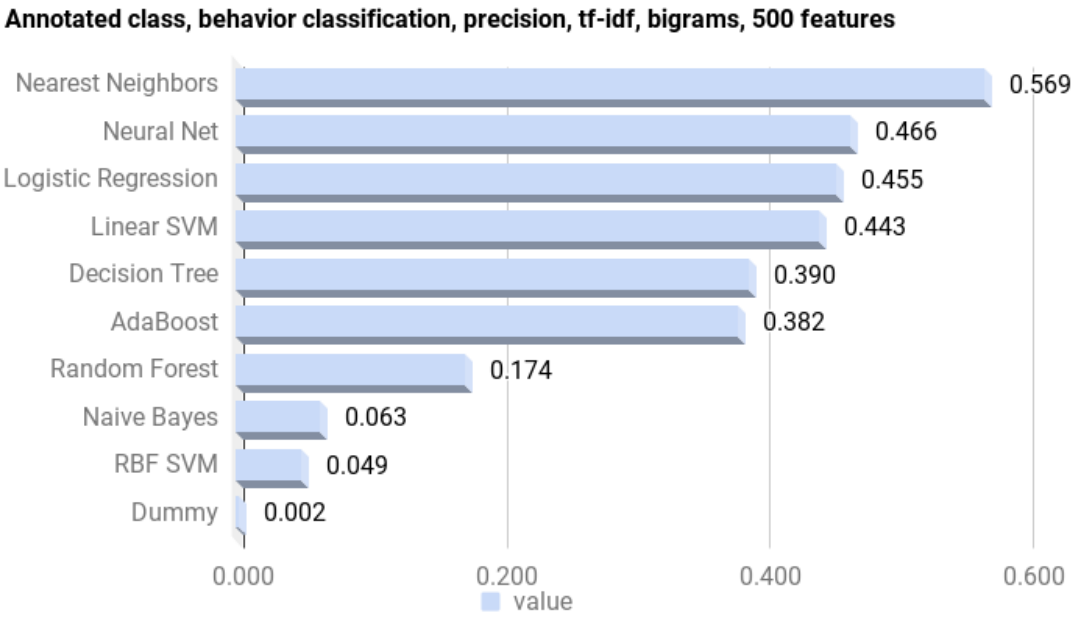

Figure 12: Best possible precision for Annotated class labels using TF-IDF features and Bi-gram model

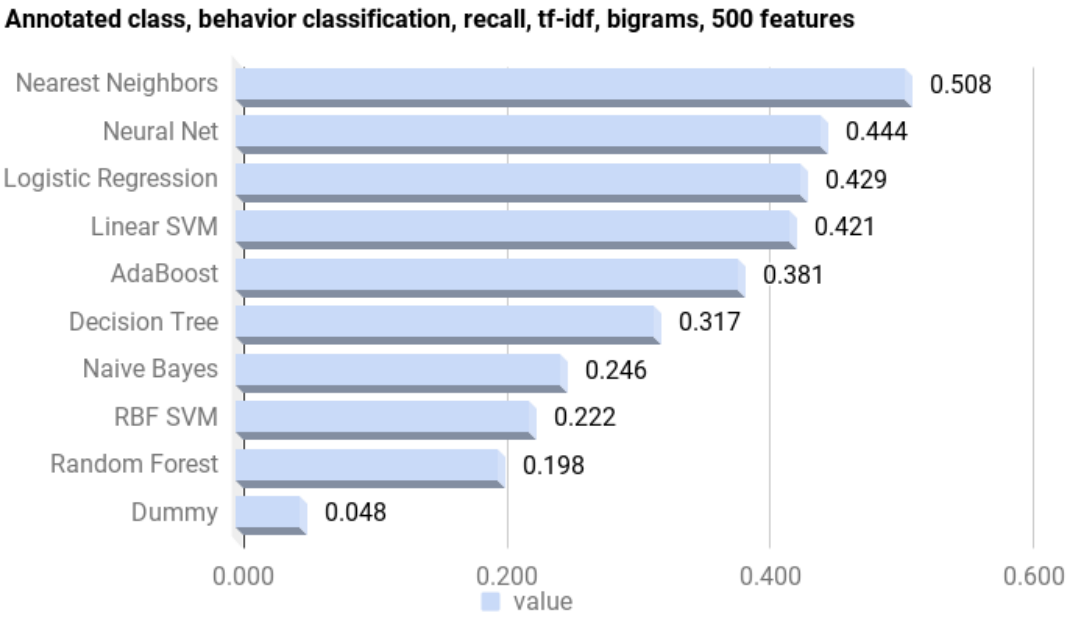

Figure 13: Best possible recall for Annotated class labels using TF-IDF features and Bi-gram model

For Google Play Store categories the best precision and recall was achieved with a unigram model and Neural Network classifier (see Figure 14 for precision and Figure 15 for recall).

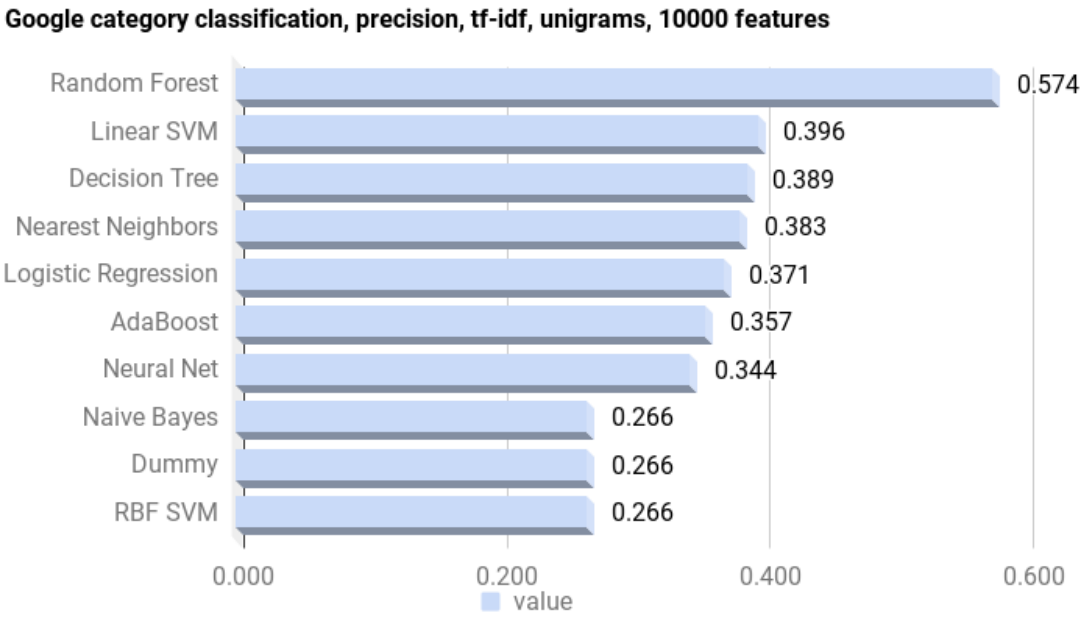

Figure 14: Best possible precision for Google categories using TF-IDF features and Uni-gram model

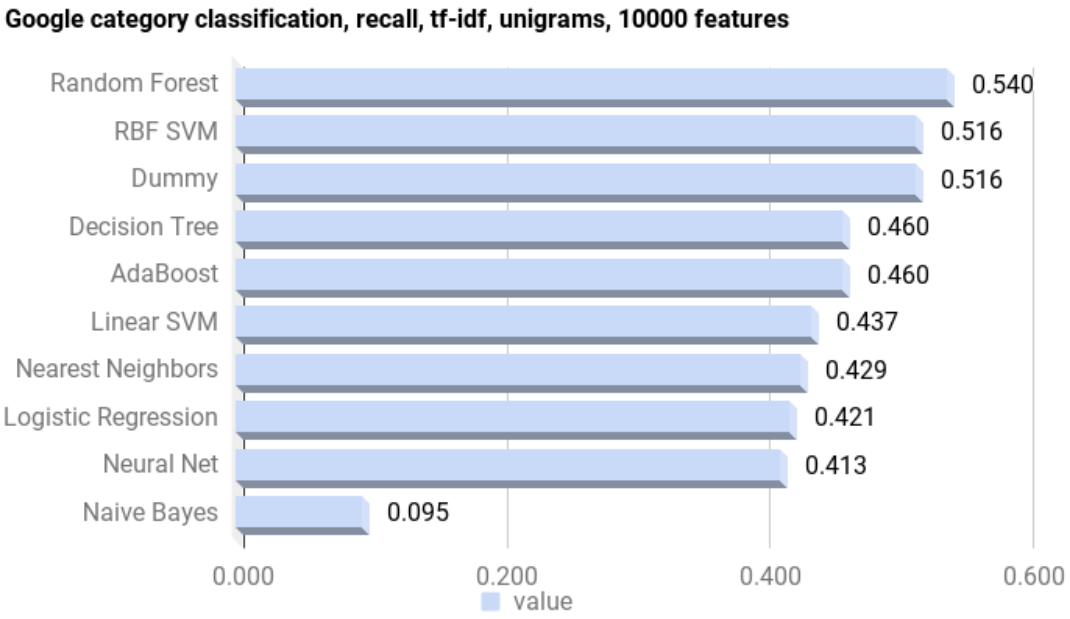

Figure 15: Best possible recall for Google categories using TF-IDF features and Unigram model

We also carried out application behavior classification using static features and found that permission features perform better at this task. Results of the same can be seen in Figure 16.

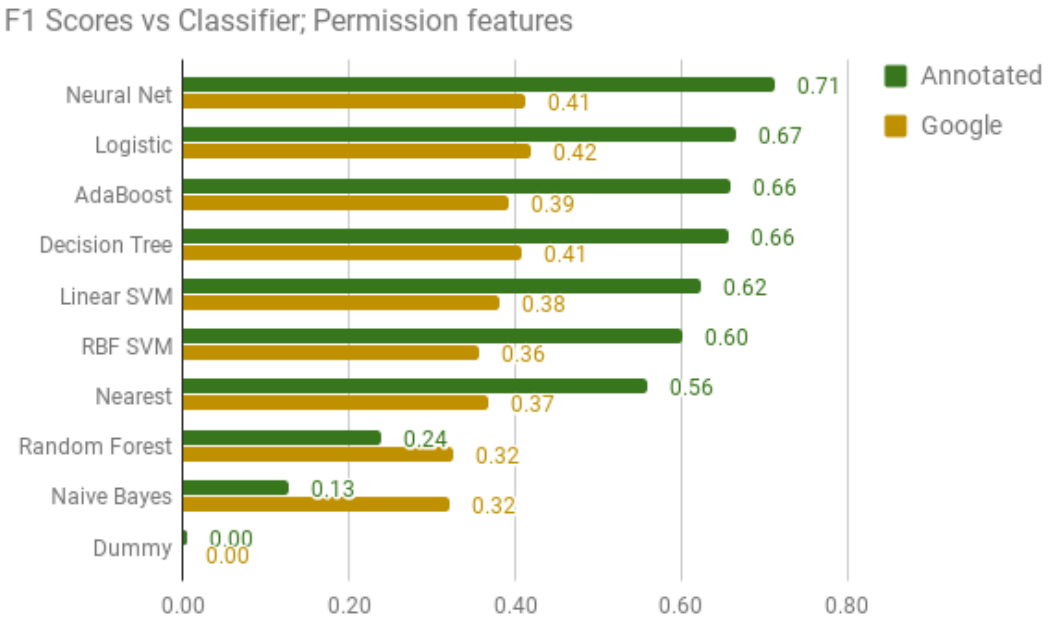

Figure 16: Static features like permissions perform better application behavior classification

8.4 Malware detection

Malware detection have has a fair bit of success till date. We study malware detection with two goals in mind. The first goal was to determine if permissions can be used as features to detect malware apps. We are not trying to beat the state-of-the-art in malware detection merely looking at the feasibility of using permissions as features to detect malwares. We create a feature importance based model created from the malware detection task for risk computation for apps. Results of malware detection can be seen in Figure 17.

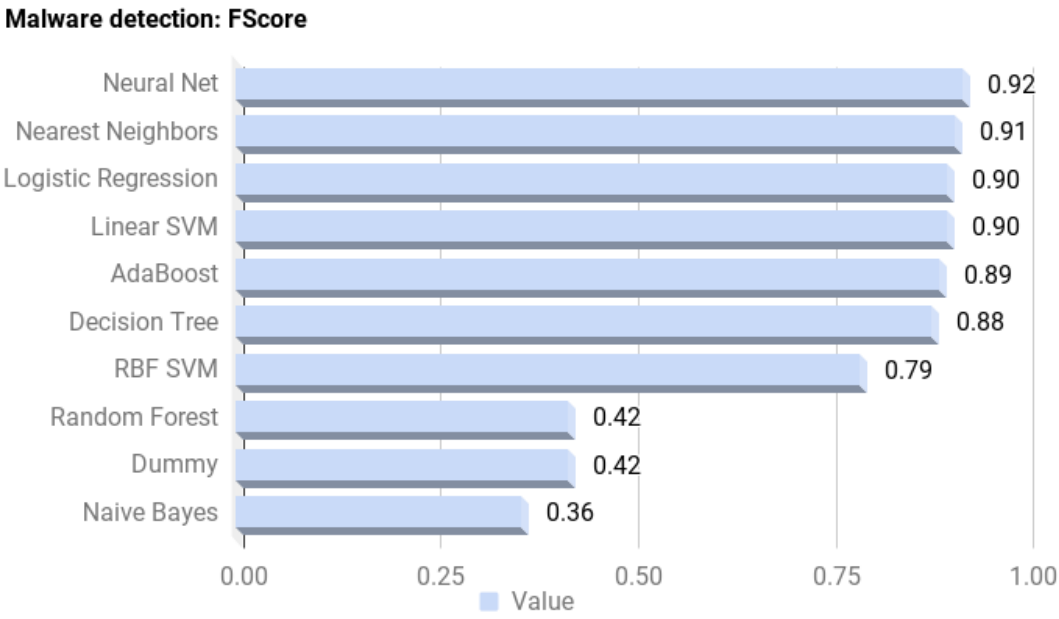

Figure 17: F1 - scores for malware detection using 10 classifiers

8.4.1 Risk computation using feature importance

The second goal of the malware detection task was to determine if there are permissions that are frequently used by malware apps. Feature importance can help in determining permissions most relevant to malware detection. We used these values to compute the risk associated with 100 applications from malware and benign app categories. Using just the top 100 important permissions and their computed importance rank values we were able to detect malwares with high accuracies.

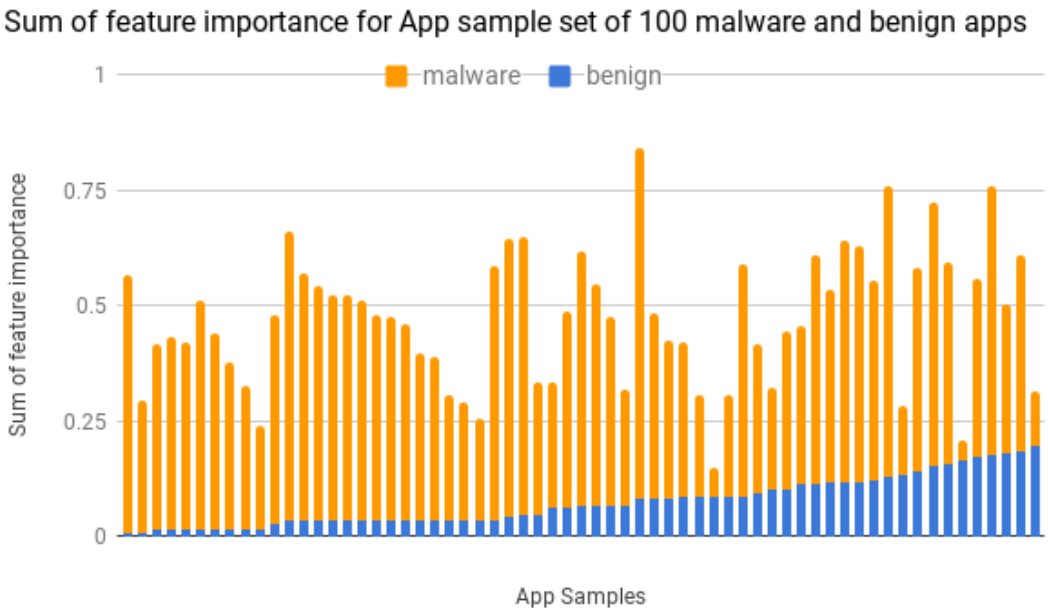

Figure 18: Feature importance can be used to determine malware

Investigating the top features for the malware detection task leads to the conclusion that the following are strong indicators of an application being a malware:

- Access running applications
- Access wifi state
- Change wifi state
- See what's on screen
- Mount filesystems
- Relaunch applications

Figure 19 shows the features that are important to the malware detection task based on feature importance:

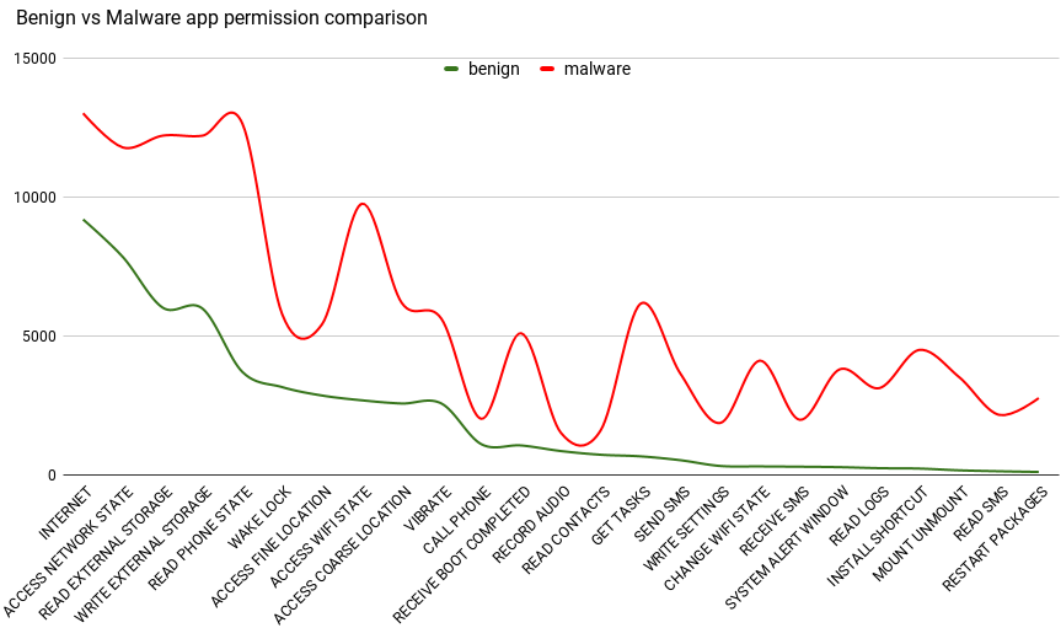

Figure 19: Features important in detecting malware

8.5 Discussion: Statistical significance

We have presented our results for the user study and application analytics till now. Next we discuss the significance of our results under varying treatments. We performed the user study under two different treatments. Under the first treatment we used a default deny policy and under the second treatment we used a curated policy generated using crowd-sourced data. Since these two treatments were performed on different sets of users a paired sample test cannot be performed. We performed an unpaired T-test with

the null hypothesis that the two different treatments had the same mean violations and non-responsive situation for users. We were able to reject the null hypothesis through our test, since the computed p value was 0.0033.

For our application behavior and malware detection tasks we were able to carry our paired sample T-tests. We compared the top two machine learning algorithms as per the F-scores with a null hypothesis that the misclassification error for the two algorithms had the same mean. We were able to reject the null hypothesis for application behavior classification with a p value at 0.000013 and for the malware detection task the p value was 0.00001.

In the next chapter we present the conclusions of this dissertation.

Chapter 9

CONCLUSION

In this dissertation, we have created an end-to-end context-dependent access control framework. We presented an approach that combines analysis of mobile application behavior with on-device mobile application monitoring to create semi-automated approaches to capture rules representing context-dependent, fine-grained privacy and security policy for a user. The key contributions of this dissertation include:

- Usage of the "Violation Metric" measure as a means for determining completion of the policy capture process
- Usage of "Application Behavior" for creating curated initial default policies as a starting point for the policy capture process
- Feature importance for malware detection as a measure of risk associated with applications

Furthermore, as part of this dissertation, we created an end-to-end context-dependent access control approach by monitoring policy violations on users' mobile devices. We were able to reduce the amount of interaction required to capture user-specific access control policies by generating initial policies through a mobile application analysis back-end. We combined crowd-sourced data with application behavior knowledge to generate the initial policies. We conducted a user study to show the feasibility of using

our violation metric and policy capture method in creating user-specific access control policies. We enriched the Mobipedia [62] KB with mobile application behavioral facts and enhanced presence context detection using nearby messages and beacons. In order to carry out the user study we built a modified Android operating system that allowed us to perform the monitoring. We used custom ROMs for that purpose.

Some key conclusions drawn as a result of our research and experiments are as follows:

- User study conducted proves that using a default deny policy creates too many policy violations and causes user fatigue leading to low number of responses.
- Using a crowd-sourced or curated initial default policy reduces user burden for policy customization.
- It is fair to use system calls to model mobile application behavior and to use such a model to create an initial default policy for users.
- Static permission features do improve classification accuracies for application behavior.
- Permissions requested by mobile applications can be used to detect malware applications.
- Permission based malware classification task allows computation of risks associated with applications using feature importances.
- We have studied application activities that were blocked by a policy but allowed by the operating system.

9.1 Future Work

One of the most difficult task that we had to perform was to get a working system that captures actual events on a mobile device and feedback from real users about those events. Given our collected data, an obvious future work is to use pattern recognition and machine learning algorithms to predict a user's preference choices. Improvements to the behavior classification process is an important goal that we hope to pursue in the future. Since static features improved classification accuracies combining them with dynamic and network features could lead to better application behavior classifications. Some of the suggestions made by users in our study could further improve the usability of the system. We hope to incorporate these in the future. Another potential future work includes building a policy that blocks applications that are determined to be "too unsafe" through the risk computation. Finally, we would like to perform study of things that were allowed in policy and blocked by the operating system.

Bibliography

[1] Subhendu Aich, Samrat Mondal, Shamik Sural, and Arun K Majumdar. Role Based Access Control with Spatiotemporal Context for Mobile Applications. *Transactions on Computational Science*, 4:177–199, 2009.

[2] Michael Benisch, Patrick Gage Kelley, Norman Sadeh, and Lorrie Faith Cranor. Capturing location-privacy preferences: Quantifying accuracy and user-burden tradeoffs. *Personal and Ubiquitous Computing*, 15(7):679–694, 2011.

[3] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):154–165, 2009.

[4] Marcel Bokhorst(M66B). Xprivacy, June 2013.

[5] Andrey Boytsov and Arkady Zaslavsky. From sensory data to situation awareness - Enhanced context spaces theory approach. *Proceedings - IEEE 9th International Conference on Dependable, Autonomic and Secure Computing, DASC 2011*, pages 207–214, 2011.

[6] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[7] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[8] Dave Chaffey. Mobile marketing statistics compilation, March 2017.

[9] Robert Charette. Smartphone app developers being criminally investigated over privacy issues?, April 2011.

[10] Harry Chen, Tim Finin, and Anupam Joshi. An ontology for context-aware pervasive computing environments. *The knowledge engineering review*, 18(03):197–207, 2003.

[11] Harry Chen, Tim Finin, and Anupam Joshi. The SOUPA Ontology for Pervasive Computing. *Computing Systems*, pages 233–258, 2005.

[12] PENG Chen, ZHAO Rong-Cai, Shan ZHENG, XUN Jia, and YAN Li-Jing. Android malware of static analysis technology based on data mining. *DEStech Transactions on Computer Science and Engineering*, 2016.

[13] Shigeru Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336. Springer, 2000.

[14] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In Mike Burmester, Gene Tsudik, Spyros Magliveras, and Ivana Ilic, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 331–345. Springer Berlin Heidelberg, 2011.

[15] Prajit Kumar Das, Anupam Joshi, and Tim Finin. App behavioral analysis using system calls. *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS): MobiSec 2017: Security, Privacy, and Digital Forensics of Mobile Systems and Networks (INFOCOM17 WKSHPS MobiSec 2017)*, 2017.

[16] Prajit Kumar Das, Abhay Kashyap, Gurpreet Singh, Cynthia Matuszek, Tim Finin, and Anupam Joshi. Semantic Knowledge and Privacy in the Physical Web. *Proceedings of the 4th Workshop on Society, Privacy and the Semantic Web - Policy and Technology (PrivOn 2016) co-located with the 15th International Semantic Web Conference (ISWC 2016)*, ISWC 2016, 2016.

[17] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference. *W3C Recommendation February*, 10, 2004.

[18] Anind K. Dey and Gregory D. Abowd. Towards a better understanding of context and context-awareness. In *First Int. symposium on Handheld and Ubiquitous Computing (HUC)*, 1999.

[19] William Enck. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security*, ICISS'11, pages 49–70, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] 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*, pages 1–6, 2010.

[21] Riccardo Falco, Aldo Gangemi, Silvio Peroni, David Shotton, and Fabio Vitali. Modelling OWL ontologies with Graffoo. In *11th Extended Semantic Web Conference (ESWC)*, pages 320–325, 2014.

[22] Denzil Ferreira, Vassilis Kostakos, and Anind K Dey. Aware: mobile context instrumentation framework. *Frontiers in ICT*, 2:6, 2015.

[23] Tim Finin, Anupam Joshi, Lalana Kagal, Jianwei Niu, Ravi Sandhu, William Winsborough, and Bhavani Thuraisingham. R owl bac: representing role based access control in owl. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 73–82. ACM, 2008.

[24] Christian Fritz and Steven Arzt. Highly precise taint analysis for android applications. *Ec Spride, Tu . . .* , 2013.

[25] Johannes Fürnkranz, Dragan Gamberger, and Nada Lavrač. *Foundations of rule learning*. Springer, 2012.

[26] Dibyajyoti Ghosh. Context based privacy and security in smartphones. Master's thesis, University of Maryland, Baltimore County, 2012.

[27] Dibyajyoti Ghosh, Anupam Joshi, Tim Finin, and Pramod Jagtap. Privacy control in smart phones using semantically rich reasoning and context modeling. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 82–85, 2012.

[28] Google. Android apps on google play, January 2015.

[29] Google. The monkeyrunner api, October 2016.

[30] Google. Introduction to android, January 2017.

[31] Google. Nearby messages api, May 2017.

[32] Google. Os: Access to low-level system functionality, January 2017.

[33] Google. Ui/application exerciser monkey, January 2017.

[34] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1025–1035, 2014.

[35] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *Communication Networks and Distributed Systems Modeling and Simulation Conf. (CNDS)*, 2004.

[36] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[37] Tin Kam Ho. Random decision forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995.

[38] Russell Holly. Cyanogen os privacy guard keeping apps from seeing your, May 2015.

[39] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[40] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al. Guide to attribute based access control (abac) definition and considerations (draft). *NIST Special Publication*, 800(162), 2013.

[41] Pramod Jagtap, Anupam Joshi, Tim Finin, and Laura Zavala. Preserving privacy in context-aware systems. In *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*, pages 149–153, Sept 2011.

[42] Pramod Jagtap, Anupam Joshi, Tim Finin, and Laura Zavala. Privacy preservation in context aware geosocial networking applications. *organization*, 2011.

[43] Xin Jin, Ram Krishnan, and Ravi S Sandhu. A unified attribute-based access control model covering dac, mac and rbac. *DBSec*, 12:41–55, 2012.

[44] Lalana Kagal and T Berners-Lee. Rein: Where policies meet rules in the semantic web. *Computer Science and Artificial . . .* , 2005.

[45] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 63–74. IEEE, 2003.

[46] Michael Kerrisk. syscalls - linux system calls, December 2016.

[47] Jay Mayfield Kerry O'Brien, Sarah Schroeder. Ftc approves final order settling charges against flashlight app creator, December 2013.

[48] Andrew P Kosoresow and Steven A Hofmeyr. Intrusion detection via system call traces. *IEEE software*, 14(5):35, 1997.

[49] Ponnurangam Kumaraguru and Lorrie Faith Cranor. Privacy indexes: a survey of westin's studies. *School of Computer Science, Carnegie Mellon University, Pittsburgh*, 2005.

[50] Kangjae Lee, Jiyeong Lee, and Mei-Po Kwan. Location-based service using ontology-based semantic queries: A study with a focus on indoor activities in a university context. *Computers, Environment and Urban Systems*, 62:41 – 52, 2017.

[51] Jialiu Lin, Jason I. Hong, Bin Liu, Norman Sadeh, and Jason I. Hong. Modeling Users ' Mobile App Privacy Preferences : Restoring Usability in a Sea of Permission Settings. *Proceedings of the tenth Symposium on Usable Privacy and Security*, 1:1–14, 2014.

[52] Jialiu Lin, Bin Liu, Norman Sadeh, and Jason I. Hong. Modeling users' mobile app privacy preferences: Restoring usability in a sea of permission settings. In *Symposium On Usable Privacy and Security (SOUPS 2014)*, pages 199–212, Menlo Park, CA, July 2014. USENIX Association.

[53] Bin Liu, Jialiu Lin, and Norman Sadeh. Reconciling mobile app privacy and usability on smartphones: Could user privacy profiles help? In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 201–212, New York, NY, USA, 2014. ACM.

[54] Teena Maddox. Research: 74 percent using or adopting byod, January 2015.

[55] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[56] mateor. Openpdroid, January 2013.

[57] Pablo N Mendes, Max Jakob, and Christian Bizer. Dbpedia: A multilingual cross-domain knowledge base. In *LREC*, pages 1813–1817, 2012.

[58] Trudy Muller and Alex Kirschner. Apple celebrates one billion iphones, July 2016.

[59] Suman Nath. Ace: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[60] Bob Pan. dex2jar, 2015.

[61] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 527–542, Berkeley, CA, USA, 2013. USENIX Association.

[62] Primal Pappachan, Roberto Yus, Prajit Kumar Das, Sharad Mehrotra, Tim Finin, and Anupam Joshi. Building a mobile applications knowledge base for the linked data cloud. In *MoDeST@ ISWC*, pages 14–25, 2015.

[63] Primal Pappachan, Roberto Yus, Prajit Kumar Das, Sharad Mehrotra, Tim Finin, and Anupam Joshi. Mobipedia: Mobile applications linked data. In *International Semantic Web Conference (Posters & Demos)*, 2015.

[64] Bill Parducci, Hal Lockhart, and Erik Rissanen. Extensible access control markup language (xacml) version 3.0. *OASIS Standard*, pages 1–154, 2013.

[65] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[66] The Linux Information Project. Kernel definition, May 2005.

[67] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1):44–53, 2015.

[68] Mengyu Qiao, Andrew H. Sung, and Qingzhong Liu. Merging permission and api features for android malware detection. *Proceedings - 2016 5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016*, pages 566–571, 2016.

[69] J-Michael Roberts. Virus share, 2014.

[70] rovo89. Xposed, August 2014.

[71] Norman Sadeh, Jason Hong, Lorrie Cranor, Ian Fette, Patrick Kelley, Madhu Prabaker, and Jinghai Rao. Understanding and capturing people's privacy policies in a mobile social networking application. *Personal Ubiquitous Comput.*, 13(6):401–412, August 2009.

[72] Saguna Saguna, Arkady Zaslavsky, and Dipanjan Chakraborty. Complex activity recognition using context-driven activity theory and activity signatures. *ACM Trans. Comput.-Hum. Interact.*, 20(6):32:1–32:34, December 2013.

[73] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.

[74] Google Android Security. The google android security teams classifications for potentially harmful applications, April 2016.

[75] Intel Security. Mcafee lbs: Threats report, November 2014.

[76] Karina Sokolova, Charles Perez, and Marc Lemercier. Android application classification and anomaly detection with graph-based permission patterns. *Decision Support Systems*, 93:62–76, 2015.

[77] Statista. Number of available applications in the google play store from december 2009 to june 2017, June 2017.

[78] Statista. Number of available apps in the apple app store from july 2008 to january 2017, 2017.

[79] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. *Ndss*, pages 8–11, February 2015.

[80] Andrzej Uszok, Jeffrey M. Bradshaw, and Renia Jeffers. KAoS: A Policy and Domain Services Framework for Grid Computing and Semantic Web Services. *Trust Management –Lecture Notes in Computer Science*, 2995/2004:16–26, 2004.

[81] CJ Van Rijsbergen. Information retrieval. dept. of computer science, university of glasgow. *URL: citeseer. ist. psu. edu/vanrijsbergen79information. html*, 1979.

[82] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *The 2014 ACM international conference on Measurement and modeling of computer systems - SIGMETRICS '14*, pages 221–233, 2014.

[83] Yue Wang. Tencent's 'super app' wechat is quietly taking over workplaces in china, August 2016.

[84] Stuart Weibel, John Kunze, Carl Lagoze, and Misha Wolf. Dublin core metadata for resource discovery. RFC 2413, RFC Editor, September 1998.

[85] Waskitho Wibisono, Arkady Zaslavsky, and Sea Ling. Situation-awareness and reasoning using uncertain context in mobile peer-to-peer environments. *International Journal of Pervasive Computing and Communications*, 9(1):52–71, 2013.

[86] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1):114–123, Feb 2016.

[87] Roberto Yus, Carlos Bobed, Guillermo Esteban, Fernando Bobillo, and Eduardo Mena. Android goes semantic: Dl reasoners on smartphones. In *Ore*, pages 46–52, 2013.

[88] Laura Zavala, Radhika Dharurkar, Pramod Jagtap, Tim Finin, and Anupam Joshi. Mobile, collaborative, context-aware systems. In *Proc. AAAI Workshop on Activity Context Representation: Techniques and Languages, AAAI. AAAI Press*, 2011.

[89] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, May 2012.